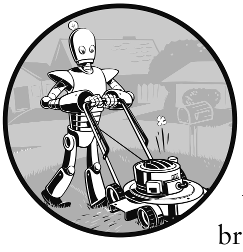
1

Grundlagen von Python



Die Programmiersprache Python bietet eine breite Palette von syntaktischen Konstruktionen, Standardbibliotheksfunktionen und Möglichkeiten zur interaktiven Entwicklung. Zum Glück brauchen Sie sich um das meiste davon nicht zu kümmern, sondern müssen nur so viel lernen, dass Sie damit praktische kleine Programme schreiben können.

Allerdings müssen Sie, bevor Sie irgendetwas tun können, zunächst einige Grundlagen der Programmierung erlernen. Wie ein Zauberlehrling werden Sie vielleicht denken, dass einige dieser Grundlagen ziemlich undurchsichtig sind und dass es viel Mühe macht, sie sich anzueignen, aber diese Kenntnisse und etwas Übung werden Sie in die Lage versetzen, Ihren Computer wie einen Zauberstab zu nutzen und damit unglaublich erscheinende Dinge zu tun.

In einigen Beispielen in diesem Kapitel werden Sie dazu aufgefordert, etwas in die *interaktive Shell*, auch *REPL* (Read-Evaluate-Print Loop, also etwa »Lesen-Auswerten-Ausgeben-Schleife«) genannt, einzugeben. Damit können Sie eine Python-Anweisung nach der anderen ausführen und die Ergebnisse unmittelbar einsehen. Die Verwendung dieser Shell eignet sich hervorragend, um zu lernen, was die grundlegenden Python-Anweisungen bewirken. Nutzen Sie sie daher, während Sie das Buch durcharbeiten. Auf diese Weise können Sie sich den Stoff besser merken, als wenn Sie ihn nur lesen würden.

Ausdrücke in die interaktive Shell eingeben

Um die interaktive Shell auszuführen, können Sie den Editor Mu starten, den Sie beim Durcharbeiten der Installationsanleitungen im Vorwort heruntergeladen haben. Auf Windows öffnen Sie dazu das Startmenü, geben **Mu** ein und starten die gleichnamige Anwendung. Auf macOS öffnen Sie den Ordner *Programme* und doppelklicken darin auf *Mu*. Klicken Sie auf die Schaltfläche *New* und speichern Sie die leere Datei als *blank.py*. Wenn Sie diese leere Datei ausführen, indem Sie auf *Run* klicken oder F5 drücken, wird die interaktive Shell als neuer Bereich am unteren Rand des Mu-Fensters geöffnet. Dort sehen Sie die Eingabeaufforderung `>>>` der Shell.

Geben Sie dort **2 + 2** ein, um Python eine einfache Berechnung ausführen zu lassen. Das Mu-Fenster zeigt jetzt Folgendes an:

```
>>> 2 + 2
4
>>>
```

In Python wird etwas wie `2 + 2` als *Ausdruck* bezeichnet. Dies ist die einfachste Form von Programmieranweisungen in dieser Sprache. Ausdrücke setzen sich aus *Werten* (wie `2`) und *Operatoren* (wie `+`) zusammen. Sie können stets *ausgewertet*, also auf einen einzigen Wert reduziert werden. Daher können Sie an allen Stellen im Python-Code, an denen ein Wert stehen soll, auch einen Ausdruck verwenden.

Im vorstehenden Beispiel wurde `2 + 2` zu dem Wert `4` ausgewertet. Ein einzelner Wert ohne Operatoren wird ebenfalls als Ausdruck angesehen, wird aber nur zu sich selbst ausgewertet:

```
>>> 2
2
```

Fehler sind kein Beinbruch

Wenn ein Programm Code enthält, den der Computer nicht versteht, stürzt es ab, woraufhin Python eine Fehlermeldung anzeigt. Ihren Computer können Sie dadurch jedoch nicht beschädigen. Daher brauchen Sie auch keine Angst vor Fehlern zu haben. Bei einem Absturz hält das Programm nur unerwartet an.

Wenn Sie mehr über eine bestimmte Fehlermeldung wissen wollen, können Sie online nach dem genauen Text suchen. Auf *www.dpunkt.de/python_automatisieren_2/* finden Sie außerdem eine Liste häufig auftretender Python-Fehlermeldungen und ihrer Bedeutungen.

Es gibt eine Menge verschiedener Operatoren, die Sie in Python-Ausdrücken verwenden können. Tabelle 1–1 führt die arithmetischen Operatoren auf.

Operator	Operation	Beispiel	Ergebnis
`**`	Exponent	`2 ** 3`	`8`
`%`	Modulo/Rest	`22 % 8`	`6`
`//`	Integerdivision/abgerundeter Quotient	`22 // 8`	`2`
`/`	Division	`22 / 8`	`2.75`
`*`	Multiplikation	`3 * 5`	`15`
`-`	Subtraktion	`5 - 2`	`3`
`+`	Addition	`2 + 2`	`4`

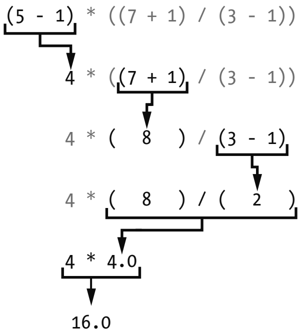
Tab. 1–1 *Arithmetische Operatoren, geordnet vom höchsten zum niedrigsten Rang*

Die Auswertungsreihenfolge oder *Rangfolge* der arithmetischen Operatoren in Python entspricht ihrer gewöhnlichen Rangfolge in der Mathematik: Als Erstes wird der Operator `**` ausgewertet, dann die Operatoren `*`, `/`, `//` und `%` von links nach rechts, und schließlich die Operatoren `+` und `-` (ebenfalls von links nach rechts). Um die Auswertungsreihenfolge zu ändern, können Sie bei Bedarf Klammern setzen. Der Weißraum zwischen den Operatoren und Werten spielt in Python keine Rolle (außer bei den Einrückungen am Zeilenanfang). Ein Abstand von einem Leerzeichen ist jedoch üblich. Zur Übung geben Sie die folgenden Ausdrücke in die interaktive Shell ein:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
```

```
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2 + 2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

Die Ausdrücke müssen Sie jeweils selbst eingeben, aber Python nimmt Ihnen die Arbeit ab, sie auf einen einzelnen Wert zu reduzieren. Wie die folgende Grafik zeigt, wertet es dabei die einzelnen Teile eines Ausdrucks nacheinander aus, bis ein einziger Wert übrig ist:



Die Regeln, nach denen Operatoren und Werte zu Ausdrücken zusammengestellt werden, bilden einen grundlegenden Bestandteil der Programmiersprache Python, vergleichbar mit den Grammatikregeln einer natürlichen Sprache. Betrachten Sie das folgende Beispiel:

Dies ist ein grammatikalisch korrekter deutscher Satz.

Dies grammatikalisch ist Satz kein deutscher korrekter.

Der zweite Satz lässt sich nur schwer verstehen (»parsen«, wie es bei einer Programmiersprache heißt), da er nicht den Regeln der deutschen Grammatik folgt. Genauso ist es, wenn Sie eine schlecht formulierte Python-Anweisung eingeben. Python versteht sie nicht und zeigt die Fehlermeldung `SyntaxError` an, wie die folgenden Beispiele zeigen:

```
>>> 5 +
  File "<stdin>", line 1
    5 +
      ^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2
  File "<stdin>", line 1
    42 + 5 + * 2
             ^
SyntaxError: invalid syntax
```

Um herauszufinden, ob eine Anweisung funktioniert oder nicht, können Sie sie einfach in die interaktive Shell eingeben. Keine Angst, dadurch können Sie nichts kaputt machen. Schlimmstenfalls zeigt Python eine Fehlermeldung an. Für professionelle Softwareentwickler gehören Fehlermeldungen zum Alltag.

Die Datentypen für ganze Zahlen, Fließkommazahlen und Strings

Ein *Datentyp* ist eine Kategorie für Werte, wobei jeder Wert zu genau einem Datentyp gehört. Die gebräuchlichsten Datentypen in Python finden Sie in Tabelle 1–2. Werte wie `-2` und `-30` sind beispielsweise *Integerwerte*. Dieser Datentyp (`int`) steht für ganze Zahlen. Zahlen mit Dezimalpunkt, z.B. `3.14`, sind dagegen *Fließkommazahlen* und weisen den Typ `float` auf. Beachten Sie, dass ein Wert wie `42` ein Integer ist, `42.0` dagegen eine Fließkommazahl.

Datentyp	Beispiele
Integer	`-2`, `-1`, `0`, `1`, `2`, `3`, `4`, `5`
Fließkommazahlen	`-1.25`, `-1.0`, `-0.5`, `0.0`, `0.5`, `1.0`, `1.25`
Strings	`'a'`, `'aa'`, `'aaa'`, `'Hello!'`, `'11 cats'`

Tab. 1–2 *Häufig verwendete Datentypen*

In Python-Programmen können auch Textwerte vorkommen, sogenannte *Strings* (`str`). Schließen Sie Strings immer in einfache Anführungszeichen ein (z.B. `'Hello'` oder `'Goodbye cruel world!'`), damit Python weiß, wo der String anfängt und wo er endet. Sie können sogar einen String erstellen, der gar keine Zeichen enthält, nämlich den *leeren String* `''`. In Kapitel 4 werden Strings ausführlicher behandelt.

Wenn Sie die Fehlermeldung `SyntaxError: EOL while scanning string literal` erhalten, haben Sie wahrscheinlich wie im folgenden Beispiel das schließende einfache Anführungszeichen am Ende eines Strings vergessen:

```
>>> 'Hello world!
SyntaxError: EOL while scanning string literal
```

Stringverkettung und -wiederholung

Die Bedeutung eines Operators kann sich in Abhängigkeit von den Datentypen der Werte ändern, die rechts und links von ihm stehen. Beispielsweise fungiert + zwischen zwei Integer- oder Fließkommawerten als Additionsoperator, zwischen zwei Strings aber als *Stringverkettungsoperator*. Probieren Sie Folgendes in der interaktiven Shell aus:

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

Dieser Ausdruck wird zu einem einzigen neuen String ausgewertet, der den Text der beiden Originalstrings enthält. Wenn Sie jedoch versuchen, den Operator + zwischen einem String und einem Integerwert einzusetzen, weiß Python nicht, wie es damit umgehen soll, und gibt eine Fehlermeldung aus:

```
>>> 'Alice' + 42
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    'Alice' + 42
TypeError: can only concatenate str (not "int") to str
```

Die Fehlermeldung `can only concatenate str (not "int") to str` bedeutet, dass Python glaubt, Sie wollten einen Integer mit dem String `'Alice'` verketten. Dazu aber müssten Sie den Integerwert ausdrücklich in einen String umwandeln, da Python dies nicht automatisch tun kann. (Die Umwandlung von Datentypen werden wir im Abschnitt »Analyse des Programms« weiter hinten in diesem Kapitel erklären und uns dabei mit den Funktionen `str()`, `int()` und `float()` beschäftigen.)

Zwischen zwei Integer- oder Fließkommawerten dient * als Multiplikationsoperator, doch zwischen einem String und einem Integerwert wird er zum *Stringwiederholungsoperator*. Um das auszuprobieren, geben Sie in die interaktive Shell Folgendes ein:

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

Der Ausdruck wird zu einem einzigen String ausgewertet, der den ursprünglichen String so oft enthält, wie der Integerwert angibt. Die Stringwiederholung ist zwar ein nützlicher Trick, wird aber längst nicht so häufig angewendet wie die Stringverkettung.

Den Operator * können Sie nur zwischen zwei numerischen Werten (zur Multiplikation) oder zwischen einem String- und einem Integerwert einsetzen (zur Stringwiederholung). In allen anderen Fällen zeigt Python eine Fehlermeldung an:

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'Alice' * 5.0
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

Es ist sinnvoll, dass Python solche Ausdrücke nicht auswertet. Schließlich ist es nicht möglich, zwei Wörter miteinander zu multiplizieren, und es dürfte auch ziemlich schwierig sein, einen willkürlichen String eine gebrochene Anzahl von Malen zu wiederholen.

Werte in Variablen speichern

Eine *Variable* können Sie sich wie eine Kiste im Arbeitsspeicher des Computers vorstellen, in der einzelne Werte abgelegt werden. Wenn Sie das Ergebnis eines ausgewerteten Ausdrucks an einer späteren Stelle in Ihrem Programm noch brauchen, können Sie es in einer Variablen festhalten.

Zuweisungsanweisungen

Um einen Wert in einer Variablen zu speichern, verwenden Sie eine *Zuweisungsanweisung*. Sie besteht aus einem Variablennamen, einem Gleichheitszeichen (das hier nicht als Gleichheitszeichen dient, sondern als *Zuweisungsoperator*) und dem zu speichernden Wert. Wenn Sie die Zuweisungsanweisung `spam = 42` eingeben, wird der Wert `42` in der Variablen `spam` gespeichert.

Sie können sich eine Variable als eine beschriftete Kiste vorstellen, in der der Wert abgelegt wird (siehe Abb. 1–1).

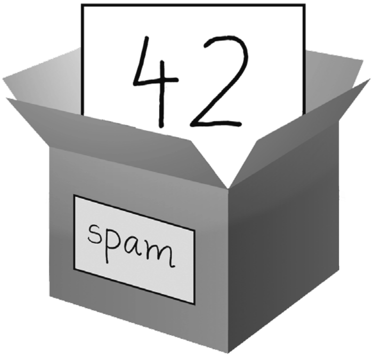


Abb. 1–1 *Die Anweisung* `spam = 42` *sagt dem Programm: »Die Variable* `spam` *enthält jetzt die Ganzzahl* 42.«

Geben Sie beispielsweise Folgendes in die interaktive Shell ein:

```
>>> spam = 40    ❶
>>> spam
40
>>> eggs = 2
>>> spam + eggs    ❷
42
>>> spam + eggs + spam
82
>>> spam = spam + 2    ❸
>>> spam
42
```

Eine Variable wird *initialisiert* (erstellt), wenn zum ersten Mal ein Wert in ihr gespeichert wird (❶). Danach können Sie sie zusammen mit anderen Variablen und Werten in Ausdrücken verwenden (❷). Wenn Sie der Variablen einen neuen Wert zuweisen (❸), geht der alte Wert verloren. Daher wird `spam` am Ende dieses Beispiels nicht mehr zu `40` ausgewertet, sondern zu `42`. Die Variable ist also *überschrieben* worden. Versuchen Sie in der interaktiven Shell wie folgt einen String zu überschreiben:

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

Wie die Kiste in Abb. 1–2 enthält die Variable `spam` in diesem Beispiel den Wert `Hello`, bis er durch `Goodbye` ersetzt wird.

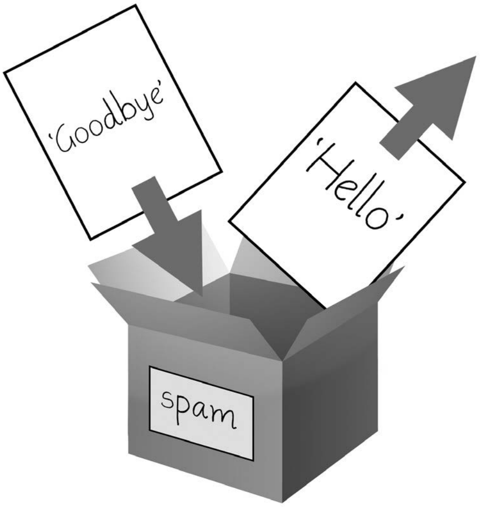


Abb. 1–2 *Wird einer Variablen ein neuer Wert zugewiesen, so wird der alte Wert vergessen.*

Variablennamen

Ein guter Variablenname beschreibt die enthaltenen Daten. Stellen Sie sich vor, Sie ziehen um und beschriften alle Kartons mit »Sachen«. Dann würden Sie Ihre Sachen nie wiederfinden! In diesem Buch und in einem Großteil der Python-Dokumentation werden allgemeine Variablennamen wie `spam`, `eggs` und `bacon` verwendet (in Anlehnung an den Spam-Sketch von Monty Python), aber in Ihren eigenen Programmen sollten Sie beschreibende Namen verwenden, um den Code leichter lesbar zu machen.

Sie können Variablen in Python fast beliebig benennen, allerdings gibt es einige Einschränkungen. Tabelle 1–3 führt Beispiele für gültige Variablennamen auf. Sie müssen die folgenden drei Regeln erfüllen:

- Der Name muss ein einzelnes Wort sein, darf also keine Leerzeichen enthalten.
- Der Name darf nur aus Buchstaben, Ziffern und dem Unterstrich bestehen.
- Der Name darf nicht mit einer Zahl beginnen.

Gültige Variablennamen	Ungültige Variablennamen
`current_balance`	`current-balance` (Bindestriche sind nicht zulässig)
`currentBalance`	`current balance` (Leerzeichen sind nicht zulässig)
`account4`	`4account` (der Name darf nicht mit einer Zahl beginnen)
`_42`	`42` (der Name darf nicht mit einer Zahl beginnen)
`TOTAL_SUM`	`TOTAL_$UM` (Sonderzeichen wie $ sind nicht zulässig)
`hello`	`'hello'` (Sonderzeichen wie ' sind nicht zulässig)

Tab. 1–3 *Gültige und ungültige Variablennamen*

Bei Variablennamen wird zwischen Groß- und Kleinschreibung unterschieden, sodass `spam`, `SPAM`, `Spam` und `sPaM` vier verschiedene Variablen bezeichnen. Zwar ist `Spam` ein zulässiger Variablenname, aber verabredungsgemäß sollten Variablennamen in Python mit einem Kleinbuchstaben beginnen.

In diesem Buch wird für Variablennamen die CamelCase-Schreibweise verwendet, also die Schreibung mit Binnenmajuskel statt mit einem Unterstrich. Variablennamen sehen also aus wie `lookLikeThis` und nicht wie `look_like_this`. Erfahrene Programmierer mögen einwenden, dass die offizielle Python-Stilrichtlinie PEP 8 Unterstriche verlangt. Ich bevorzuge allerdings die CamelCase-Schreibweise und möchte dazu auf den Abschnitt »Sinnlose Übereinstimmung ist die Plage kleiner Geister« aus PEP 8 verweisen:

> *»Übereinstimmung mit der Stilrichtlinie ist wichtig. Am wichtigsten ist es jedoch zu wissen, wann man diese Übereinstimmung aufgeben muss. Für manche Fälle ist die Stilrichtlinie einfach ungeeignet. Urteilen Sie dann selbst nach bestem Wissen und Gewissen.«*

Ihr erstes Programm

In der interaktiven Shell können Sie einzelne Python-Anweisungen nacheinander ausführen, aber um ein vollständiges Python-Programm zu schreiben, müssen Sie die Anweisungen in den *Dateieditor* eingeben. Er ähnelt Texteditoren wie dem Windows-Editor oder TextMate, verfügt aber zusätzlich über einige Sonderfunktionen für die Eingabe von Quellcode. Um in Mu eine neue Datei anzulegen, klicken Sie in der obersten Zeile auf *New*.

In dem Fenster, das jetzt erscheint, sehen Sie einen Cursor, der auf Ihre Eingaben wartet. Dieses Fenster unterscheidet sich jedoch von der interaktiven Shell, in der Python-Anweisungen ausgeführt werden, sobald Sie die Eingabetaste drücken. Im Dateieditor können Sie viele Anweisungen eingeben, die Datei speichern und dann das Programm ausführen. Anhand der folgenden Merkmale können Sie erkennen, in welchem der beiden Fenster Sie sich gerade befinden:

- Das Fenster der interaktiven Shell zeigt die Eingabeaufforderung >>> an.
- Im Dateieditorfenster gibt es die Eingabeaufforderung >>> nicht.

Nun ist es an der Zeit, Ihr erstes Programm zu schreiben! Geben Sie im Fenster des Dateieditors Folgendes ein:

```
# Dieses Programm sagt "Hallo" und fragt nach Ihrem Namen.   ❶

print('Hello world!')   ❷
print('What is your name?')    # Fragt nach dem Namen
myName = input()   ❸
print('It is good to meet you, ' + myName)   ❹
print('The length of your name is:')   ❺
print(len(myName))
print('What is your age?')     # Fragt nach dem Alter   ❻
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

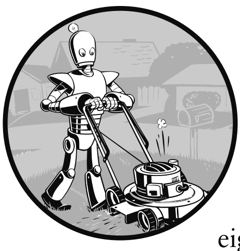
Nachdem Sie den Quellcode eingegeben haben, speichern Sie ihn, damit Sie ihn nicht jedes Mal neu eingeben müssen, wenn Sie Mu starten. Klicken Sie auf *Save*, geben Sie im Feld *File Name* den Namen **hello.py** ein und klicken Sie auf *Save*.

Während Sie ein Programm eingeben, sollten Sie es zwischendurch immer mal wieder speichern. Sollte Ihr Computer abstürzen oder sollten Sie versehentlich Mu beenden, verlieren Sie dann keinen Code. Als Tastaturkürzel zum Speichern einer Datei drücken Sie `Strg` + `S` auf Windows und Linux bzw. `Cmd` + `S` auf macOS.

Nachdem Sie das Programm gespeichert haben, führen Sie es aus. Drücken Sie dazu `F5`. Das Programm läuft jetzt im Fenster der interaktiven Shell. Beachten Sie aber, dass Sie `F5` im Editorfenster drücken müssen, nicht im Shell-Fenster. Geben Sie Ihren Namen ein, wenn das Programm Sie danach fragt. Die Programmausgabe im Fenster der interaktiven Shell sieht wie folgt aus:

```
Python 3.7.0b4 (v3.7.0b4:eb96c37699, May 2 2018, 19:02:22) [MSC v.1913 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Hello world!
What is your name?
Al
It is good to meet you, Al
The length of your name is:
2
```

3
Funktionen



In den vorhergehenden Kapiteln haben Sie bereits die Funktionen `print()`, `input()` und `len()` kennengelernt. Python bietet noch weitere integrierte Funktionen wie diese, aber Sie können auch Ihre eigenen schreiben. Eine *Funktion* ist ein Miniprogramm innerhalb eines Programms.

Um besser zu verstehen, wie Funktionen aufgebaut sind und was sie tun, wollen wir eine erstellen. Geben Sie das folgende Programm im Dateieditor ein und speichern Sie es als *helloFunc.py*:

```
def hello():   ❶
    print('Howdy!')   ❷
    print('Howdy!!!')
    print('Hello there.')

hello()   ❸
hello()
hello()
```

Die Ausführung dieses Programms können Sie sich auf *https://autbor.com/hellofunc/* ansehen. Die erste Zeile enthält die Anweisung `def` (❶), die eine Funktion namens `hello()` definiert. Der Code in dem darauf folgenden Block (❷) stellt den Rumpf der Funktion dar. Während der Funktionsdefinition wird er nicht ausgeführt, sondern erst, wenn die Funktion aufgerufen wird.

Die `hello()`-Zeilen im Anschluss an die Funktionsdefinition (❸) sind Funktionsaufrufe. Im Code wird ein solcher Aufruf in Form des Funktionsnamens gefolgt von den Klammern geschrieben, wobei in den Klammern Zahlen als Argumente stehen können. Wenn die Programmausführung diese Aufrufe erreicht, springt sie zur ersten Zeile in der Funktion und führt den dort vorhandenen Code aus. Am Ende der Funktion angelangt, kehrt die Ausführung wieder zu der Zeile zurück, in der die Funktion aufgerufen wurde. Der Code wird dann wie gehabt weiter abgearbeitet.

Da das Programm die Funktion `hello()` dreimal aufruft, wird der Code von `hello()` auch dreimal ausgeführt. Wenn Sie das Programm starten, sehen Sie folgende Ausgabe:

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

Ein wichtiger Zweck von Funktionen besteht darin, Code, der mehrfach ausgeführt wird, an einer Stelle zentral vorzuhalten. Ohne Funktionen müssten Sie den Code überall dort, wo er benötigt wird, komplett einfügen. Das Programm sähe dann wie folgt aus:

```
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
```

Eine solche Duplizierung von Code sollten Sie in jedem Fall vermeiden, denn wenn Sie Ihren Code irgendwann ändern – zum Beispiel, um einen Fehler zu korrigieren –, müssten Sie ihn sonst an jeder einzelnen Stelle anpassen, an der Sie ihn eingefügt haben.

Je mehr Programmiererfahrung Sie haben, umso häufiger werden Sie duplizierten oder kopierten Code entfernen können. Dadurch werden Ihre Programme kürzer, besser lesbar und leichter zu ändern.

Def-Anweisungen mit Parametern

Beim Aufruf von Funktionen wie `print()` und `len()` übergeben Sie Werte, sogenannte *Argumente*, indem Sie sie in die Klammern schreiben. Auch Ihre eigenen Funktionen können Sie so definieren, dass sie Argumente annehmen. Geben Sie das folgende Beispiel im Dateieditor ein und speichern Sie es als *helloFunc2.py*:

```
def hello(name):   ❶
    print('Hello ' + name)   ❷

hello('Alice')   ❸
hello('Bob')
```

Wenn Sie dieses Programm ausführen, erhalten Sie folgende Ausgabe:

```
Hello Alice
Hello Bob
```

Die Ausführung dieses Programms können Sie sich auf *https://autbor.com/hellofunc2/* ansehen. Die Definition der Funktion `hello()` in diesem Programm schließt den Parameter `name` ein (❶). Ein *Parameter* ist eine Variable, die Argumente enthält. Wird eine Funktion mit Argumenten aufgerufen, so werden diese Argumente in den Parametern gespeichert. Beim ersten Mal wird die Funktion `hello()` mit dem Argument `'Alice'` aufgerufen (❸). Die Programmausführung fährt mit der Funktion fort, wobei die Variable `name` automatisch auf `'Alice'` gesetzt wird. Dieser Text wird in die Ausgabe der Anweisung `print()` aufgenommen (❷).

Sobald die Funktion die Steuerung zurückgibt (wenn die Programmausführung also die Funktion verlässt und normal fortfährt), geht der in dem Parameter gespeicherte Wert jedoch verloren. Wenn Sie in dem vorstehenden Programm hinter `hello('Bob')` die Anweisung `print(name)` einfügen, erhalten Sie die Fehlermeldung `NameError`, da es zu diesem Zeitpunkt keine Variable namens `name` mehr gibt – sie ist nach dem Abschluss des Funktionsaufrufs `hello('Bob')` zerstört worden.

Das ähnelt dem Prinzip, dass auch die Variablen im Programm nach Beendigung des Programms vergessen werden. Weiter hinten in diesem Kapitel werde ich im Zusammenhang mit dem lokalen Gültigkeitsbereich von Funktionen noch ausführlicher darauf eingehen.

Terminologie

Die Begriffe *definieren*, *aufrufen*, *übergeben*, *Argument* und *Parameter* haben genau festgelegte Bedeutungen. Sehen wir uns das anhand eines Codebeispiels an:

```
def sayHello(name):   ❶
    print('Hello, ' + name)
sayHello('Al')   ❷
```

Eine Funktion zu *definieren* bedeutet, sie zu erstellen – auf ähnliche Weise, wie Sie mit einer Zuweisung wie `spam = 42` die Variable `spam` anlegen. Mit der `def`-Anweisung bei ❶ wird die Funktion `sayHello()` definiert. In der Zeile `sayHello('Al')` (❷) wird diese Funktion *aufgerufen*, wobei die Ausführung an den Anfang des Funktionscodes springt. Dieser Funktionsaufruf *übergibt* außerdem den Stringwert `'Al'` an die Funktion. Ein solcher in einem Funktionsaufruf übergebener Wert ist ein *Argument*. Hier wird das Argument `'Al'` der lokalen Variablen `name` zugewiesen. Variablen, denen Argumente zugewiesen werden, heißen *Parameter*.

Man kann diese Begriffe leicht verwechseln, es ist aber wichtig, dass Sie den Überblick behalten, denn dadurch ist sichergestellt, dass Sie die Bedeutung des Textes in diesem Kapitel verstehen.

Rückgabewerte und die Anweisung return

Wenn Sie die Funktion `len()` aufrufen und ihr ein Argument wie `'Hello'` übergeben, wird der Funktionsaufruf zur Länge des übergebenen Strings ausgewertet, hier also zu dem Integerwert `5`. Der Wert, zu dem ein Funktionsaufruf ausgewertet wird, ist der sogenannte *Rückgabewert* der Funktion.

Wenn Sie mit `def` eine eigene Funktion erstellen, können Sie mithilfe der Anweisung `return` festlegen, was der Rückgabewert sein soll. Eine `return`-Anweisung weist folgende Bestandteile auf:

- Das Schlüsselwort `return`
- Den Wert oder Ausdruck, den die Funktion zurückgeben soll

Wenn Sie in der `return`-Anweisung einen Ausdruck angeben, ist der Rückgabewert der Wert, zu dem dieser Ausdruck ausgewertet wird. Betrachten Sie als Beispiel das folgende Programm, das je nachdem, welche Zahl als Argument übergeben wird, einen anderen String zurückgibt. Geben Sie den folgenden Code in den Dateieditor ein und speichern Sie ihn als *magic8Ball.py*:

```
import random  ❶

def getAnswer(answerNumber):  ❷
    if answerNumber == 1:  ❸
        return 'It is certain'
    elif answerNumber == 2:
        return 'It is decidedly so'
    elif answerNumber == 3:
        return 'Yes'
    elif answerNumber == 4:
        return 'Reply hazy try again'
    elif answerNumber == 5:
        return 'Ask again later'
    elif answerNumber == 6:
        return 'Concentrate and ask again'
    elif answerNumber == 7:
        return 'My reply is no'
    elif answerNumber == 8:
        return 'Outlook not so good'
    elif answerNumber == 9:
        return 'Very doubtful'

r = random.randint(1, 9)  ❹
fortune = getAnswer(r)  ❺
print(fortune)  ❻
```

Die Ausführung dieses Programms können Sie sich auf *https://autbor.com/magic8ball.py/* ansehen. Zu Beginn importiert Python das Modul `random` (❶). Anschließend wird die Funktion `getAnswer()` definiert (❷). Da dies nur die Definition der Funktion ist, aber kein Aufruf, wird der darin enthaltene Code übersprungen. Die Ausführung fährt mit dem Aufruf der Funktion `random.randint()` mit den beiden Argumenten `1` und `9` fort (❹). Das Ergebnis ist ein Zufallsinteger zwischen 1 und 9 (einschließlich 1 und 9) und wird in der Variablen `r` gespeichert.

Als Nächstes wird die Funktion `getAnswer()` mit `r` als Argument aufgerufen (❺). Die Programmausführung springt zum Anfang dieser Funktion (❸), wo der Wert `r` im Parameter `answerNumber` gespeichert wird. Abhängig von dem Wert dieses Parameters gibt die Funktion nun einen der vielen möglichen Stringwerte zurück. Die Programmausführung kehrt anschließend zu der Zeile im Programm zurück, in der `getAnswer()` aufgerufen wurde (❺). Der zurückgegebene String wird der Variablen `fortune` zugewiesen, die an den Aufruf der Funktion `print()` übergeben (❻) und damit auf dem Bildschirm ausgegeben wird.

Da Sie Rückgabewerte als Argumente an andere Funktionsaufrufe übergeben können, lassen sich die drei folgenden Zeilen auch abkürzen:

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

Die folgende einzelne Zeile macht genau das Gleiche:

```
print(getAnswer(random.randint(1, 9)))
```

Wie Sie wissen, bestehen Ausdrücke aus Werten und Operatoren. Da ein Funktionsaufruf zu seinem Rückgabewert ausgewertet wird, kann er daher auch in einem Ausdruck verwendet werden.

Der Wert None

In Python gibt es den Wert `None`, der die Abwesenheit eines Wertes bedeutet. Dies ist der einzige Wert des Datentyps `NoneType`. (In anderen Programmiersprachen wird er auch als `null`, `nil` oder `undefined` bezeichnet.) Ebenso wie die booleschen Werte `True` und `False` muss auch `None` mit großem Anfangsbuchstaben geschrieben werden.

Dieser Nicht-Wert ist praktisch, wenn Sie etwas speichern müssen, das nicht mit einem echten Wert in einer Variablen verwechselt werden darf. Eine mögliche Anwendung für `None` ist der Rückgabewert von `print()`. Diese Funktion zeigt Text auf dem Bildschirm an, muss im Gegensatz zu `len()` oder `input()` aber eigentlich nichts zurückgeben. Da aber alle Funktionsaufrufe zu einem Rückgabewert ausgewertet werden, gibt `print()` pro forma `None` zurück. Um sich das anzusehen, geben Sie Folgendes in die interaktive Shell ein:

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

Wenn eine Funktionsdefinition keine `return`-Anweisung hat, hängt Python stillschweigend `return None` an, ebenso wie eine `while`- oder `for`-Schleife implizit mit einer `continue`-Anweisung endet. Wenn Sie eine `return`-Anweisung ohne Wert schreiben (also nur das Schlüsselwert `return` verwenden), wird ebenfalls `None` zurückgegeben.

Schlüsselwortargumente und print()

Die meisten Argumente werden anhand ihrer Position im Funktionsaufruf identifiziert. Beispielsweise ist `random.randint(1, 10)` etwas anderes als `random.randint(10, 1)`. Die erste Funktion gibt eine Zufallszahl zwischen 1 und 10 zurück, da das erste Argument die Untergrenze und das zweite die Obergrenze des Intervalls darstellt. Dagegen führt `random.randint(10, 1)` zu einem Fehler.

Schlüsselwortargumente werden dagegen durch das Schlüsselwort bezeichnet, das ihnen in dem Funktionsaufruf vorangestellt wird. Diese Art von Argumenten wird häufig für *optionale Parameter* verwendet. Beispielsweise können Sie bei der Funktion `print()` mit den optionalen Parametern `end` und `sep` angeben, was am Ende der Argumente ausgegeben werden soll und was dazwischen (als Trennzeichen).

Betrachten Sie das folgende Beispielprogramm:

```
print('Hello')
print('World')
```

Die Ausgabe sieht wie folgt aus:

```
Hello
World
```

Die beiden Strings werden auf getrennten Zeilen ausgegeben, da `print()` am Ende des übergebenen Strings automatisch einen Zeilenumbruch einfügt. Mit dem Schlüsselwortargument `end` können Sie dieses Verhalten jedoch ändern. Nehmen wir an, wir haben folgendes Programm:

```
print('Hello', end='')
print('World')
```

Hier sieht die Ausgabe wie folgt aus:

```
HelloWorld
```

Die Ausgabe erscheint auf einer einzigen Zeile, da hinter `'Hello'` kein Zeilenumbruch mehr ausgegeben wird, sondern ein leerer String. Das ist nützlich, wenn Sie den automatischen Zeilenumbruch hinter den Funktionsaufrufen von `print()` aufheben wollen.

Wenn Sie mehrere Stringwerte an `print()` übergeben, trennt die Funktion sie automatisch durch ein Leerzeichen. Geben Sie beispielsweise Folgendes in die interaktive Shell ein:

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

Das Standardtrennzeichen können Sie mit dem Schlüsselwortargument sep ändern. Probieren Sie in der interaktiven Shell Folgendes aus:

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

Auch zu Ihren selbst geschriebenen Funktionen können Sie Schlüsselwortargumente hinzufügen. Dazu müssen Sie sich jedoch mit den Datentypen für Listen und Wörterbücher (Dictionaries) auskennen, die wir in den nächsten beiden Kapiteln behandeln werden. Merken Sie sich zunächst nur, dass einige Funktionen auch über optionale Schlüsselwortargumente verfügen, die Sie beim Aufruf der Funktion angeben können.

Der Aufrufstack

Stellen Sie sich eines dieser Gespräche vor, in denen Sie vom Hundertsten ins Tausendste kommen. Beispielsweise wollen Sie eigentlich über Ihre Freundin Alice sprechen, was Sie an eine Geschichte über Ihren Kollegen Bob erinnert, die Sie aber nicht gleich erzählen können, weil Sie dazu erst einmal etwas über Ihre Cousine Carol berichten müssen. Wenn Sie mit den Hinweisen zu Carol fertig sind, kehren Sie zu der Geschichte über Bob zurück, und danach reden Sie wieder von Alice. Dabei aber werden Sie an Ihren Bruder David erinnert, weshalb Sie eine Bemerkung über ihn einschieben, bevor Sie endlich Ihre Geschichte über Alice abschließen. Ihre Äußerungen folgen dem Muster aus Abb. 3–1, in dem die Gesprächsthemen einen »Stapel« (*Stack*) bilden und das aktuelle Thema jeweils obenauf liegt.

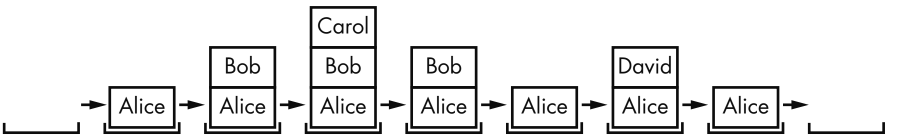


Abb. 3–1 *Der Stack einer Unterhaltung mit vielen Abschweifungen*

Ebenso führt auch der Aufruf einer Funktion nicht dazu, dass die Ausführung wie eine Reise ohne Wiederkehr zum Anfang der betreffenden Funktion umgeleitet wird. Python merkt sich, in welcher Zeile die Funktion aufgerufen wurde, sodass die Ausführung nach dem Auftreten einer return-Anweisung dort fortgesetzt werden kann. Ruft die ursprüngliche Funktion weitere Funktionen auf, kehrt die

Steuerung nach deren Verarbeitung zunächst zu dem Aufruf dieser verschachtelten Funktionen und erst dann zum Aufruf der ursprünglichen Funktion zurück.

Geben Sie im Dateieditor den folgenden Code ein und speichern Sie ihn als *abcdCallstack.py*:

```
def a():
    print('a() starts')
    b()   ❶
    d()   ❷
    print('a() returns')

def b():
    print('b() starts')
    c()   ❸
    print('b() returns')

def c():
    print('c() starts')   ❹
    print('c() returns')

def d():
    print('d() starts')
    print('d() returns')

a()   ❺
```

Wenn Sie dieses Programm ausführen, erhalten Sie die folgende Ausgabe:

```
a() starts
b() starts
c() starts
c() returns
b() returns
d() starts
d() returns
a() returns
```

Die Ausführung dieses Programms können Sie sich auf *https://autbor.com/abcdcallstack/* ansehen. Wird die Funktion `a()` aufgerufen (❺), so ruft sie ihrerseits `b()` auf (❶), die wiederum `c()` aufruft (❸). Die Funktion `c()` ruft nichts auf, sondern gibt lediglich die Zeilen `c() starts` (❹) und `c() returns` aus, bevor die Steuerung zu der Zeile in `b()` zurückspringt, in der `c()` aufgerufen wurde (❸). Dort angekommen, springt die Steuerung zu der Zeile in `a()` zurück, in der `b()` aufgerufen wurde (❶). Die Ausführung geht jetzt mit der nächsten Zeile in `a()` weiter, nämlich mit dem Aufruf von `d()` (❷). Ebenso wie `c()` ruft auch `d()` nichts auf, sondern gibt lediglich `d() starts` und `d() returns` aus, bevor sie die Steuerung an die Zeile in `a()` zurückgibt, in der sie aufgerufen wurde. Die letzte Zeile in `a()`gibt `a() returns` aus und springt dann zu dem ursprünglichen Aufruf von `a()` am Ende des Programms zurück (❺).

Mithilfe des *Aufrufstacks* merkt sich Python, wohin die Steuerung nach einem Funktionsaufruf jeweils zurückspringen muss. Dieser Stack wird jedoch nicht in einer Variablen gespeichert, sondern hinter den Kulissen von Python gehandhabt. Wenn das Programm eine Funktion aufruft, erstellt Python ein *Frameobjekt* an der Spitze des Aufrufstacks. In Frameobjekten ist die Zeilennummer des ursprünglichen Funktionsaufrufs gespeichert, sodass sich Python merken kann, wohin es zurückspringen muss. Erfolgt ein weiterer Funktionsaufruf, legt Python im Stack ein weiteres Frameobjekt auf das vorhergehende.

Beim Rücksprung aus einer aufgerufenen Funktion entfernt Python das zugehörige Frameobjekt vom Stapel und fährt mit der Ausführung in der Zeile mit der gespeicherten Nummer fort. Frameobjekte werden dabei immer oben auf den Stapel gelegt und auch dort wieder von ihm heruntergenommen, niemals an einer anderen Stelle. Abb. 3–2 zeigt die verschiedenen Zustände des Aufrufstacks von *abcdCallStack.py*, während die einzelnen Funktionen aufgerufen werden und zurückspringen.

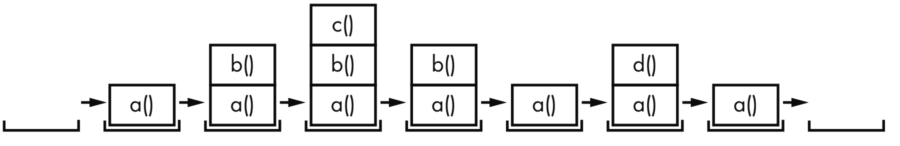


Abb. 3–2 *Frameobjekte im Aufrufstack im Verlauf der Funktionsaufrufe und Rücksprünge in abcdCallStack.py*

Das oberste Objekt auf dem Stack besagt, welche Funktion gerade ausgeführt wird. Wenn der Aufrufstack leer ist, erfolgt die Ausführung außerhalb von Funktionen.

Beim Aufrufstack handelt es sich um eine technische Hintergrundeinrichtung, die Sie nicht unbedingt kennen müssen, um Programme schreiben zu können. Es reicht zu wissen, dass Funktionsaufrufe zu der Zeile zurückkehren, von der aus sie aufgerufen wurden. Allerdings erleichtern Kenntnisse des Aufrufstacks das Verständnis der lokalen und globalen Gültigkeitsbereiche, die wir uns im folgenden Abschnitt ansehen.

Lokaler und globaler Gültigkeitsbereich

Parameter und Variablen, die innerhalb einer aufgerufenen Funktion zugewiesen werden, befinden sich im *lokalen Gültigkeitsbereich* der Funktion. Dagegen haben

Variablen, die außerhalb von Funktionen zugewiesen werden, einen *globalen Gültigkeitsbereich*. Variablen in einem lokalen Gültigkeitsbereich werden als *lokale Variablen* bezeichnet, Variablen im globalen Gültigkeitsbereich als *globale Variablen*. Eine Variable ist entweder lokal oder global, aber niemals beides.

Den Gültigkeitsbereich können Sie sich wie einen Behälter für Variablen vorstellen. Wird ein Gültigkeitsbereich zerstört, so gehen alle Werte der darin enthaltenen Variablen verloren. Es gibt nur einen globalen Gültigkeitsbereich. Er wird erstellt, wenn das Programm beginnt, und am Ende des Programms zerstört. Damit sind alle seine Variablen vergessen. Wäre das nicht der Fall, so würden beim nächsten Start des Programms alle Variablen immer noch die Werte aufweisen, die sie bei der letzten Ausführung hatten.

Ein lokaler Gültigkeitsbereich entsteht beim Aufruf einer Funktion. Jegliche Variablen, die in dieser Funktion zugewiesen werden, befinden sich in diesem lokalen Gültigkeitsbereich. Wenn die Funktion die Steuerung zurückgibt, wird der lokale Gültigkeitsbereich zerstört, sodass seine Variablen verloren gehen. Beim nächsten Aufruf der Funktion sind die Werte, die beim letzten Aufruf in den lokalen Variablen gespeichert waren, nicht mehr vorhanden. Die lokalen Variablen werden auch in Frameobjekten auf dem Aufrufstack abgelegt.

Das Prinzip der Gültigkeitsbereiche hat einige wichtige Auswirkungen:

- Code im globalen Gültigkeitsbereich, also außerhalb von Funktionen, kann keine lokalen Variablen nutzen.
- Code im lokalen Gültigkeitsbereich kann dagegen auf globale Variablen zugreifen.
- Code im lokalen Gültigkeitsbereich einer Funktion kann keine Variablen aus anderen lokalen Gültigkeitsbereichen nutzen.
- Sie können für zwei Variablen den gleichen Namen wählen, sofern sie sich in unterschiedlichen Gültigkeitsbereichen befinden. Es kann also beispielsweise sowohl eine lokale als auch eine globale Variable namens `spam` geben.

Warum gibt es in Python verschiedene Gültigkeitsbereiche, anstatt alle Variablen global zu machen? Wenn eine Variable durch den Code in einem bestimmten Funktionsaufruf bearbeitet wird, dann interagiert die Funktion mit dem Rest des Programms nur durch ihre Parameter und den Rückgabewert. Bei einem Fehler schränkt das die Menge der Codezeilen ein, die dafür verantwortlich sein können. Wenn in einem Programm, das ausschließlich globale Variablen enthält, ein Fehler dafür sorgt, dass einer Variablen ein falscher Wert zugewiesen wird, ist es ziemlich schwer, die entsprechende Stelle zu finden. Dieser Wert könnte überall in dem Programm zugewiesen worden sein – und das kann irgendwo in Hunderten oder gar Tausenden von Zeilen sein! Wenn aber eine lokale Variable einen falschen Wert hat, dann wissen Sie, dass er nur im Code der entsprechenden Funktion zugewiesen worden sein kann.

In kurzen Programmen ist die Verwendung globaler Variablen kein Problem, doch bei längeren Programmen sollten Sie sich lieber nicht darauf stützen.

Lokale Variablen können im globalen Gültigkeitsbereich nicht verwendet werden

Wenn Sie das folgende Programm ausführen, erhalten Sie eine Fehlermeldung:

```
def spam():
    eggs = 31337   ❶
spam()
print(eggs)
```

Die Ausgabe lautet wie folgt:

```
Traceback (most recent call last):
  File "C:/test1.py", line 4, in <module>
   print(eggs)
NameError: name 'eggs' is not defined
```

Das liegt daran, dass die Variable `eggs` nur in dem lokalen Gültigkeitsbereich existiert, der beim Aufruf von `spam()` erstellt wird (❶). Sobald die Programmausführung `spam` verlässt, wird dieser lokale Gültigkeitsbereich aber zerstört, weshalb es keine Variable namens `eggs` mehr gibt. Wenn das Programm versucht, `print(eggs)` auszuführen, meldet Python, dass `eggs` nicht definiert ist. Wenn Sie ein wenig darüber nachdenken, ist das auch tatsächlich sinnvoll. Während sich die Programmausführung im globalen Gültigkeitsbereich bewegt, gibt es keine lokalen Gültigkeitsbereiche und damit kann es auch keine lokalen Variablen geben. Aus diesem Grunde lassen sich im globalen Gültigkeitsbereich ausschließlich globale Variablen verwenden.

Lokale Gültigkeitsbereiche können keine Variablen aus anderen lokalen Gültigkeitsbereichen verwenden

Beim Aufruf einer Funktion wird ein neuer lokaler Gültigkeitsbereich erstellt. Das gilt auch dann, wenn die Funktion aus einer anderen Funktion heraus aufgerufen wird. Betrachten Sie dazu das folgende Programm:

```
def spam():
    eggs = 99   ❶
    bacon()   ❷
    print(eggs)   ❸
```

```
def bacon():
    ham = 101
    eggs = 0   ❹

spam()   ❺
```

Die Ausführung dieses Programms können Sie sich auf *https://autbor.com/otherlocalscopes/* ansehen. Zu Beginn wird hier die Funktion `spam()` aufgerufen (❺) und damit ein lokaler Gültigkeitsbereich erstellt. Die lokale Variable `eggs` (❶) wird auf `99` gesetzt. Daraufhin wird die Funktion `bacon()` aufgerufen (❷) und ein zweiter lokaler Gültigkeitsbereich erstellt. Mehrere lokale Gültigkeitsbereiche können zur selben Zeit existieren. In diesem neuen lokalen Gültigkeitsbereich wird die lokale Variable `ham` auf `101` gesetzt. Außerdem wird eine lokale Variable namens `eggs` erstellt und auf `0` gesetzt (❹). Allerdings ist dies eine andere Variable als die im lokalen Gültigkeitsbereich von `spam()`.

Wenn `bacon()` die Steuerung zurückgibt, wird der lokale Gültigkeitsbereich für diesen Aufruf und damit auch die Variable `eggs` zerstört. Die Programmausführung fährt mit der Funktion `spam()` fort, um den Wert von `eggs` auszugeben (❸). Der lokale Gültigkeitsbereich für den Aufruf von `spam()` existiert immer noch. Die einzige Variable namens `eggs`, die es jetzt noch gibt, ist diejenige der Funktion `spam()`, die auf `99` gesetzt wurde. Das ist der Wert, den das Programm ausgibt.

Was lernen wir daraus? Lokale Variablen in einer Funktion sind komplett von den lokalen Variablen in einer anderen Funktion getrennt.

Globale Variablen können von einem lokalen Gültigkeitsbereich aus gelesen werden

Betrachten Sie das folgende Programm:

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

Die Ausführung dieses Programms können Sie sich auf *https://autbor.com/readglobal/* ansehen. Da es in der Funktion `spam()` keinen Parameter namens `eggs` und auch keinen Code gibt, der `eggs` bei der Verwendung in `spam()` einen Wert zuweist, geht Python davon aus, dass hier auf die globale Variable `eggs` verwiesen wird. Daher gibt das vorstehende Programm den Wert `42` aus.

Lokale und globale Variablen mit demselben Namen

In Python ist es zwar technisch möglich, denselben Namen für eine globale und eine lokale Variable und für lokale Variablen in verschiedenen Gültigkeitsbereichen zu verwenden, doch um sich das Leben zu erleichtern, sollten Sie so etwas tunlichst vermeiden. Um zu sehen, was in einem solchen Fall geschieht, geben Sie folgenden Code in den Dateieditor ein und speichern ihn als *localGlobalSameName.py*:

```
def spam():
    eggs = 'spam local'   ❶
    print(eggs)   # Gibt 'spam local' aus

def bacon():
    eggs = 'bacon local'   ❷
    print(eggs)   # Gibt 'bacon local' aus
    spam()
    print(eggs)   # Gibt 'bacon local' aus

eggs = 'global'   ❸
bacon()
print(eggs)      # Gibt 'global' aus
```

Wenn Sie dieses Programm ausführen, erhalten Sie folgende Ausgabe:

```
bacon local
spam local
bacon local
global
```

Die Ausführung dieses Programms können Sie sich auf *https://autbor.com/localglobalsamename/* ansehen. Tatsächlich enthält dieses Programm drei verschiedene Variablen, die alle den Namen `eggs` tragen, was nicht gerade übersichtlich ist. Es handelt sich dabei um folgende Variablen:

1. Die Variable `eggs` im lokalen Gültigkeitsbereich des Aufrufs von `spam()` (❶)
2. Die Variable `eggs` im lokalen Gültigkeitsbereich des Aufrufs von `bacon()` (❷)
3. Die Variable `eggs` im globalen Gültigkeitsbereich (❸)

Da diese drei Variablen alle denselben Namen haben, kann es ziemlich kompliziert werden, nachzuvollziehen, welche zu einem bestimmten Zeitpunkt gerade verwendet wird. Daher sollten Sie es vermeiden, Variablen in unterschiedlichen Gültigkeitsbereichen denselben Namen zu geben.

Die Anweisung global

Wenn Sie innerhalb einer Funktion eine globale Variable bearbeiten wollen, müssen Sie die Anweisung `global` verwenden. Eine Zeile wie `global eggs` am Anfang einer Funktion weist Python an: »In dieser Funktion bezieht sich `eggs` auf die globale Variable, also erstelle keine lokale Variable mit diesem Namen!« Geben Sie als Beispiel den folgenden Code in den Dateieditor ein und speichern Sie ihn als *globalStatement.py*:

```
def spam():
    global eggs   ❶
    eggs = 'spam'   ❷

eggs = 'global'
spam()
print(eggs)
```

Wenn Sie dieses Programm ausführen, gibt der letzte Aufruf von `print()` Folgendes aus:

```
spam
```

Die Ausführung dieses Programms können Sie sich auf *https://autbor.com/globalstatement/* ansehen. Da `eggs` am Anfang von `spam()` als `global` deklariert ist (❶), erfolgt die Zuweisung von `'spam'` zur globalen Variablen `eggs` (❷). Es wird keine lokale Variable namens `eggs` erstellt.

Es gibt vier Regeln, um lokale und globale Variablen zu unterscheiden:

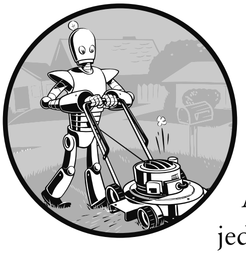
- Wenn eine Variable in einem globalen Gültigkeitsbereich verwendet wird (also außerhalb irgendeiner Funktion), dann handelt es sich um eine globale Variable.
- Wird in einer Funktion die Anweisung `global` für die Variable verwendet, handelt es sich um eine globale Variable.
- Wird die Variable in einer Zuweisungsanweisung innerhalb der Funktion verwendet, handelt es sich um eine lokale Variable.
- Wird die Variable dagegen nicht in einer Zuweisungsanweisung verwendet, handelt es sich um eine globale Variable.

Um ein besseres Gefühl für diese Regeln zu bekommen, schauen Sie sich das folgende Beispielprogramm an. Geben Sie den folgenden Code in den Dateieditor ein und speichern Sie ihn als *sameNameLocalGlobal.py*:

```
def spam():
    global eggs   ❶
    eggs = 'spam' # Globale Variable
```

7

Mustervergleich mit regulären Ausdrücken



Sicherlich sind Sie damit vertraut, zur Suche nach Text `Strg` + `F` zu drücken und dann die Wörter einzugeben, nach denen Sie Ausschau halten wollen. Mit *regulären Ausdrücken* können Sie jedoch noch einen Schritt weitergehen und ein *Textmuster* eingeben, nach dem gesucht werden sollen. Beispielsweise werden Telefonnummern in den USA und Kanada in Form von drei Ziffern, einem Bindestrich und vier weiteren Ziffern angegeben (mit einer optionalen dreistelligen Bereichsvorwahl ganz am Anfang). Wenn Sie also so etwas wie 415–555-1234 sehen, können Sie zumindest erkennen, dass es sich um eine Telefonnummer handelt (auch wenn Sie nicht sagen können, dass es die richtige Nummer für den gewünschten Anschluss ist), während Ihnen klar ist, dass 4.155.551,234 keine Telefonnummer ist.

Im täglichen Leben sind wir es auch gewohnt, andere Arten von Textmustern zu erkennen: E-Mail-Adressen haben ein `@`-Symbol in der Mitte, Kfz-Kennzeichen bestehen aus zwei bis drei Buchstaben für den Kreis gefolgt von ein bis zwei weiteren Buchstaben und ein bis drei Ziffern, die URLs von Websites weisen oft

Punkte und Schrägstriche auf, Social-Media-Hashtags beginnen mit # und enthalten keine Leerzeichen usw.

Reguläre Ausdrücke sind äußerst nützlich, aber viele Nicht-Programmierer wissen nicht viel darüber, obwohl moderne Textverarbeitungssysteme wie Microsoft Word und OpenOffice auch Funktionen zum Suchen und Ersetzen auf der Grundlage von regulären Ausdrücken anbieten. Dabei können reguläre Ausdrücke enorm viel Zeit ersparen, nicht nur bei der Anwendung von Software, sondern auch bei der Programmierung. Der Fachbuchautor Cory Doctorow hat sogar dafür plädiert, Anfängern zunächst reguläre Ausdrücke beizubringen und erst danach das eigentliche Programmieren:

> *»Sich mit [regulären Ausdrücken] auszukennen, kann darüber entscheiden, ob man zur Lösung eines Problems drei Schritte benötigt oder 3000. Als Computerfreak ist Ihnen oft nicht mehr bewusst, dass die Probleme, die Sie mit wenigen Tastendrücken lösen, für andere Leute eine mühsame, fehleranfällige Quälerei darstellen.«*[1]

In diesem Kapitel schreiben Sie als Erstes ein Programm, das Textmuster *ohne* Zuhilfenahme regulärer Ausdrücke findet, um dann zu sehen, wie Sie den Code für die gleiche Aufgabe mithilfe regulärer Ausdrücke viel knapper formulieren können. Ich zeige Ihnen die Grundlagen des Mustervergleichs mit regulären Ausdrücken und gehe dann zu einigen äußerst praktischen Vorgehensweisen wie Stringersetzung und der Erstellung von eigenen Zeichenklassen über. Am Ende des Kapitels schreiben Sie ein Programm, das automatisch Telefonnummern und E-Mail-Adressen aus einem Textblock extrahieren kann.

Textmuster ohne reguläre Ausdrücke finden

Nehmen wir an, Sie möchten eine (amerikanische) Telefonnummer in einem String finden. Das Muster haben Sie weiter vorn schon kennengelernt: drei Ziffern, ein Bindestrich, drei Ziffern, ein weiterer Bindestrich und schließlich vier Ziffern, beispielsweise 415–555-4242.

Schreiben wir nun die Funktion `isPhoneNumber()`, die prüft, ob ein String diesem Muster genügt, und dementsprechend entweder `True` oder `False` zurückgibt. Geben Sie dazu folgenden Code im Dateieditor ein und speichern Sie ihn als *isPhoneNumber.py*:

1 Cory Doctorow, »Here's what ICT should really teach kids: how to do regular expressions«, Guardian, 4. Dezember 2012, *http://www.theguardian.com/technology/2012/dec/04/ict-teach-kids-regular-expressions/*.

```
def isPhoneNumber(text):
    if len(text) != 12:   ❶
        return False
    for i in range(0, 3):
        if not text[i].isdecimal():   ❷
            return False
    if text[3] != '-':   ❸
        return False
    for i in range(4, 7):
        if not text[i].isdecimal():   ❹
            return False
    if text[7] != '-':   ❺
        return False
    for i in range(8, 12):
        if not text[i].isdecimal():   ❻
            return False
    return True   ❼

print('415-555-4242 is a phone number:')
print(isPhoneNumber('415-555-4242'))
print('Moshi moshi is a phone number:')
print(isPhoneNumber('Moshi moshi'))
```

Die Ausgabe sieht wie folgt aus:

```
415-555-4242 is a phone number:
True
Moshi moshi is a phone number:
False
```

Der Code der Funktion `isPhoneNumber()` führt mehrere Prüfungen durch, um herauszufinden, ob der String in `text` eine gültige Telefonnummer ist. Wird nur einer dieser Tests nicht bestanden, gibt die Funktion `False` zurück. Als Erstes prüft der Code, ob der String genau zwölf Zeichen lang ist (❶). Danach prüft er, ob die Bereichsvorwahl (die ersten drei Zeichen des Textes) ausschließlich aus numerischen Zeichen besteht (❷). Der Rest der Funktion untersucht nacheinander die einzelnen Bestandteile des Musters für amerikanische Telefonnummern: Hinter der Bereichsvorwahl muss ein Bindestrich stehen (❸) und darauf müssen drei weitere numerische Zeichen (❹), ein weiterer Bindestrich (❺) und schließlich noch einmal vier Ziffern folgen (❻). Wurden alle Tests bestanden, gibt das Programm `True` zurück (❼).

Der Aufruf von `isPhoneNumber()` mit dem Argument `'415-555-4242'` gibt daher `True` zurück, der Aufruf mit `'Moshi moshi'` dagegen `False`. Hier wird schon der erste Test nicht bestanden, da `'Moshi moshi'` nicht zwölf Zeichen lang ist.

Um dieses Textmuster in einem längeren String zu finden, müssen Sie noch mehr Code hinzufügen. Ersetzen Sie die letzten vier Aufrufe der Funktion `print()` in *isPhoneNumber.py* durch folgenden Code:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'
for i in range(len(message)):
    chunk = message[i:i+12]   ❶
    if isPhoneNumber(chunk):   ❷
        print('Phone number found: ' + chunk)
print('Done')
```

Die Ausgabe sieht wie folgt aus:

```
Phone number found: 415-555-1011
Phone number found: 415-555-9999
Done
```

Bei jedem Durchlauf durch die `for`-Schleife wird der Variablen `chunk` ein neuer Abschnitt von zwölf Zeichen aus `message` zugewiesen (❶). Beim ersten Durchlauf ist `i` gleich `0`, weshalb `message[0:12]` zu `chunk` zugewiesen wird (also der String `'Call me at 4'`). Bei der nächsten Iteration haben wir `i` gleich `1` und damit `message[1:13]` (der String `'all be at 41'`). Im Verlauf der einzelnen Iterationen der `for`-Schleife nimmt `chunk` also nacheinander die folgenden Werte an:

- `'Call me at 4'`
- `'all me at 41'`
- `'ll me at 415'`
- `'l me at 415-'`
- usw.

Dabei wird `chunk` jeweils der Funktion `isPhoneNumber()` übergeben, um herauszufinden, ob der Textabschnitt dem Muster einer Telefonnummer entspricht (❷). Wenn ja, wird der Abschnitt ausgegeben.

Wenn Sie die Schleife den gesamten Text von `message` durchlaufen lassen, stehen am Ende tatsächlich die zwölf Zeichen einer Telefonnummer in `chunk`. Die Schleife geht den gesamten String durch und prüft jeden einzelnen Abschnitt von zwölf Zeichen. Wenn sie dabei einen Abschnitt findet, der die Bedingungen von `isPhoneNumber()` erfüllt, gibt sie diesen aus. Nachdem `message` komplett abgearbeitet ist, geben wir `Done` aus.

In diesem Beispiel ist der String in `message` sehr kurz, er könnte aber auch Millionen von Zeichen lang sein. Das Programm würde auch in diesem Fall weniger als eine Sekunde benötigen, um ihn zu untersuchen. Ein vergleichbares Programm, das Telefonnummern mithilfe regulärer Ausdrücke findet, hat eine ähnliche Ausführungszeit, lässt sich aber viel schneller schreiben.

Textmuster mithilfe regulärer Ausdrücke finden

Das Telefonnummernsuchprogramm aus dem vorherigen Abschnitt funktioniert zwar, braucht aber eine ganze Menge Code, um eine sehr beschränkte Aufgabe auszuführen: Die Funktion `isPhoneNumber()` umfasst 17 Zeilen und kann nur ein einziges Muster von Telefonnummern finden. Was ist aber, wenn jemand eine Telefonnummer in einem Format wie 415.555.4242 oder (415) 555–4242 schreibt? Oder wenn die Telefonnummer über eine Nebenstellenerweiterung verfügt, z.B. 415–555-4242 x99? Bei der Überprüfung solcher Werte würde `isPhoneNumber()` versagen. Natürlich könnten Sie noch mehr Code hinzufügen, um auch diese Muster abzudecken, doch es gibt eine einfachere Möglichkeit.

Reguläre Ausdrücke (englisch »regular expressions«, was oft zu »regex« abgekürzt wird) beschreiben Textmuster. Beispielsweise steht `\d` in einem regulären Ausdruck für eine Ziffer, also einen einzelnen numerischen Wert von 0 bis 9. Mit dem regulären Ausdruck `\d\d\d-\d\d\d-\d\d\d\d` können Sie in Python Text finden, der genauso aufgebaut ist wie derjenige, den Sie auch mit der Funktion `isPhoneNumber()` ermittelt haben, also Text aus einem String aus drei Ziffern, einem Bindestrich, drei weiteren Ziffern, einem zweiten Bindestrich und schließlich vier Ziffern. Kein anderer String stimmt mit dem regulären Ausdruck `\d\d\d-\d\d\d-\d\d\d\d` überein.

Reguläre Ausdrücke bieten aber noch mehr und weitreichendere Möglichkeiten. Beispielsweise können Sie eine Zahl in geschweifte Klammern schreiben (z.B. `{3}`), um anzugeben, dass das vorstehende Muster dreimal vorhanden sein muss. Das amerikanische Telefonnummernformat können Sie daher auch mithilfe des kürzeren regulären Ausdrucks `\d{3}-\d{3}-\d{4}` ausdrücken.

Regex-Objekte erstellen

Alle Regex-Funktionen von Python befinden sich im Modul `re`. Um es zu importieren, geben Sie Folgendes in die interaktive Shell ein:

```
>>> import re
```

Hinweis

Da Sie für die meisten Beispiele in diesem Kapitel das Modul `re` benötigen, müssen Sie es zu Anfang jedes Skripts bzw. bei jedem Neustart von Mu importieren. Anderenfalls erhalten Sie die Fehlermeldung `NameError: name 're' is not defined`.

Wenn Sie re.compile() einen Stringwert mit einem regulären Ausdruck übergeben, gibt die Funktion ein Regex-Musterobjekt zurück (oder kurz gesagt, ein Regex-Objekt).

Um ein Regex-Objekt für das Telefonnummernmuster zu erstellen, geben Sie in der interaktiven Shell Folgendes ein:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

Jetzt enthält die Variable phoneNumRegex ein Regex-Objekt.

Vergleiche mit einem Regex-Objekt

Die Methode search() eines Regex-Objekts durchsucht den ihr übergebenen String nach Übereinstimmungen mit dem regulären Ausdruck. Sie gibt None zurück, wenn das Muster des Ausdrucks in dem String nicht zu finden ist, und anderenfalls ein Match-Objekt. Solche Objekte wiederum verfügen über die Methode group(), die den übereinstimmenden Text aus dem durchsuchten String zurückgibt. (Was es mit den namensgebenden Gruppen auf sich hat, erkläre ich in Kürze.) Geben Sie zum Ausprobieren Folgendes in die interaktive Shell ein:

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

Die Bezeichnung mo ist lediglich ein generischer Variablenname für Match-Objekte. Dieses Beispiel mag zunächst ziemlich kompliziert aussehen, allerdings können Sie schon auf einen Blick erkennen, dass es viel kürzer ist als *iPhoneNumber.py*, obwohl es genau die gleiche Aufgabe erledigt.

Hier übergeben wir das gesuchte Muster an re.compile() und speichern das resultierende Regex-Objekt in phoneNumRegex. Anschließend rufen wir die Methode search() für phoneNumRegex auf und übergeben ihr den String, den wir nach Übereinstimmungen durchsuchen wollen. Das Ergebnis der Suche wird in der Variablen mo gespeichert. In diesem Beispiel wissen wir bereits, dass das Muster in dem String enthalten ist und dass daher ein Match-Objekt zurückgeben wird. Da mo also mit Sicherheit ein Match-Objekt enthält und nicht den Wert None, können wir group() für mo aufrufen, um den übereinstimmenden String zurückzugeben. Da wir mo.group() in einer print-Anweisung verwenden, wird der übereinstimmende Text (415-555-4242) ausgegeben.

Zusammenfassung: Mustervergleich mit regulären Ausdrücken

Die Verwendung regulärer Ausdrücke in Python umfasst mehrere Schritte, die aber alle ziemlich einfach sind:

1. Importieren Sie das Regex-Modul mit `import re`.
2. Erstellen Sie mithilfe der Funktion `re.compile()` ein `Regex`-Objekt. (Verwenden Sie dazu am besten einen Rohstring.)
3. Übergeben Sie den String, den Sie durchsuchen wollen, an die Methode `search()` des `Regex`-Objekts. Sie gibt ein `Match`-Objekt zurück.
4. Rufen Sie die Methode `group()` des `Match`-Objekts auf, um einen String mit dem übereinstimmenden Text zurückzugeben.

Hinweis

Ich ermutige Sie zwar immer dazu, den Beispielcode in der interaktiven Shell auszuprobieren, aber Sie sollten sich auch einige der Testdienste für reguläre Ausdrücke im Internet ansehen, die Ihnen genau zeigen, ob ein regulärer Ausdruck mit einem von Ihnen eingegebenen Text übereinstimmt. Dazu empfehle ich Ihnen den Testdienst auf *https://pythex.org/*.

Weitere Möglichkeiten für den Mustervergleich mithilfe regulärer Ausdrücke

Nachdem Sie nun die grundlegenden Schritte kennen, um reguläre Ausdrücke in Python zu erstellen und für den Mustervergleich zu verwenden, wollen wir uns einige Besonderheiten ansehen, die die Möglichkeiten noch erweitern.

Gruppierung durch Klammern

Nehmen wir an, Sie wollen die Bereichsvorwahl vom Rest der Telefonnummer trennen. Mithilfe von Klammern können Sie dazu innerhalb des regulären Ausdrucks *Gruppen* anlegen: `(\d\d\d)-(\d\d\d-\d\d\d\d)`. Anschließend können Sie mit der Methode `group()` des `Match`-Objekts gezielt den übereinstimmenden Text für eine einzelne Gruppe abrufen.

Der erste Satz von Klammern in einem Regex-String ist Gruppe 1, der zweite ist Gruppe 2. Durch die Übergabe des Integers `1` oder `2` an `group()` rufen Sie unterschiedliche Teile des übereinstimmenden Textes ab. Falls Sie `0` oder gar nichts übergeben, bekommen Sie den gesamten übereinstimmenden Text. Das können Sie wie folgt in der interaktiven Shell ausprobieren:

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
```

9

Dateien lesen und schreiben

Variablen sind eine praktische Einrichtung, um Daten festzuhalten, während das Programm ausgeführt wird, aber wenn die Daten erhalten bleiben sollen, nachdem das Programm beendet ist, müssen Sie sie in einer Datei speichern. Sie können sich den Inhalt einer Datei als einen einzigen String vorstellen, dessen Größe sich durchaus in Gigabyte misst. In diesem Kapitel erfahren Sie, wie Sie in Python Dateien auf der Festplatte erstellen, lesen und speichern.

Dateien und Dateipfade

Die beiden wichtigsten Merkmale einer Datei sind der *Dateiname* (der gewöhnlich als einzelnes Wort geschrieben wird) und der *Pfad*, der den Speicherort der Datei auf dem Computer angibt. Auf meinem Windows-Laptop habe ich beispielsweise eine Datei mit dem Namen *projects.docx* im Pfad *C:\Users\Al\Documents*. Der Teil des Dateinamens hinter dem Punkt wird als *Dateinamenerweiterung* oder *Endung* bezeichnet und gibt den Typ der Datei an. Bei *project.docx* handelt es sich um ein Word-Dokument, und *Users*, *Al* und *Documents* sind *Ordner* (oder *Ver-*

zeichnisse). Ordner können Dateien sowie andere Ordner enthalten. Beispielsweise befindet sich *projects.docx* im Ordner *Documents*, der wiederum im Ordner *Al* innerhalb des Ordners *Users* steckt. Diese Ordnerstruktur sehen Sie in Abb. 9–1.

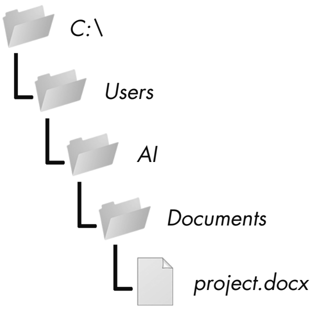


Abb. 9–1 *Eine Datei in einer Ordnerhierarchie*

Die Angabe *C:* im Pfad ist der *Wurzelordner* oder (unter Windows) *Stammordner*, der alle anderen Ordner enthält. Unter Windows heißt dieser Stammordner *C:* und wird auch als »Laufwerk *C:*« bezeichnet. Der Wurzelordner unter macOS und Linux ist /. In diesem Buch verwende ich den Windows-Stammordner *C:*. Wenn Sie die Beispiele in einer interaktiven Shell unter macOS oder Linux eingeben, müssen Sie diese Angabe durch / ersetzen.

Zusätzliche Datenträger oder *Volumes*, etwa ein DVD-Laufwerk oder ein USB-Stick, werden in den verschiedenen Betriebssystemen unterschiedlich dargestellt. Unter Windows erscheinen sie als weitere, mit Buchstaben gekennzeichnete Stammlaufwerke wie *D:* oder *E:*, unter macOS als neue Ordner im Ordner */Volumes* und unter Linux als neue Ordner unter */mnt* (»mount«, was das Bereitstellen oder Einhängen eines solchen Laufwerks bezeichnet). Beachten Sie auch, dass bei Datei- und Ordnernamen unter Linux zwischen Groß- und Kleinschreibung unterschieden wird, unter Windows und macOS jedoch nicht.

Hinweis

Auf Ihrem System befinden sich natürlich andere Dateien und Ordner als auf meinem. Die Ergebnisse der Beispiele in diesem Kapitel werden daher nicht exakt gleich aussehen. Versuchen Sie aber, sie mithilfe Ihrer Dateien und Ordner nachzuvollziehen.

Backslash unter Windows und Schrägstrich unter macOS und Linux

Unter Windows werden Pfade mit Backslashes, also umgekehrten Schrägstrichen (\), als Trennzeichen zwischen den Ordnernamen geschrieben, unter macOS und Linux jedoch mit einem normalen Schrägstrich (/). Wenn Ihre Programme auf allen Betriebssystemen laufen sollen, müssen Sie Ihre Python-Skripte für beide Fälle auslegen.

Zum Glück lässt sich das mit der Funktion `Path()` aus dem Modul `pathlib` leicht erledigen. Wenn Sie ihr die Strings der einzelnen Datei- und Ordnernamen eines Pfades übergeben, gibt sie den Dateipfad als String mit den richtigen Trennzeichen für das vorliegende Betriebssystem zurück. Probieren Sie Folgendes in der interaktiven Shell aus:

```
>>> from pathlib import Path
>>> Path('spam', 'bacon', 'eggs')

WindowsPath('spam/bacon/eggs')

>>> str(Path('spam', 'bacon', 'eggs'))
'spam\\bacon\\eggs'
```

Die übliche Vorgehensweise für den Import von `pathlib` besteht darin, `from pathlib import Path` anzugeben, da Sie sonst überall, wo Sie `Path` in Ihrem Code brauchen, `pathlib.Path` schreiben müssten, was ziemlich viel Tipparbeit wäre.

In der interaktiven Shell auf Windows gibt `Path('spam', 'bacon', 'eggs')` ein `WindowsPath`-Objekt für den zusammengefügten Pfad zurück. Dargestellt wird dieses Objekt als `WindowsPath('spam/bacon/eggs')`. Obwohl Windows-Dateipfade Backslashes enthalten, werden bei der Anzeige in der interaktiven Shell normale Schrägstriche verwendet, da Open-Source-Softwareentwickler nun mal Linux bevorzugen.

Wenn Sie aus diesem Pfad einen einfachen Textstring gewinnen wollen, können Sie ihn an die Funktion `str()` übergeben. In unserem Fall führt das zu dem Rückgabewert `'spam\\bacon\\eggs'` (mit doppelten Backslashes, da jeder Backslash mit einem zweiten maskiert werden muss). Auf Linux dagegen gibt `Path()` ein `PosixPath`-Objekt zurück, dessen Übergabe an `str()` den Rückgabewert `'spam/bacon/eggs'` ergibt. (Bei *POSIX* handelt es sich um einen Satz von Standards für Unix-artige Betriebssysteme wie Linux.)

Diese `Path`-Objekte (je nach Betriebssystem also `WindowsPath`- oder `PosixPath`-Objekte) können Sie an verschiedene der Funktionen zur Dateibearbeitung übergeben, die Sie in diesem Kapitel noch kennenlernen werden. Beispielsweise hängt der folgende Code Dateinamen aus einer Liste hinten an den Ordnernamen an:

```
>>> from pathlib import Path
>>> myFiles = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in myFiles:
        print(Path(r'C:\Users\Al', filename))
C:\Users\Al\accounts.txt
```

```
C:\Users\Al\details.csv
C:\Users\Al\invite.docx
```

Auf Windows trennt der Backslash Verzeichnisse voneinander, weshalb Sie ihn nicht in Dateinamen verwenden können. Dagegen sind Backslashes in Dateinamen auf macOS und Linux zulässig. Während `Path(r'spam\eggs')` auf Windows zwei verschiedene Ordner bezeichnet (oder die Datei *eggs* im Ordner *spam*), steht derselbe Befehl auf macOS und Linux für einen einzelnen Ordner (oder eine Datei) namens *spam\eggs*. Aus diesem Grund ist es sinnvoll, in Python-Code nur normale Schrägstriche zu verwenden (was ich auch im Rest dieses Kapitels so handhaben werde). Das Modul `pathlib` stellt sicher, dass der Code auf allen Betriebssystemen funktioniert.

Das Modul `pathlib` wurde in Python 3.4 als Ersatz für die älteren `os.path`-Funktionen eingeführt. Seit Python 3.6 wird es von den Modulen der Python-Standardbibliothek unterstützt, aber wenn Sie noch mit Python 2 arbeiten, sollten Sie `pathlib2` verwenden. Damit steht Ihnen die Funktionalität von `pathlib` auch auf Python 2.7 zur Verfügung. In Anhang A finden Sie eine Anleitung, um `pathlib2` mithilfe von `pip` zu installieren. Immer wenn ich eine ältere `os.path`-Funktion durch `pathlib` ersetzt habe, mache ich mir eine Notiz dazu. Mehr über die älteren Funktionen erfahren Sie auf *https://docs.python.org/3/library/os.path.html*.

Pfade mit dem Operator / zusammenfügen

Den Operator `+` können wir verwenden, um zwei Integer- oder Fließkommazahlen wie in dem Ausdruck `2 + 2` zu addieren, der zu `4` ausgewertet wird, aber auch, um zwei Strings zu verketten, etwa in `'Hello' + 'World'`, was `'HelloWorld'` ergibt. Ebenso ist es auch möglich, den Operator `/`, der gewöhnlich zur Division dient, zur Verbindung von `Path`-Objekten und Strings zu nutzen. Das ist praktisch, um ein von der Funktion `Path()` erstelltes `Path`-Objekt zu bearbeiten.

Probieren Sie das wie folgt in der interaktiven Shell aus:

```
>>> from pathlib import Path
>>> Path('spam') / 'bacon' / 'eggs'
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon/eggs')
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon', 'eggs')
WindowsPath('spam/bacon/eggs')
```

Die Anwendung des Operators / auf `Path`-Objekte macht die Kombination von Pfaden so einfach wie die Stringverkettung. Außerdem ist es sicherer als eine Stringverkettung oder die Methode `join()`. Betrachten Sie dazu die folgenden Beispiele:

```
>>> homeFolder = r'C:\Users\Al'
>>> subFolder = 'spam'
>>> homeFolder + '\\' + subFolder
'C:\\Users\\Al\\spam'
>>> '\\'.join([homeFolder, subFolder])
'C:\\Users\\Al\\spam'
```

Ein Skript, das diesen Code nutzt, ist nicht sicher, da die Backslashes nur auf Windows funktionieren. Sie könnten zwar eine `if`-Anweisung hinzufügen, die `sys.platform` überprüft (eine Systemvariable, die einen String mit der Angabe des Betriebssystems enthält), um zu entscheiden, welche Art von Schrägstrich zu verwenden ist, aber solchen Code überall dort einzufügen, wo er gebraucht wird, kann mühsam und fehleranfällig sein.

Das Modul `pathlib` löst all diese Probleme durch die Verwendung des Operators /, der die Pfade unabhängig vom Betriebssystem korrekt kombiniert. Im folgenden Beispiel werden damit dieselben Pfade konstruiert wie im vorherigen Code:

```
>>> homeFolder = Path('C:/Users/Al')
>>> subFolder = Path('spam')
>>> homeFolder / subFolder
WindowsPath('C:/Users/Al/spam')
>>> str(homeFolder / subFolder)
'C:\\Users\\Al\\spam'
```

Bei der Verwendung des Operators / zur Kombination von Pfaden müssen Sie lediglich darauf achten, dass einer der beiden ersten Werte ein `path`-Objekt ist. Python gibt eine Fehlermeldung aus, wenn Sie versuchen, Folgendes in die interaktive Shell einzugeben:

```
>>> 'spam' / 'bacon' / 'eggs'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Python wertet einen Ausdruck mit dem Operator / von links nach rechts zu einem `Path`-Objekt aus. Damit sich ein solches Objekt ergeben kann, muss entweder der erste oder der zweite Wert von links selbst ein `Path`-Objekt sein. Die Auswertung läuft wie in der folgenden Grafik gezeigt ab:

```
Path('spam)/'bacon'      /'eggs'/'ham'

WindowsPath('spam/bacon')/'eggs'/'ham'

WindowsPath('spam/bacon/eggs')  /'ham'

WindowsPath('spam/bacon/eggs/ham')
```

Wenn Sie die Fehlermeldung `TypeError: unsupported operand type(s) for /: 'str' and 'str'` erhalten, müssen Sie auf die linke Seite des Ausdrucks ein `Path`-Objekt stellen.

Der Operator / ersetzt die ältere Funktion `os.path.join()`, die auf *https://docs.python.org/3/library/os.path.html#os.path.join* genauer beschrieben wird.

Das aktuelle Arbeitsverzeichnis

Jedes Programm, das auf einem Computer läuft, hat ein *aktuelles Arbeitsverzeichnis*. Bei allen Dateinamen und Pfaden, die nicht mit dem Stammordner beginnen, wird angenommen, dass sie sich in diesem Arbeitsverzeichnis befinden.

Hinweis

Auch wenn »Ordner« die modernere Bezeichnung für ein Verzeichnis ist, lautet der übliche Begriff *aktuelles Arbeitsverzeichnis* und nicht etwa »aktueller Arbeitsordner«.

Um den String des aktuellen Arbeitsverzeichnisses abzurufen, verwenden Sie die Funktion `Path.cwd()` (wobei `cwd` für *current working directory* steht), und um es zu ändern die Funktion `os.chdir()`:

```
>>> from pathlib import Path
>>> import os
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')'
>>> os.chdir('C:\\Windows\\System32')
>>> Path.cwd()
WindowsPath('C:/Windows/System32')
```

Hier ist das aktuelle Arbeitsverzeichnis zunächst *C:\Users\Al\AppData\Local\Programs\Python\Python37*, weshalb der Dateiname *project.docx* auf *C:\Users\Al\AppData\Local\Programs\Python\Python37\project.docx* verweist. Wenn wir das aktuelle Arbeitsverzeichnis anschließend in *C:\Windows\System32* ändern, wird *project.docx* als *C:\Windows\System32\project.docx* interpretiert.

Wenn Sie versuchen, zu einem Verzeichnis zu wechseln, das gar nicht existiert, gibt Python eine Fehlermeldung aus:

```
>>> os.chdir('C:/ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:/ThisFolderDoesNotExist'
```

Es gibt keine `pathlib`-Funktion, um das Arbeitsverzeichnis zu ändern, da ein solcher Wechsel bei laufendem Programm zu schwer auffindbaren Bugs führen kann.

Die ältere Funktion, um das aktuelle Arbeitsverzeichnis als String abzurufen, ist `os.getcwd()`.

Das Benutzerverzeichnis

Alle Benutzer eines Computers haben einen Ordner für ihre eigenen Dateien, der als *Benutzerordner* oder *Benutzerverzeichnis* bezeichnet wird. Mit `Path.home()` können Sie ein `Path`-Objekt für diesen Ordner abrufen:

```
>>> Path.home()
WindowsPath('C:/Users/Al')
```

An welchem Speicherort sich die Benutzerverzeichnisse befinden, hängt vom Betriebssystem ab:

- Auf Windows befinden sich Benutzerverzeichnisse in *C:\Users*.
- Auf macOS befinden sich Benutzerverzeichnisse in */Users*.
- Auf Linux befinden sich Benutzerverzeichnisse meistens in */home*.

Da Ihre Skripte sehr wahrscheinlich die Berechtigung haben, Dateien in Ihrem Benutzerverzeichnis zu lesen und zu schreiben, stellt es den idealen Speicherort für die Dateien dar, mit denen Ihre Python-Programme arbeiten.

Absolute und relative Pfade

Es gibt zwei Möglichkeiten, um einen Dateipfad anzugeben:

- Als *absoluter Pfad*, der stets mit dem Stammordner beginnt.
- Als *relativer Pfad*, der relativ zum aktuellen Arbeitsverzeichnis des Programms angegeben ist.

In Pfaden finden Sie oft auch die Angaben . und .. . Dabei handelt es sich nicht um echte Ordner, sondern um besondere Bezeichnungen. Der einzelne Punkt . steht dabei für das vorliegende Verzeichnis, zwei Punkte .. für den Elternordner.

Abb. 9–2 zeigt Beispiele für Ordner und Dateien. Wenn *C:\bacon* das aktuelle Arbeitsverzeichnis ist, dann lauten die relativen Pfade für die dargestellten Ordner und Dateien wie in dem Bild angegeben.

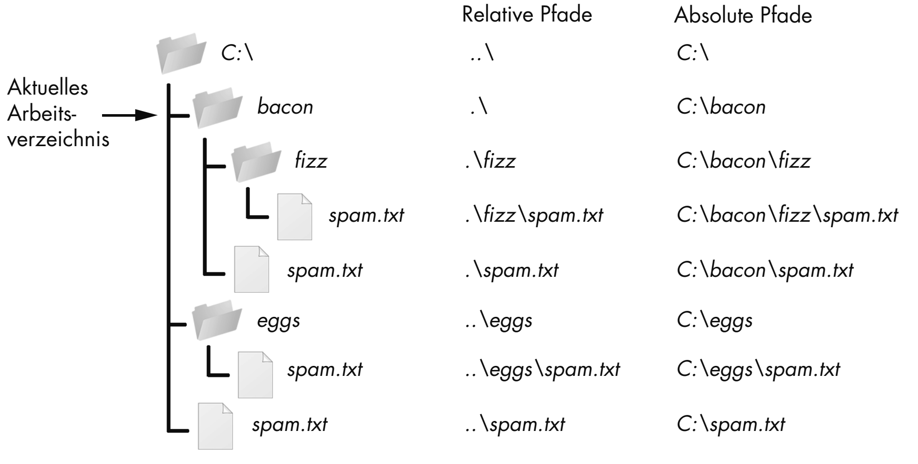


Abb. 9–2 *Relative und absolute Pfade für die Ordner und Dateien im Arbeitsverzeichnis C:\bacon*

Die Angabe .\ zu Beginn eines relativen Pfads ist optional. Beispielsweise verweisen sowohl *.\spam.txt* als auch *spam.txt* auf dieselbe Datei.

Neue Ordner mit os.makedirs() erstellen

Mithilfe der Funktion `os.makedirs()` können Ihre Programme neue Ordner erstellen (wobei sich *dirs* in dem Funktionsnamen auf die alternative Bezeichnung *directories* für Verzeichnisse bezieht):

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

Dadurch wird nicht nur der Ordner *C:\delicious* erstellt, sondern auch der Ordner *walnut* innerhalb von *C:\delicious* und der Ordner *waffles* innerhalb von *C:\delicious\walnut*. Die Funktion `os.makedirs()` erstellt also alle erforderlichen Zwischenordner, um dafür zu sorgen, dass der vollständige Pfad existiert. Diese Ordnerhierarchie sehen Sie in Abb. 9–3.

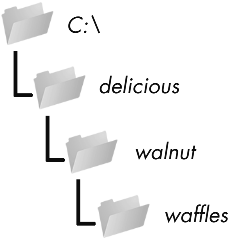


Abb. 9–3 *Das Ergebnis von* `os.makedirs('C:\\delicious\\walnut\\waffles')`

Um ein Verzeichnis aus einem `Path`-Objekt zu erstellen, rufen Sie die Methode `mkdir()` auf. Zum Beispiel legt der folgende Code den Ordner *spam* im Benutzerordner auf meinem Computer an:

```
>>> from pathlib import Path
>>> Path(r'C:\Users\Al\spam').mkdir()
```

Beachten Sie, dass `mkdir()` immer nur ein Verzeichnis auf einmal erstellt, also nicht mehrere Unterverzeichnisse wie `os.makedirs()`.

Absolute und relative Pfade verwenden

Das Modul `pathlib` enthält Methoden, um zu prüfen, ob ein gegebener Pfad ein absoluter Pfad ist, und um den absoluten Pfad zu einem relativen Pfad zurückzugeben.

Der Aufruf der Methode `is_absolute()` für ein `Path`-Objekt gibt `True` zurück, wenn das Objekt einen absoluten Pfad darstellt, anderenfalls `False`. Probieren Sie das wie folgt in der interaktiven Shell aus, wobei Sie jedoch Dateien und Ordner verwenden müssen, die auf Ihrem System vorhanden sind, und nicht die hier angegebenen:

```
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')
>>> Path.cwd().is_absolute()
True
>>> Path('spam/bacon/eggs').is_absolute()
False
```

Um den absoluten Pfad zu einem relativen zu erhalten, können Sie `Path.cwd() /` vor das `Path`-Objekt für den relativen Pfad stellen. Schließlich meinen wir mit »relativer Pfad« fast ausschließlich einen Pfad relativ zum aktuellen Arbeitsverzeichnis. Probieren Sie das wie folgt aus:

```
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.cwd() / Path('my/relative/path')
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37/my/relative/
path')
```

Ist der relative Pfad auf einen anderen Pfad als das aktuelle Arbeitsverzeichnis bezogen, ersetzen Sie einfach `Path.cwd()` durch diesen anderen Pfad. Das folgende Beispiel ruft einen absoluten Pfad im Benutzerverzeichnis statt im aktuellen Arbeitsverzeichnis ab:

```
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.home() / Path('my/relative/path')
WindowsPath('C:/Users/Al/my/relative/path')
```

Auch das Modul `os.path` enthält nützliche Funktionen für absolute und relative Pfade:

- Die Funktion `os.path.abspath(`*path*`)` gibt den absoluten Pfad des Arguments als String zurück. Das bietet eine einfache Möglichkeit, um einen relativen in einen absoluten Pfad umzuwandeln.
- Die Funktion `os.path.isabs(`*path*`)` gibt `True` zurück, wenn das Argument ein absoluter Pfad ist, und `False`, wenn es sich um einen relativen Pfad handelt.
- Die Funktion `os.path.relpath(`*path*`, `*start*`)` gibt den String des relativen Pfads vom Ausgangspunkt (*start*) bis zu *path* zurück. Wenn Sie *start* nicht angeben, wird das aktuelle Arbeitsverzeichnis als Ausgangspunkt genommen.

Probieren Sie diese Funktionen in der interaktiven Shell aus:

```
>>> os.path.abspath('.')
'C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37'
>>> os.path.abspath('.\\Scripts')
'C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37\\Scripts'
>>> os.path.isabs('.')
False
>>> os.path.isabs(os.path.abspath('.'))
True
```

Da *C:\Users\Al\AppData\Local\Programs\Python\Python37* beim Aufruf von `os.path.abspath()` das Arbeitsverzeichnis war, steht der Punkt für den absoluten Pfad `'C:\\Users\\Al\\AppData\\Local\\Programs\\Python\\Python37'`.

Probieren Sie auch folgende Aufrufe von `os.path.relpath()` in der interaktiven Shell aus:

```
>>> os.path.relpath('C:\\Windows', 'C:\\')
'Windows'
>>> os.path.relpath('C:\\Windows', 'C:\\spam\\eggs')
'..\\..\\Windows'
```

Wenn sich der relative Pfad zwar im selben Elternordner befindet wie der gegebene Pfad, aber im Unterverzeichnis eines anderen Pfades, z. B. `'C:\\Windows'` und `'C:\\spam\\eggs'`, können Sie mit der Zwei-Punkte-Schreibweise zu dem Elternordner zurückkehren.

Die Komponenten eines Dateipfads abrufen

Aus den Attributen eines gegebenen `Path`-Objekts können Sie die einzelnen Komponenten des Dateipfads als Strings abrufen. Das kann praktisch sein, um einen neuen Dateipfad auf der Grundlage eines vorhandenen zusammenzustellen. Die verschiedenen Attribute sehen Sie in Abb. 9–4.

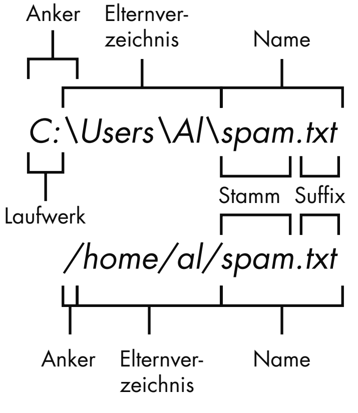


Abb. 9–4 *Komponenten von Dateipfaden auf Windows (oben) und macOS und Linux (unten)*

Ein Dateipfad setzt sich aus folgenden Komponenten zusammen:

- Der *Anker* (`anchor`) ist der Wurzelordner des Dateisystems (auf Windows Stammordner genannt).
- Auf Windows wird als *Laufwerksangabe* (`drive`) ein einzelner Buchstabe verwendet, der häufig eine physische Festplatte oder ein anderes Speichermedium bezeichnet.
- Der *Elternordner* (`parent`) ist das Verzeichnis, in dem sich die Datei befindet.
- Der *Name* (`name`) der Datei besteht aus dem *Stamm* (`stem`) oder *Grundnamen* (nicht zu verwechseln mit dem Stammordner auf Windows!) und dem *Suffix* (`suffix`), also der Erweiterung.

Windows-`Path`-Objekte haben das Attribut `drive`, macOS- und Linux-`Path`-Objekte dagegen nicht. Der erste Backslash ist im Attribut `drive` nicht enthalten.

Wie Sie die einzelnen Attribute aus dem Dateipfad abrufen, zeigen die folgenden Beispiele:

```
>>> p = Path('C:/Users/Al/spam.txt')
>>> p.anchor
'C:\\'
>>> p.parent # Dies ist ein Path-Objekt, kein String
WindowsPath('C:/Users/Al')
>>> p.name
'spam.txt'
>>> p.stem
'spam'
>>> p.suffix
'.txt'
>>> p.drive
'C:'
```

Der Abruf dieser Attribute ergibt jeweils einen einfachen String. Die einzige Ausnahme bildet `parent`, das zu einem weiteren `Path`-Objekt ausgewertet wird.

Das Attribut `parents` (nicht zu verwechseln mit `parent`!) wird mithilfe eines Integerindex zu den einzelnen Vorfahrenordnern des `Path`-Objekts ausgewertet:

```
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python37')
>>> Path.cwd().parents[0]
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python')
>>> Path.cwd().parents[1]
WindowsPath('C:/Users/Al/AppData/Local/Programs')
>>> Path.cwd().parents[2]
WindowsPath('C:/Users/Al/AppData/Local')
>>> Path.cwd().parents[3]
WindowsPath('C:/Users/Al/AppData')
>>> Path.cwd().parents[4]
WindowsPath('C:/Users/Al')
>>> Path.cwd().parents[5]
WindowsPath('C:/Users')
>>> Path.cwd().parents[6]
WindowsPath('C:/')
```

Das ältere Modul `os.path` verfügt über ähnliche Funktionen, um die einzelnen Teile eines Pfades als String zurückzugeben. So gibt `os.path.dirname(`*`path`*`)` einen String mit allem zurück, was vor dem letzten (umgekehrten) Schrägstrich im Argument `path` steht, die Funktion `os.path.basename(`*`path`*`)` dagegen einen String mit allem, was hinter diesem letzten Schrägstrich steht. Verzeichnisname (*dir name*) und Grundname (*base name*) eines Pfades sind in Abb. 9–5 dargestellt.

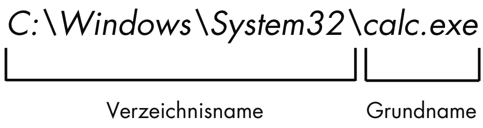


Abb. 9–5 *Der Grundname folgt auf den letzten Schrägstrich im Pfad und ist mit dem Dateinamen identisch. Alles, was links von dem letzten Schrägstrich steht, gehört dagegen zum Verzeichnisnamen.*

Geben Sie beispielsweise Folgendes in die interaktive Shell ein:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.basename(calcFilePath)
'calc.exe'
>>> os.path.dirname(calcFilePath)
'C:\\Windows\\System32'
```

Wenn Sie sowohl den Verzeichnis- als auch den Grundnamen eines Pfades benötigen, können Sie `os.path.split()` aufrufen, um einen Tupelwert mit diesen beiden Strings zu erhalten:

```
>>> calcFilePath = 'C:\\Windows\\System32\\calc.exe'
>>> os.path.split(calcFilePath)
('C:\\Windows\\System32', 'calc.exe')
```

Das gleiche Tupel können Sie auch erstellen, indem Sie `os.path.dirname()` und `os.path.basename()` aufrufen und die Rückgabewerte in ein Tupel aufnehmen:

```
>>> (os.path.dirname(calcFilePath), os.path.basename(calcFilePath))
('C:\\Windows\\System32', 'calc.exe')
```

Allerdings ist `os.path.split()` eine praktische Abkürzung.

Beachten Sie aber, dass es nicht möglich ist, `os.path.split()` einen Dateipfad zu übergeben und daraus eine Liste der Strings für die einzelnen Ordner zu gewinnen. Für diesen Zweck müssen Sie den Pfad mit der Stringmethode `split()` zerlegen und ihr die Variable `os.sep` übergeben (beachten Sie, dass sich `sep` in `os` befindet, nicht in `os.path`), in der das Ordnertrennzeichen für das vorliegende Betriebssystem gespeichert ist, also `'\\'` für Windows und `'/'` für macOS und Linux. Eine Aufteilung an diesem Trennzeichen ergibt eine Liste der einzelnen Ordner.

Betrachten Sie dazu das folgende Beispiel in der interaktiven Shell:

```
>>> calcFilePath.split(os.sep)
['C:', 'Windows', 'System32', 'calc.exe']
```

Dies gibt alle Teile des Pfades als Strings zurück.

Unter macOS und Linux steht am Anfang der zurückgegebenen Liste ein leerer String:

```
>>> '/usr/bin'.split(os.sep)
['', 'usr', 'bin']
```

Die Stringmethode `split()` gibt eine Liste der einzelnen Teile des Pfades zurück.

Dateigrößen und Ordnerinhalte ermitteln

Nachdem Sie nun wissen, wie Sie mit Dateipfaden umgehen müssen, können Sie Informationen über einzelne Dateien und Ordner abrufen. Das Modul `os` enthält Funktionen, mit denen Sie die Größe eines Ordners in Byte ermitteln und feststellen können, welche Dateien und Ordner sich innerhalb eines gegebenen Ordners befinden.

- Die Funktion `os.path.getsize(`*`path`*`)` gibt die Größe der im Argument `path` übergebenen Datei in Byte zurück.
- Die Funktion `os.listdir(`*`path`*`)` gibt eine Liste der Namenstrings aller Dateien und Ordner zurück, die sich in dem als *`path`* übergebenen Pfad befinden. (Beachten Sie, dass sich diese Funktion nicht im Modul `os.path`, sondern in `os` befindet.)

Wenn ich diese Funktionen in der interaktiven Shell ausprobiere, erhalte ich folgende Ergebnisse:

```
>>> os.path.getsize('C:\\Windows\\System32\\calc.exe')
27648
>>> os.listdir('C:\\Windows\\System32')
['0409', '12520437.cpx', '12520850.cpx', '5U877.ax', 'aaclient.dll',
-- schnipp --
'xwtpdui.dll', 'xwtpw32.dll', 'zh-CN', 'zh-HK', 'zh-TW', 'zipfldr.dll']
```

Wie Sie sehen, ist das Programm *calc.exe* auf meinem Computer 27.648 Bytes groß, und im Ordner *C:\Windows\system32* befinden sich eine ganze Menge Dateien. Wenn ich die Gesamtgröße aller Dateien in diesem Verzeichnis herausfinden möchte, kann ich `os.path.getsize()` und `os.listdir()` zusammen einsetzen:

```
>>> totalSize = 0
>>> for filename in os.listdir('C:\\Windows\\System32'):
   totalSize = totalSize +
   os.path.getsize(os.path.join('C:\\Windows\\System32', filename))

>>> print(totalSize)
2559970473
```

Dieser Code durchläuft alle Dateien im Ordner *C:\Windows\System32* und erhöht die Variable `totalSize` um die Größe der jeweiligen Datei. Beachten Sie, dass ich hier beim Aufruf von `os.path.getsize()` die Funktion `os.path.join()` verwende, um den Ordnernamen mit dem Namen der aktuellen Datei zu verknüpfen. Der von `os.path.getsize()` zurückgegebene Integer wird dann zu dem Wert in `totalSize` addiert. Nachdem alle Dateien durchlaufen wurden, gibt das Programm `totalSize` aus, um die Gesamtgröße aller Dateien im Ordner *C:\Windows\System32* anzuzeigen.

Eine Dateiliste mit Glob-Mustern bearbeiten

Wenn Sie mit bestimmten Dateien arbeiten wollen, geht das mit der Methode `glob()` einfacher als mit `listdir()`. `Path`-Objekte verfügen über die Methode `glob()`, die den Inhalt eines Ordners anhand von *Glob-Mustern* auflistet. Dabei handelt es sich um eine vereinfachte Form von regulären Ausdrücken, die oftmals an der Befehlszeile verwendet werden. Die Methode `glob()` gibt ein Generatorobjekt zurück (dessen Erklärung hier zu weit führen würde), das Sie zur einfachen Anzeige in der interaktiven Shell an `list()` übergeben müssen:

```
>>> p = Path('C:/Users/Al/Desktop')
>>> p.glob('*')
<generator object Path.glob at 0x000002A6E389DED0>
>>> list(p.glob('*')) # Erstellt eine Liste aus dem Generator.
[WindowsPath('C:/Users/Al/Desktop/1.png'), WindowsPath('C:/Users/Al/
Desktop/22-ap.pdf'), WindowsPath('C:/Users/Al/Desktop/cat.jpg'),
-- schnipp --
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

Das Sternchen steht für »mehrere beliebige Zeichen«, weshalb `p.glob('*')` einen Generator für alle Dateien in dem in `p` gespeicherten Pfad zurückgibt.

Ebenso wie bei regulären Ausdrücken können Sie auch anspruchsvolle Glob-Ausdrücke zusammenstellen:

```
>>> list(p.glob('*.txt') # Listet alle Textdateien auf
[WindowsPath('C:/Users/Al/Desktop/foo.txt'),
-- schnipp --
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

Das Glob-Muster `'*.txt'` gibt Dateien zurück, deren Namen mit einer beliebigen Kombination von Zeichen beginnen, aber mit dem String `'.txt'` enden, also der Erweiterung für Textdateien.

Im Gegensatz zum Sternchen steht das Fragezeichen nur für ein einzelnes beliebiges Zeichen:

```
>>> list(p.glob('project?.docx')
[WindowsPath('C:/Users/Al/Desktop/project1.docx'), WindowsPath('C:/Users/Al/
Desktop/project2.docx'),
-- schnipp --
WindowsPath('C:/Users/Al/Desktop/project9.docx')]
```

Der Glob-Ausdruck `'project?.docx'` findet sowohl `'project1.docx'` als auch `'project5.docx'`, aber nicht `'project10.docx'`, da `?` nur für ein einziges Zeichen steht und nicht für einen zweistelligen String wie `'10'`.

Um noch komplexere Glob-Ausdrücke zu erstellen, können Sie das Sternchen und das Fragezeichen auch kombinieren:

```
>>> list(p.glob('*.?x?')
[WindowsPath('C:/Users/Al/Desktop/calc.exe'), WindowsPath('C:/Users/Al/
Desktop/foo.txt'),
-- schnipp --

WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

Der Glob-Ausdruck `'*.?x?'` findet Dateien mit einem beliebigen Namen und einer dreistelligen Erweiterung, in deren Mitte ein `x` steht.

Wenn Sie mit `glob()` Dateien mit bestimmten Attributen auswählen, können Sie sehr leicht die Dateien in einem Verzeichnis angeben, an denen Sie bestimmte Operationen durchführen wollen. Dazu können Sie in einer `for`-Schleife den von `glob()` zurückgegebenen Generator durchlaufen:

```
>>> p = Path('C:/Users/Al/Desktop')
>>> for textFilePathObj in p.glob('*.txt'):
...     print(textFilePathObj) # Gibt das Path-Objekt als String aus
...     # Macht irgendetwas mit der Textdatei
...
C:\Users\Al\Desktop\foo.txt
C:\Users\Al\Desktop\spam.txt
C:\Users\Al\Desktop\zzz.txt
```

Wenn Sie eine Operation auf allen Dateien in einem Verzeichnis ausführen wollen, können Sie dazu `os.listdir(p)` oder `p.glob('*')` verwenden.

Die Gültigkeit von Pfaden prüfen

Viele Python-Funktionen werden mit einer Fehlermeldung beendet, wenn Sie ihnen einen Pfad übergeben, den es gar nicht gibt. Zum Glück verfügen `Path`-Objekte über Methoden, die prüfen, ob ein gegebener Pfad existiert und ob es sich dabei um eine Datei oder einen Ordner handelt. Unter der Voraussetzung, dass die Variable `p` ein `Path`-Objekt enthält, gilt Folgendes:

Speichern Sie den verketteten String in der Variablen `euroFilename` (❶) und übergeben Sie dann den ursprünglichen Dateinamen aus `amerFilename` und den neuen aus `euroFilename` an die Funktion `shutil.move()`, um die Datei umzubenennen (❸).

Der Aufruf von `shutil.move()` ist vorläufig auskommentiert; stattdessen werden die umzubenennenden und die neuen Dateinamen ausgegeben (❷). Auf diese Weise können Sie zunächst überprüfen, ob die Umbenennungsaktion auch tatsächlich richtig erfolgt. Nach diesem Test können Sie das Kommentarzeichen vor dem Aufruf von `shutil.move()` entfernen und das Programm erneut ausführen, um die Dateien tatsächlich umzubenennen.

Vorschläge für ähnliche Programme

Es gibt noch viele weitere Gründe, um größere Mengen von Dateien umzubenennen:

- Um dem Dateinamen ein Präfix voranzustellen, beispielsweise *spam_*, sodass *eggs.txt* in *spam_eggs.txt* geändert wird
- Um europäische Datumsangaben in Dateinamen in amerikanische umzuwandeln
- Um Nullen aus Dateinamen wie *spam0042.txt* zu entfernen

Projekt: Einen Ordner in einer ZIP-Datei sichern

Nehmen wir an, Sie haben alle Dateien eines Projekts im Ordner *C:\AlsPythonBook* gespeichert. Da Sie all die Arbeit, die Sie in das Projekt investiert haben, nur ungern verlieren möchten, wollen Sie »Momentaufnahmen« dieses Ordners in Form von ZIP-Dateien anlegen. Da Sie außerdem unterschiedliche Versionen aufbewahren möchten, sollen die Dateinamen dieser ZIP-Dateien eine laufende Nummer aufweisen, also etwa *AlsPythonBook_1.zip*, *AlsPythonBook_2.zip* usw. Das könnten Sie zwar auch manuell machen, allerdings ist das eine eher lästige Aufgabe. Außerdem besteht die Gefahr, dass Sie die ZIP-Dateien versehentlich falsch nummerieren. Es wäre viel einfacher, wenn ein Programm diese langweilige Arbeit für Sie übernehmen würde.

Öffnen Sie für dieses Projekt ein neues Dateieditorfenster und speichern Sie das Programm als *backupToZip.py*.

Schritt 1: Den Namen der ZIP-Datei bestimmen

In diesem Programm schreiben wir den Code in die Funktion `backupToZip()`, um die Komprimierung auch ganz einfach in anderen Programmen zur Verfügung stellen zu können. Am Ende des Programms wird die Funktion aufgerufen, um die Sicherung vorzunehmen. Schreiben Sie zunächst folgenden Code:

```
#! python3
# backupToZip.py - Kopiert einen Ordner und seinen gesamten Inhalt in eine
# ZIP-Datei mit laufender Nummerierung

import zipfile, os   ❶

def backupToZip(folder):
    # Sichert den ganzen Inhalt von "folder" in einer ZIP-Datei

    folder = os.path.abspath(folder) # "folder" muss ein absoluter Pfad sein

    # Ermittelt aufgrund der bereits vorhandenen Dateinamen den Namen für die
    # aktuelle Datei
    number = 1   ❷
    while True:   ❸
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

    # TODO: ZIP-Datei erstellen   ❹

    # TODO: Den Verzeichnisbaum durchlaufen und alle Dateien in allen Ordnern
    # komprimieren
    print('Done.')

backupToZip('C:\\delicious')
```

Als Erstes schreiben Sie als Grundgerüst die Shebang-Zeile (`#!`), eine Beschreibung, was das Programm macht, und die Befehle für den Import der Module `zipfile` und `os` (❶).

Anschließend definieren Sie die Funktion `backToZip()`. Ihr einziger Parameter ist der String `folder`, der den Pfad zu dem zu komprimierenden Ordner angibt. Die Funktion ermittelt den Dateinamen für die neue ZIP-Datei, erstellt diese, durchläuft den in `folder` angegebenen Ordner und fügt alle darin enthaltenen Unterordner und Dateien zu der ZIP-Datei hinzu. Für die anderen Schritte, die noch zu erledigen sind, geben Sie vorläufig `TODO`-Kommentare als Gedächtnisstütze im Code an (❹).

Für die Benennung der ZIP-Datei greift die Funktion auf den Grundnamen des in `folder` angegebenen absoluten Pfads zurück. Soll beispielsweise der Ordner *C:\delicious* gesichert werden, lautet der Name der ZIP-Datei *delicious_N.zip*, wobei *N* bei der ersten Ausführung des Programms *1* ist, bei der zweiten Ausführung *2* usw.

Den Wert von *N* ermitteln Sie, indem Sie prüfen, ob *delicious_1.zip* bereits vorhanden ist; wenn ja, müssen Sie nachsehen, ob es auch *delicious_2.zip* schon gibt, usw. Für *N* wird die Variable `number` verwendet (❷). In der Schleife, die `os.path.exists()` aufruft, um zu prüfen, ob die Dateinamen schon vorhanden sind, wird diese Variable ständig inkrementiert (❸). Bei dem ersten noch nicht vorhandenen Dateinamen wird die Schleife mit `break` abgebrochen. Damit haben wir den Dateinamen für das neue ZIP-Archiv gefunden.

Schritt 2: Die neue ZIP-Datei erstellen

Als Nächstes geht es daran, die ZIP-Datei tatsächlich anzulegen. Das Programm muss jetzt wie folgt aussehen:

```
#! python3
# backupToZip.py - Kopiert einen Ordner und seinen gesamten Inhalt in eine
# ZIP-Datei mit laufender Nummerierung

-- schnipp --

    while True:
        zipFilename = os.path.basename(folder) + '_' + str(number) + '.zip'
        if not os.path.exists(zipFilename):
            break
        number = number + 1

    # Erstellt die ZIP-Datei.
    print(f'Creating {zipFilename}...')
    backupZip = zipfile.ZipFile(zipFilename, 'w')    ❶

    # TODO: Den Verzeichnisbaum durchlaufen und alle Dateien in allen Ordnern
    # komprimieren
    print('Done.')

backupToZip('C:\\delicious')
```

Da der vorgesehene Name für die neue ZIP-Datei jetzt in der Variablen `zipFilename` gespeichert ist, können Sie `zipfile.ZipFile()` aufrufen, um das Archiv tatsächlich anzulegen (❶). Als zweites Argument müssen Sie dabei `'w'` übergeben, damit die ZIP-Datei im Schreibmodus geöffnet wird.

Schritt 3: Den Verzeichnisbaum durchlaufen und Inhalte zur ZIP-Datei hinzufügen

Nun müssen wir die Funktion `os.walk()` verwenden, um sämtliche Dateien in dem Ordner und allen seinen Unterordnern zu durchlaufen. Ergänzen Sie das Programm wie folgt:

```
#! python3
# backupToZip.py - Kopiert einen Ordner und seinen gesamten Inhalt in eine
# ZIP-Datei mit laufender Nummerierung

-- schnipp --

    # Durchläuft den Verzeichnisbaum und komprimiert alle Dateien in allen
    # Ordnern
    for foldername, subfolders, filenames in os.walk(folder):   ❶
        print(f'Adding files in {foldername}...')
        # Fügt den aktuellen Ordner zur ZIP-Datei hinzu
        backupZip.write(foldername)   ❷
        # Fügt alle Dateien in diesem Ordner zur ZIP-Datei hinzu

        for filename in filenames:   ❸
            newBase = os.path.basename(folder) + '_'
            if filename.startswith(newBase) and filename.endswith('.zip')
                continue # Keine Sicherung der ZIP-Sicherungsdateien!
            backupZip.write(os.path.join(foldername, filename))
    backupZip.close()
    print('Done.')

backupToZip('C:\\delicious')
```

Wenn Sie die Funktion `os.walk()` in einer `for`-Schleife verwenden (❶), gibt sie in jeder Iteration den Namen des aktuellen Ordners sowie die Namen der darin enthaltenen Dateien und Unterordner zurück.

In der `for`-Schleife wird der laufende Ordner zu der ZIP-Datei hinzugefügt (❷). Die verschachtelte `for`-Schleife geht alle Dateinamen in der Liste `filenames` durch (❸). Alle auf diese Weise gefundenen Dateien werden der ZIP-Datei hinzugefügt. Die einzigen Ausnahmen bilden zuvor angelegte ZIP-Archive.

Bei der ersten Ausführung gibt dieses Programm Folgendes aus:

```
Creating delicious_1.zip...
Adding files in C:\delicious...
Adding files in C:\delicious\cats...
Adding files in C:\delicious\waffles...
Adding files in C:\delicious\walnut...
Adding files in C:\delicious\walnut\waffles...
Done.
```