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/magic-8ball.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.

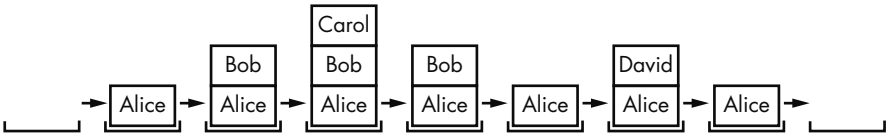

Abb. 3–1 *Der Stack einer Unterhaltung mit vielen Abschweifungen*

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.

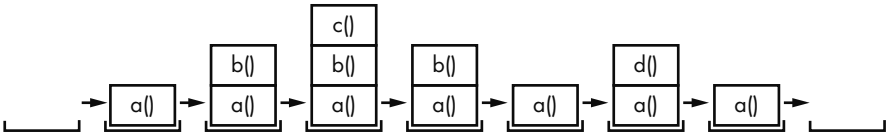

Abb. 3–2 *Frameobjekte im Aufrufstack im Verlauf der Funktionsaufrufe und Rücksprünge in abcdCallStack.py*

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
```