3. AUFLAGE

ROUTINEAUFGABEN MIT PYTHON AUTOMATISIEREN

PRAKTISCHE PROGRAMMIERLÖSUNGEN FÜR EINSTEIGER*INNEN

AL SWEIGART

no starch press

dpunkt.verlag

Inhalt

4
Funktionen

Eine Funktion ist wie ein Mini-Programm innerhalb eines Programms. Python stellt verschiedene eingebaute Funktionen bereit, wie etwa die Funktionen `print()`, `input()` und `len()` aus den vorherigen Kapiteln. Sie können jedoch auch eigene Funktionen schreiben. In diesem Kapitel werden Sie eigene Funktionen erstellen, den Call Stack untersuchen, der die Ausführungsreihenfolge der Funktionen im Programm festlegt, und mehr über den Gültigkeitsbereich von Variablen innerhalb und außerhalb von Funktionen erfahren.

Funktionen erstellen

Um besser zu verstehen, wie Funktionen funktionieren, erstellen wir eine solche. Geben Sie dieses Programm in den Dateieditor ein und speichern Sie es unter *helloFunc.py*:

```
def hallo():
  # Gibt drei Grüße aus
  print('Guten Morgen!')
  print('Guten Tag!')
  print('Guten Abend!')
hallo()
hallo()
```

```
print('NOCH EINMAL!')
hallo()
```

Die erste Zeile ist eine `def`-Anweisung, die eine Funktion mit dem Namen `hallo()` definiert. Der Code im Block, der auf die `def`-Anweisung folgt, bildet den Körper der Funktion. Dieser Code wird ausgeführt, wenn die Funktion aufgerufen wird, nicht bei der ersten Definition der Funktion.

Die `hallo()`-Zeilen nach der Funktion sind Funktionsaufrufe. Im Code ist ein Funktionsaufruf einfach der Name der Funktion, gefolgt von Klammern, wobei zwischen den Klammern verschiedene Argumente stehen können. Wenn die Programmausführung diese Aufrufe erreicht, springt sie zur ersten Zeile der Funktion und beginnt dort mit der Ausführung des Codes. Am Ende der Funktion kehrt die Ausführung zu der Zeile zurück, welche die Funktion aufgerufen hat, und fährt anschließend wie gewohnt fort.

Da dieses Programm `hallo()` dreimal aufruft, wird der Code in der Funktion `hallo()` ebenfalls dreimal ausgeführt. Wenn Sie dieses Programm ausführen, sieht die Ausgabe folgendermaßen aus:

```
Guten Morgen!
Guten Tag!
Guten Abend!
Guten Morgen!
Guten Tag!
Guten Abend!
NOCH EINMAL!
Guten Morgen!
Guten Tag!
Guten Abend!
```

Ein Hauptzweck von Funktionen besteht darin, Code zu bündeln, der mehrfach ausgeführt wird. Ohne eine definierte Funktion müssten Sie diesen Code jedes Mal kopieren und einfügen, wenn Sie ihn ausführen möchten; das Programm würde dann so aussehen:

```
print('Guten Morgen!')
print('Guten Tag!')
```

```
print('Guten Abend!')
print('Guten Morgen!')
print('Guten Tag!')
print('Guten Abend!')
print('NOCH EINMAL!')
print('Guten Morgen!')
print('Guten Tag!')
print('Guten Abend!')
```

Generell sollten Sie Codeduplizierungen vermeiden, da Sie bei einer späteren Änderung des Codes – etwa um einen Fehler zu beheben – daran denken müssen, den Code an allen kopierten Stellen zu aktualisieren.

Mit wachsender Programmiererfahrung werden Sie oft damit beschäftigt sein, *Duplizierungen zu entfernen*, also kopierten und eingefügten Code zu beseitigen. Die Entfernung von Duplizierungen macht Ihre Programme kürzer, besser lesbar und einfacher zu aktualisieren.

Argumente und Parameter

Wenn Sie die Funktion `print()` oder `len()` aufrufen, übergeben Sie ihr Werte, sogenannte *Argumente*, indem Sie diese in die Klammern eintragen. Sie können auch eigene Funktionen definieren, die Argumente annehmen. Geben Sie dieses Beispiel in den Dateieditor ein und speichern Sie es als *helloFunc2.py*:

```
def sage_hallo_zu(name): ❶
  # Gibt drei Grüße für den angegebenen Namen aus
  print('Guten Morgen, ' + name) ❷
  print('Guten Tag, ' + name)
  print('Guten Abend, ' + name)
sage_hallo_zu('Alice') ❸
sage_hallo_zu('Bob')
```

Wenn Sie dieses Programm ausführen, sieht die Ausgabe folgendermaßen aus:

```
Guten Morgen, Alice
Guten Tag, Alice
```

```
Guten Abend, Alice
Guten Morgen, Bob
Guten Tag, Bob
Guten Abend, Bob
```

Die Definition der `sage_hallo_zu()`-Funktion besitzt einen Parameter namens `name` ❶. Parameter sind Variablen, die Argumente enthalten. Wenn eine Funktion mit Argumenten aufgerufen wird, werden diese Argumente in den Parametern gespeichert. Beim ersten Aufruf der Funktion `sage_hallo_zu()` wird ihr das Argument `'Alice'` ❷ übergeben. Die Programmausführung tritt in die Funktion ein und der Parameter `name` wird automatisch auf `'Alice'` gesetzt, was die `print()`-Anweisung ausgibt ❸. Sie sollten Parameter in Ihrer Funktion verwenden, wenn sie abhängig von den Werten, die Sie beim Funktionsaufruf übergeben, leicht unterschiedliche Anweisungen ausführen soll.

Eine Besonderheit von Parametern ist, dass der im Parameter gespeicherte Wert vergessen wird, sobald die Funktion zurückkehrt. Wenn Sie beispielsweise `print(name)` nach `sage_hallo_zu('Bob')` im vorherigen Programm hinzufügen, würde das Programm einen Fehler anzeigen, da es keine Variable mit dem Namen `name` gibt. Diese Variable wird nach dem Funktionsaufruf `sage_hallo_zu('Bob')` zerstört, sodass sich `print(name)` auf eine `name`-Variable bezieht, die nicht existiert.

Die Begriffe *definieren, aufrufen, übergeben, Argument* und *Parameter* können verwirrend sein. Um deren Bedeutungen zu rekapitulieren, betrachten Sie folgendes Codebeispiel:

```
def sage_hallo_zu(name): ❶
  # Gibt drei Grüße für den angegebenen Namen aus
  print('Guten Morgen, ' + name)
  print('Guten Tag, ' + name)
  print('Guten Abend, ' + name)
sage_hallo_zu('Al') ❷
```

Eine Funktion zu *definieren* bedeutet, sie zu erstellen, ebenso wie eine Zuweisungsanweisung wie `spam = 42` die Variable `spam` erzeugt. Die `def`-Anweisung definiert die Funktion `sage_hallo_zu()` ❶. Der *Aufruf* der erstellten Funktion geschieht mit der Zeile `sage_hallo_zu('Al')` ❷. Das

übergibt die Ausführung an den Beginn des Funktionscodes. Mit diesem Funktionsaufruf wird der Funktion der String `'Al'` *übergeben*. Ein Wert, der in einem Funktionsaufruf übergeben wird, ist ein *Argument*. Argumente werden lokalen Variablen zugewiesen, die *Parameter* genannt werden. Das Argument `'Al'` wird dem Parameter `name` zugewiesen.

Diese Begriffe kann man leicht verwechseln. Wenn Sie sie jedoch auseinanderhalten können, wissen Sie genau, was mit dem Text in diesem Kapitel gemeint ist.

Rückgabewerte und return-Anweisungen

Wenn Sie die Funktion `len()` aufrufen und ihr ein Argument wie `'Hello'` übergeben, ergibt der Funktionsaufruf den Ganzzahlwert `5`, was der Länge des Strings entspricht, den Sie übergeben haben. Allgemein bezeichnet man den Wert, zu dem ein Funktionsaufruf ausgewertet wird, als den *Rückgabewert* der Funktion.

Wenn Sie eine Funktion mit der `def`-Anweisung erstellen, können Sie den Rückgabewert mit einer `return`-Anweisung festlegen, die aus Folgendem besteht:

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

Handelt es sich um einen Ausdruck, ist der Rückgabewert das Ergebnis, zu dem dieser Ausdruck ausgewertet wird. Das folgende Programm definiert beispielsweise eine Funktion, die – abhängig von der als Argument übergebenen Zahl – einen unterschiedlichen String zurückgibt. Geben Sie diesen Code in den Dateieditor ein und speichern Sie ihn als *magic8Ball.py*:

```
import random ❶

def hole_antwort(antwort_zahl): ❷
    # Weissagt eine Antwort basierend auf dem int
    antwort_zahl, 1 bis 9
    if antwort_zahl == 1: ❸
        return 'Es ist sicher'
    elif antwort_zahl == 2:
        return 'Es ist eindeutig so'
    elif antwort_zahl == 3:
```

```
        return 'Ja'
    elif antwort_zahl == 4:
        return 'Antwort unklar, versuche es erneut'
    elif antwort_zahl == 5:
        return 'Frage später erneut'
    elif antwort_zahl == 6:
        return 'Konzentriere dich und frage erneut'
    elif antwort_zahl == 7:
        return 'Meine Antwort ist nein'
    elif antwort_zahl == 8:
        return 'Aussichten nicht so gut'
    elif antwort_zahl == 9:
        return 'Sehr zweifelhaft'
print('Stelle eine Ja-oder-Nein-Frage:')
input('>')
r = random.randint(1, 9) ❹
weissagung = hole_antwort(r) ❺
print(weissagung) ❻
```

Wenn das Programm startet, importiert Python zunächst das `random`-Modul ❶. Anschließend erfolgt die Definition der `hole_antwort()`-Funktion ❷. Da die Funktion nicht sofort aufgerufen wird, wird der Code darin nicht ausgeführt. Als Nächstes ruft das Programm die `random.randint()`-Funktion mit zwei Argumenten auf: `1` und `9` ❹. Diese Funktion ermittelt eine zufällige Ganzzahl zwischen `1` und `9` (einschließlich `1` und `9`) und speichert sie in einer Variablen namens `r`.

Jetzt ruft das Programm die `hole_antwort()`-Funktion mit `r` als Argument ❺ auf. Die Programmausführung springt dann an den Beginn dieser Funktion ❸ und legt den Wert von `r` in einem Parameter mit dem Namen `antwort_zahl` ab. Dann gibt die Funktion abhängig vom Wert in `antwort_zahl` einen von vielen möglichen Strings zurück. Die Ausführung kehrt zu der Zeile am Ende des Programms zurück, die ursprünglich `hole_antwort()` ❺ aufgerufen hat, und weist den zurückgegebenen String

einer Variablen namens `weissagung` zu, die anschließend an einen `print()`-Aufruf ❻ übergeben und auf dem Bildschirm ausgegeben wird.

Beachten Sie, dass Sie Rückgabewerte als Argumente an andere Funktionsaufrufe übergeben können. Dadurch können Sie diese drei Zeilen abkürzen:

```
r = random.randint(1, 9)
weissagung = hole_antwort(r)
print(weissagung)
```

zu dieser einzelnen, aber gleichwertigen Zeile:

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

Denken Sie daran, dass Ausdrücke aus Werten und Operatoren bestehen; Sie können einen Funktionsaufruf in einem Ausdruck verwenden, da der Aufruf zu seinem Rückgabewert ausgewertet wird.

Der Wert »None«

In Python steht der Wert `None` für das Fehlen eines Werts. `None` ist der einzige Wert des Datentyps `NoneType`. (Andere Programmiersprachen könnten diesen Wert `null`, `nil` oder `undefined` nennen.) Genauso wie die booleschen Werte `True` und `False` müssen Sie `None` stets mit einem großen N schreiben.

Dieser »Wert ohne Wert« kann hilfreich sein, wenn Sie etwas speichern müssen, das in einer Variablen nicht mit einem echten Wert verwechselt werden soll. Ein Anwendungsfall für `None` ist der Rückgabewert von `print()`. Die Funktion `print()` gibt Text auf dem Bildschirm aus und muss, anders als etwa `len()` oder `input()`, keinen Wert zurückgeben. Da jedoch alle Funktionsaufrufe einen Rückgabewert liefern müssen, gibt `print()` `None` zurück. Um dies in der Praxis zu sehen, geben Sie bitte Folgendes in die interaktive Shell ein:

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

Im Hintergrund fügt Python `return None` ans Ende jeder Funktionsdefinition ohne `return`-Anweisung hinzu. Dieses Verhalten ähnelt der Art und Weise, wie

eine `while`- oder `for`-Schleife implizit mit einer `continue`-Anweisung endet. Funktionen geben ebenfalls `None` zurück, wenn Sie eine `return`-Anweisung ohne Wert verwenden (also nur das Schlüsselwort `return` allein).

Benannte Parameter

Python identifiziert die meisten Argumente anhand ihrer Position im Funktionsaufruf. Zum Beispiel ist `random.randint(1, 10)` etwas anderes als `random.randint(10, 1)`. Der erste Aufruf gibt eine zufällige Ganzzahl zwischen `1` und `10` zurück, da das erste Argument das untere Ende des Bereichs bestimmt und das nächste Argument das obere Ende festlegt, während der zweite Funktionsaufruf einen Fehler verursacht.

Andererseits erkennt Python *benannte Parameter* am Namen, der ihnen im Funktionsaufruf vorangestellt wird. Sie werden auch hören, dass benannte Parameter als Schlüsselwortparameter oder Schlüsselwortargumente bezeichnet werden (*keyword arguments*), obwohl sie nichts mit den Python-Schlüsselwörtern, also »keywords«, zu tun haben. Programmierer verwenden benannte Parameter häufig, um optionale Argumente bereitzustellen. Beispielsweise verwendet die Funktion `print()` die optionalen Parameter `end` und `sep`, um festzulegen, welches Trennzeichen am Ende beziehungsweise zwischen den Argumenten ausgegeben wird. Wenn Sie ein Programm ohne diese Argumente ausführen,

```
print('Hallo')
print('World')
```

würde die Ausgabe folgendermaßen aussehen:

```
Hallo
World
```

Die beiden Strings erscheinen auf getrennten Zeilen, da die Funktion `print()` automatisch ein Neue-Zeile-Zeichen `'\n'` an das Ende des Strings anhängt. Sie können jedoch den benannten Parameter `end` setzen, um das neue Zeilenendezeichen durch einen anderen String zu ersetzen. Wenn der Code beispielsweise so aussieht:

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

würde die Ausgabe folgendermaßen aussehen:

```
HalloWorld
```

Die Ausgabe erscheint in einer einzigen Zeile, weil Python nach `'Hallo'` keine neue Zeile mehr ausgibt. Stattdessen wird ein leerer String ausgegeben. Dies ist nützlich, wenn Sie den Umbruch in eine neue Zeile deaktivieren möchten, der am Ende jedes Funktionsaufrufs von `print()` passiert. Angenommen, Sie möchten die Ergebnisse einer Serie von Münzwürfen – Kopf oder Zahl – ausgeben. Wenn Sie die Ausgabe in einer einzigen Zeile darstellen, wirkt das Ergebnis ansprechender, wie in diesem *coinflip.py*-Programm:

```
import random
for i in range(100): # Führe 100 Münzwürfe aus.
  if random.randint(0, 1) == 0:
    print('K', end=' ')
  else:
    print('Z', end=' ')
print() # Gib am Ende eine neue Zeile aus.
```

Dieses Programm zeigt die K- und Z-Ergebnisse kompakt in einer einzigen Zeile an, anstatt jedes K- oder Z-Ergebnis einzeln pro Zeile auszugeben:

```
Z K Z Z Z K K Z Z Z Z K K K K Z K K Z Z Z Z Z K Z
Z Z Z Z K Z Z Z Z Z K Z K Z K K K Z Z K Z Z Z Z K
Z K K K Z K K Z K Z Z Z Z Z K Z Z K Z Z Z Z K Z K
K K Z Z Z Z K Z Z Z Z K K K Z K Z K K K K Z K K Z
```

Wenn Sie ebenso mehrere String-Werte an `print()` übergeben, trennt die Funktion diese automatisch durch ein einzelnes Leerzeichen. Um dieses Verhalten zu beobachten, geben Sie Folgendes in die interaktive Shell ein:

```
>>> print('Katzen', 'Hunde', 'Mäuse')
Katzen Hunde Mäuse
```

Sie können das standardmäßige Trennzeichen ersetzen, indem Sie dem benannten Parameter `sep` einen anderen String zuweisen. Geben Sie Folgendes in die interaktive Shell ein:

```
>>> print('Katzen', 'Hunde', 'Mäuse', sep=',')
Katzen,Hunde,Mäuse
```

Sie können benannte Parameter auch zu den Funktionen hinzufügen, die Sie selbst schreiben. Dafür müssen Sie jedoch zunächst die Liste und den Dictionary-Datentyp in Kapitel 6 und Kapitel 7 kennenlernen. Merken Sie sich zunächst, dass einige Funktionen optionale benannte Parameter besitzen, die Sie beim Funktionsaufruf angeben können.

Der Call Stack

Stellen Sie sich vor, Sie führen ein ausschweifendes Gespräch mit jemandem. Sie sprechen über Ihre Freundin Alice, was Sie wiederum an eine Geschichte über Ihren Kollegen Bob erinnert. Zuvor müssen Sie jedoch noch etwas zu Ihrer Cousine Carol erklären. Sie beenden die Geschichte über Carol und kehren dann zu Bob zurück. Wenn Sie Ihre Ausführungen zu Bob abgeschlossen haben, sprechen Sie wieder über Alice. Dann werden Sie an Ihren Bruder David erinnert, erzählen eine Geschichte über ihn und kehren anschließend nochmals zur ursprünglichen Geschichte über Alice zurück. Ihr Gespräch folgte einer Stapel-ähnlichen Struktur (engl. *stack*), wie in Abbildung 4.1. In einem *Stack* werden Elemente ausschließlich oben hinzugefügt oder entfernt, und das aktuelle Thema befindet sich stets an der Spitze des Stacks.

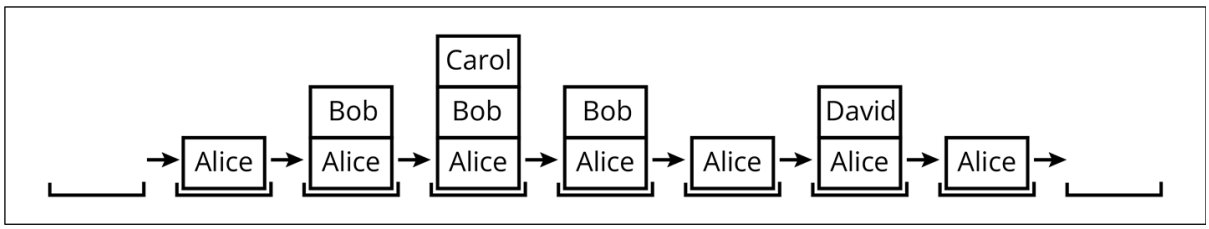

Abb. 4.1 *Ihr verschachtelter Gesprächs-Stack*

Wie bei Ihrem verschachtelten Gespräch führt ein Funktionsaufruf nicht dazu, dass die Ausführung auf direktem Weg nur an die Spitze einer Funktion gelangt. Python merkt sich, welche Codezeile die Funktion aufgerufen hat, sodass die Ausführung dorthin zurückkehren kann, wenn ein `return`-Statement erreicht wird. Falls die ursprüngliche Funktion weitere Funktionen aufgerufen hat, kehrt die Ausführung zunächst zu *diesen* Funktionsaufrufen zurück, bevor sie aus dem ursprünglichen Funktionsaufruf zurückkehrt. Der Funktionsaufruf an der Spitze des Stacks ist die aktuelle Position der Ausführung.

Öffnen Sie ein Dateieditor-Fenster und geben Sie den folgenden Code ein. Speichern Sie ihn anschließend als *abcdCallStack.py*:

```
def a():
  print('a() startet')
  b() ❶
```

```
    d() ❷
    print('a() kehrt zurück')
  def b():
    print('b() startet')
    c() ❸
    print('b() kehrt zurück')
  def c():
    print('c() startet') ❹
    print('c() kehrt zurück')
  def d():
    print('d() startet')
    print('d() kehrt zurück')
  a() ❺
```

Wenn Sie dieses Programm ausführen, wird die Ausgabe wie folgt aussehen:

```
a() startet
b() startet
c() startet
c() kehrt zurück
b() kehrt zurück
d() startet
d() kehrt zurück
a() kehrt zurück
```

Wenn `a()` aufgerufen wird ❺, ruft die Funktion `b()` ❶ auf, die wiederum `c()` ❸ aufruft. Die Funktion `c()` ruft nichts weiter auf, sie gibt lediglich `c() startet` ❹ und `c() kehrt zurück` aus, bevor sie zur Aufrufstelle in `b()` ❸ zurückkehrt. Sobald die Ausführung zu dem Code in `b()` zurückkehrt, der `c()` aufgerufen hat, kehrt sie in `a()` dorthin zurück, wo `b()` aufgerufen wurde ❶. Die Ausführung wird in der nächsten Zeile der Funktion `a()` fortgesetzt ❷, die einen Aufruf von `d()` darstellt. Wie die Funktion `c()` ruft auch `d()` nichts weiter auf. Sie gibt lediglich `d() startet` und `d() kehrt zurück` aus, bevor sie zur Aufrufstelle in `a()` zurückkehrt. Da `d()` keinen weiteren Code

enthält, kehrt die Ausführung zu der Stelle in `a()` zurück, an der `d()` aufgerufen wurde ❷. Die letzte Zeile in `a()` gibt `a() kehrt zurück` aus, bevor am Ende des Programms die Ausführung zum ursprünglichen Aufruf von `a()` ❺ zurückkehrt.

Der Call Stack ermöglicht es Python, nach jedem Funktionsaufruf zu wissen, wohin die Ausführung zurückkehren soll. Der Call Stack wird nicht in einer Variable Ihres Programms gespeichert, vielmehr ist er ein Bereich im Arbeitsspeicher Ihres Computers, den Python automatisch im Hintergrund verwaltet. Wenn Ihr Programm eine Funktion aufruft, legt Python ein *Stackframe* (oder *frame object*) oben auf dem Call Stack an. Stackframes speichern die Zeilennummer des ursprünglichen Funktionsaufrufs, damit Python weiß, wohin es zurückkehren soll. Wenn das Programm einen weiteren Funktionsaufruf ausführt, fügt Python ein weiteres Stackframe über dem vorherigen im Call Stack hinzu.

Wenn ein Funktionsaufruf zurückkehrt, entfernt Python ein Stackframe vom oberen Ende des Stacks und setzt die Ausführung an der darin gespeicherten Zeilennummer fort. Beachten Sie, dass Stackframes immer am oberen Ende des Stacks hinzugefügt und entfernt werden und niemals an einer anderen Stelle. Abbildung 4.2 veranschaulicht den Zustand des Call Stacks in *abcdCallStack.py*, während jede Funktion aufgerufen wird und zurückkehrt.

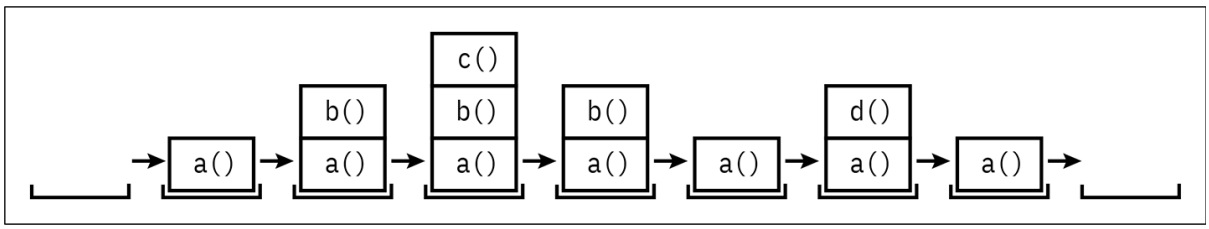

Abb. 4.2 *Die Frame-Objekte des Call Stacks, während abcdCallStack.py Funktionen aufruft und verlässt.*

Das oberste Element des Call Stacks ist die aktuell ausgeführte Funktion. Ist der Call Stack leer, befindet sich die Ausführung auf einer Zeile außerhalb aller Funktionen.

Den Call Stack müssen Sie nicht zwingend kennen, um Programme zu schreiben – er ist ein technisches Detail. Es genügt zu wissen, dass Funktionsaufrufe zur Zeilennummer zurückkehren, von der sie aufgerufen wurden. Ein Verständnis des Call Stacks erleichtert es jedoch, den lokalen und globalen Geltungsbereich zu verstehen, wie im nächsten Abschnitt beschrieben.

Lokaler und globaler Geltungsbereich

Nur Code innerhalb einer aufgerufenen Funktion kann auf die in dieser Funktion zugewiesenen Parameter und Variablen zugreifen. Diese Variablen befinden sich im *lokalen Gültigkeitsbereich* (engl. *local scope*) der Funktion. Im Gegensatz dazu kann Code an beliebiger Stelle im Programm auf Variablen zugreifen, die außerhalb aller Funktionen definiert wurden. Diese Variablen existieren im globalen Gültigkeitsbereich (engl. *global scope*). Eine Variable, die in einem lokalen Gültigkeitsbereich existiert, nennt man *lokale Variable*, während eine Variable imglobalen Gültigkeitsbereich als *globale Variable* bezeichnet wird. Eine Variable muss entweder das eine oder das andere sein, sie kann nicht zugleich lokal und global sein.

Stellen Sie sich einen Gültigkeitsbereich als einen Behälter für Variablen vor. Es gibt nur einen globalen Gültigkeitsbereich, der beim Start Ihres Programms angelegt wird. Beendet Ihr Programm die Ausführung, wird der globale Gültigkeitsbereich zerstört und alle darin enthaltenen Variablen werden verworfen. Ein neuer lokaler Gültigkeitsbereich entsteht, sobald ein Programm eine Funktion aufruft. Alle in der Funktion zugewiesenen Variablen existieren im lokalen Gültigkeitsbereich dieser Funktion. Wenn die Funktion zurückkehrt, wird der lokale Gültigkeitsbereich zusammen mit diesen Variablen aufgehoben.

Python verwendet Gültigkeitsbereiche, da so eine Funktion ihre eigenen Variablen anpassen kann und dennoch ausschließlich über ihre Parameter und ihren Rückgabewert mit dem restlichen Programm kommuniziert. Dadurch reduziert sich die Anzahl der Codezeilen, die möglicherweise einen Fehler verursachen. Wenn Ihr Programm ausschließlich aus globalen Variablen bestünde und ein Fehler durch eine Variable mit einem falschen Wert verursacht würde, hätten Sie möglicherweise Schwierigkeiten, die Quelle dieses fehlerhaften Wertes zu ermitteln. Sie könnte von überall im Programm gesetzt worden sein, das Hunderte oder Tausende Zeilen lang sein kann! Tritt der Fehler jedoch in einer lokalen Variablen auf, können Sie Ihre Suche auf eine einzelne Funktion beschränken.

Aus diesem Grund ist es zwar in kleinen Programmen unproblematisch, globale Variablen zu verwenden, doch es ist eine schlechte Angewohnheit, sich bei immer umfangreicheren Programmen auf globale Variablen zu verlassen.

Regeln zum Gültigkeitsbereich

Beachten Sie beim Arbeiten mit lokalen und globalen Variablen die folgenden Regeln:

- Code, der sich im globalen Gültigkeitsbereich außerhalb aller Funktionen befindet, kann keine lokalen Variablen verwenden.

- Code, der sich im lokalen Gültigkeitsbereich einer Funktion befindet, kann keine Variablen aus anderen lokalen Gültigkeitsbereichen verwenden.
- Code in einem lokalen Gültigkeitsbereich kann auf globale Variablen zugreifen.
- Sie können denselben Namen für verschiedene Variablen verwenden, sofern sie sich in unterschiedlichen Gültigkeitsbereichen befinden. Das heißt, es kann eine lokale Variable mit dem Namen `spam` und eine globale Variable ebenfalls mit dem Namen `spam` geben.

Lassen Sie uns diese Regeln anhand von Beispielen nachvollziehen.

Code im globalen Gültigkeitsbereich kann keine lokalen Variablen verwenden.

Betrachten Sie den folgenden Code, der beim Ausführen zu einem Fehler führt:

```
def spam():
  eggs = 'sss' ❶
spam()
print(eggs)
```

Die Ausgabe des Programms wird wie folgt aussehen:

```
Traceback (most recent call last):
  Datei “C:/test1.py”, line 4, in
    print(eggs)
NameError: name 'eggs' is not defined
```

Der Fehler tritt auf, weil die Variable `eggs` nur im lokalen Gültigkeitsbereich existiert, der beim Aufruf von `spam()` erzeugt wird ❶. Sobald die Programmausführung aus `spam()` zurückkehrt, wird dieser lokale Gültigkeitsbereich zerstört und die Variable `eggs` existiert nicht mehr. Wenn Ihr Programm dann versucht, `print(eggs)` auszuführen, gibt Ihnen Python einen Fehler zurück, der besagt, dass `eggs` nicht definiert ist. Das ist nachvollziehbar, wenn Sie darüber nachdenken: Befindet sich die Programmausführung im globalen Gültigkeitsbereich, existieren keine lokalen Gültigkeitsbereiche, sodass keine lokalen Variablen vorhanden sein können. Deshalb können Sie globale Variablen nur im globalen Gültigkeitsbereich referenzieren.

Code, der sich in einem lokalen Gültigkeitsbereich befindet, kann keine Variablen aus anderen lokalen Gültigkeitsbereichen verwenden.

Python erstellt jedes Mal einen neuen lokalen Gültigkeitsbereich, wenn ein Programm eine Funktion aufruft – auch dann, wenn diese Funktion von einer anderen Funktion aufgerufen wird. Betrachten Sie folgendes Programm:

```
def spam():
  eggs = 'SPAMSPAM'
  bacon() ❶
  print(eggs) # Gibt 'SPAMSPAM' aus ❷
def bacon():
  ham = 'hamham'
  eggs = 'BACONBACON'
spam() ❸
```

Zuerst ruft das Programm `spam()` auf ❸. Innerhalb dieser Funktion entsteht ein neuer Gültigkeitsbereich mit einer eigenen Variablen `eggs = 'SPAMSPAM'`. Danach wird `bacon()` ❶ aufgerufen. Auch hier legt Python einen neuen Gültigkeitsbereich an. In diesem gibt es zwei Variablen: `ham = 'hamham'` und eine neue, eigene `eggs = 'BACONBACON'`.

Beide `eggs`-Variablen existieren gleichzeitig, aber in verschiedenen Bereichen. Wenn `bacon()` endet, verschwindet seine lokale `eggs`-Variable wieder. Zurück in `spam()` existiert nur noch die dortige `eggs`-Variable mit dem Wert `'SPAMSPAM'`, die ausgegeben wird ❷.

Code, der sich im lokalen Geltungsbereich befindet, kann globale Variablen verwenden.

Bisher habe ich gezeigt, dass Code im globalen Geltungsbereich nicht auf Variablen im lokalen Geltungsbereich zugreifen kann; auch Code in einem anderen lokalen Geltungsbereich kann dies nicht. Betrachten Sie nun das folgende Programm:

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

Da die `spam()`-Funktion keinen Parameter namens `eggs` hat und auch kein Code existiert, der `eggs` einen Wert zuweist, betrachtet Python die Verwendung von `eggs` in dieser Funktion als Referenz auf die globale Variable `eggs`. Deshalb gibt das Programm beim Ausführen `'GLOBALGLOBAL'` aus.

Lokale und globale Variablen können denselben Namen haben.

Technisch gesehen ist es völlig zulässig, denselben Variablennamen sowohl für eine globale Variable als auch für lokale Variablen in unterschiedlichen Gültigkeitsbereichen zu verwenden. Wenn Sie sich das Leben erleichtern wollen, sollten Sie dies jedoch vermeiden. Um zu sehen, was passieren kann, geben Sie den folgenden Code in den Dateieditor ein und speichern Sie diesen 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, erscheint die folgende Ausgabe:

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

Dieses Programm enthält tatsächlich drei verschiedene Variablen, die jedoch alle `eggs` heißen, was zu Verwirrung führen kann. Die Variablen lauten wie folgt:

- eine Variable namens `eggs`, die im lokalen Gültigkeitsbereich existiert, wenn `spam()` aufgerufen wird ❶
- eine Variable namens `eggs`, die im lokalen Gültigkeitsbereich existiert, wenn `bacon()` aufgerufen wird ❷
- eine Variable namens `eggs`, die im globalen Gültigkeitsbereich existiert ❸

Da diese drei separaten Variablen denselben Namen tragen, kann es schwierig sein, jeweils nachzuvollziehen, welche gerade verwendet wird. Vergeben Sie stattdessen für alle Variablen eindeutige Namen, selbst wenn sie in verschiedenen Gültigkeitsbereichen auftreten.

Die global-Anweisung

Wenn Sie eine globale Variable innerhalb einer Funktion verändern müssen, verwenden Sie die Anweisung `global`. Wenn Sie zu Beginn einer Funktion eine Zeile wie `global eggs` einfügen, weist dies Python an: »In dieser Funktion bezieht sich `eggs` auf die globale Variable, daher soll keine lokale Variable mit diesem Namen erstellt werden.« Geben Sie zum 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) # Gibt 'spam' aus
```

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

```
spam
```

Da `eggs` am Anfang von `spam()` als `global` deklariert wird ❶, bewirkt die Zuweisung von `eggs` auf `'spam'` ❷ eine Änderung des Werts der globalen Variable `eggs`. Es wird niemals eine lokale `eggs`-Variable erstellt.

Identifikation des Gültigkeitsbereichs

Nutzen Sie diese vier Regeln, um festzustellen, ob eine Variable zum lokalen oder zum globalen Gültigkeitsbereich gehört:

1. Eine Variable im globalen Gültigkeitsbereich (also außerhalb aller Funktionen) ist immer eine globale Variable.
2. Eine Variable in einer Funktion mit einer `global`-Anweisung ist in dieser Funktion immer eine globale Variable.
3. Andernfalls gilt: Wenn eine Funktion eine Variable in einer Zuweisungsanweisung verwendet, handelt es sich um eine lokale Variable.
4. Wenn die Funktion jedoch eine Variable verwendet, jedoch nie in einer Zuweisungsanweisung, handelt es sich um eine globale Variable.

Um diese Regeln besser zu verstehen, finden Sie hier ein Beispielprogramm. Geben Sie den folgenden Code in den Dateieditor ein und speichern Sie ihn unter *sameNameLocalGlobal.py*:

```
def spam():
  global eggs ❶
  eggs = 'spam' # Dies ist die globale Variable.
def bacon():
  eggs = 'bacon' # Dies ist eine lokale Variable.
  ❷

def ham():
  print(eggs) # Dies ist die globale Variable. ❸
eggs = 'global' # Dies ist die globale Variable.
spam()
print(eggs)
```

In der Funktion `spam()` bezieht sich `eggs` auf die globale Variable `eggs`, da die Funktion eine `global`-Anweisung dafür enthält ❶. In `bacon()` ist `eggs` eine lokale Variable, weil die Funktion eine Zuweisungsanweisung für sie enthält ❷. In `ham()` ❸ ist `eggs` die globale Variable, da die Funktion weder eine Zuweisungsanweisung noch eine `global`-Anweisung für diese Variable enthält. Wenn Sie *sameNameLocalGlobal.py* ausführen, wird die Ausgabe wie folgt aussehen:

```
spam
```

Wenn Sie versuchen, in einer Funktion eine lokale Variable zu verwenden, bevor Sie ihr einen Wert zuweisen, wie im folgenden Programm, gibt Python Ihnen

einen Fehler aus. Um dies zu veranschaulichen, geben Sie Folgendes in den Dateieditor ein und speichern Sie es als sameNameError.py:

```
def spam():
  print(eggs) # FEHLER!
  eggs = 'spam local' ❶
eggs = 'global' ❷
spam()
```

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

```
Traceback (most recent call last):
  File “C:/sameNameError.py”, line 6, in
  spam()
File “C:/sameNameError.py”, line 2, in spam
  print(eggs) # FEHLER!
UnboundLocalError: local variable 'eggs' referenced
before assignment
```

Dieser Fehler tritt auf, weil Python erkennt, dass es in der `spam()`-Funktion eine Zuweisungsanweisung für `eggs` gibt ❶ und daher jede Erwähnung der Variable `eggs` in `spam()` als lokale Variable betrachtet. Da jedoch `print(eggs)` ausgeführt wird, bevor `eggs` ein Wert zugewiesen wurde, existiert die lokale Variable `eggs` nicht. Python greift in diesem Fall nicht auf die globale `eggs`-Variable zurück ❷.

Der Name des Fehlers, `UnboundLocalError`, kann etwas verwirrend sein. In Python bedeutet *binden* (engl. *binding*) so viel wie zuweisen. Dieser Fehler weist daher darauf hin, dass im Programm eine lokale Variable verwendet wurde, bevor ihr ein Wert zugewiesen war.

Funktionen als »Black Boxes«

Oft müssen Sie bei einer Funktion lediglich deren Eingaben (Parameter) und ihren Ausgabewert kennen; es ist nicht immer notwendig, sich mit den Details des Funktionscodes zu beschäftigen. Wenn Sie auf diese Weise abstrahiert über Funktionen nachdenken, spricht man oft davon, eine Funktion als »Black Box« zu behandeln.

Dieses Konzept ist grundlegend für die moderne Programmierung. In späteren Kapiteln dieses Buches werden Ihnen verschiedene Module mit Funktionen vorgestellt, die von anderen entwickelt wurden. Auch wenn Sie aus Interesse den Quellcode einsehen können, müssen Sie nicht verstehen, wie diese Funktionen intern arbeiten, um sie zu nutzen. Da das Schreiben von Funktionen ohne globale Variablen ratsam ist, müssen Sie sich in der Regel keine Sorgen machen, dass der Funktionscode mit dem restlichen Programm interagiert.

Ausnahmebehandlung

Wenn aktuell ein Fehler oder eine *Ausnahme* (engl. *exception*) in Ihrem Python-Programm auftritt, stürzt das gesamte Programm ab – es »crasht«. Das möchten Sie in realen Programmen vermeiden. Stattdessen soll das Programm Fehler erkennen, sie behandeln und anschließend weiterlaufen.

Betrachten Sie hierzu das folgende Programm, das einen Fehler durch Division durch null enthält. Öffnen Sie ein Dateieditor-Fenster und geben Sie den folgenden Code ein. Speichern Sie die Datei als *zeroDivide.py*:

```
def spam(teile_durch):
    return 42 / teile_durch
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Wir haben eine Funktion namens `spam` definiert, ihr einen Parameter übergeben und anschließend den Wert dieser Funktion mit verschiedenen Parametern ausgegeben, um das Verhalten zu beobachten. Das ist die Ausgabe, wenn Sie den vorangegangenen Code ausführen:

```
21.0
3.5
Traceback (most recent call last):
  File “C:/zeroDivide.py”, line 6, in
    print(spam(0))
  File “C:/zeroDivide.py”, line 2, in spam
```

```
    return 42 / teile_durch
ZeroDivisionError: division by zero
```

Ein `ZeroDivisionError` tritt immer dann auf, wenn Sie versuchen, eine Zahl durch null zu teilen. Anhand der in der Fehlermeldung angegebenen Zeilennummer erkennen Sie, dass die `return`-Anweisung in `spam()` einen Fehler verursacht. In den meisten Fällen weisen Ausnahmen auf einen Bug in Ihrem Code hin, den Sie beheben müssen. Manchmal kann man jedoch Ausnahmen voraussehen und sie behandeln. Beispielsweise lernen Sie in Kapitel 10, wie Sie Text aus Dateien lesen. Wenn Sie einen Dateinamen angeben, der nicht existiert, löst Python eine `FileNotFoundError`-Ausnahme aus. Sie können diese Ausnahme so behandeln, dass Sie den Benutzer bitten, den Dateinamen erneut einzugeben, anstatt das Programm aufgrund einer unbehandelten Ausnahme sofort abstürzen zu lassen.

Fehler lassen sich mit `try`- und `except`-Anweisungen behandeln. Der Code, der potenziell einen Fehler verursachen könnte, wird in einen `try`-Block gesetzt. Die Ausführung des Programms springt zum Beginn des nachfolgenden `except`-Blocks, wenn ein Fehler auftritt.

Sie können den zuvor gezeigten Code zur Division durch null in einen `try`-Block setzen und den Code zur Fehlerbehandlung in einen `except`-Block auslagern:

```
def spam(divide_by):
  try:
      # ZeroDivisionError-Code in diesem Block,
      führt nicht zum
Crash:
      return 42 / divide_by
  except ZeroDivisionError:
      # Bei einem ZeroDivisionError wird der Code
      in diesem Block ausgeführt:
  print('Fehler: Ungültiges Argument.')
print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Wenn der Code in einem `try`-Block einen Fehler verursacht, springt die Programmausführung sofort zum Code in den `except`-Block. Nach der Ausführung dieses Codes läuft das Programm wie gewohnt weiter. Tritt im `try`-Block keine Ausnahme auf, wird der Code im `except`-Block übersprungen. Die Ausgabe des vorherigen Programms lautet wie folgt:

```
21.0
3.5
Fehler: Ungültiges Argument.
None
42.0
```

Beachten Sie, dass auch alle Fehler, die bei Funktionsaufrufen innerhalb des `try`-Blocks auftreten, abgefangen werden. Schauen Sie sich das folgende Programm mit `spam()`-Aufrufen im `try`-Block an:

```
def spam(divide_by):
  return 42 / divide_by
try:
  print(spam(2))
  print(spam(12))
  print(spam(0))
  print(spam(1))
except ZeroDivisionError:
  print('Fehler: Ungültiges Argument.')
```

Wenn Sie dieses Programm ausführen, sieht die Ausgabe wie folgt aus:

```
21.0
3.5
Fehler: Ungültiges Argument.
```

Der Grund, warum `print(spam(1))` nie ausgeführt wird, liegt darin, dass das Programm nach dem Wechsel in den Code des `except`-Blocks nicht mehr zum `try`-Block zurückkehrt. Stattdessen fährt es regulär mit dem weiteren Programmablauf nach dem `except`-Block fort.

Ein kurzes Programm: Zigzag

Nutzen wir die bisherigen Programmierkonzepte, um ein kleines Animationsprogramm zu erstellen. Dieses Programm erzeugt ein Zickzack-Muster, das sich hin- und herbewegt, bis Sie es durch Drücken der Stopp-Schaltfläche im Mu-Editor oder durch Betätigen von Strg-C anhalten. Beim Ausführen des Programms erscheint die folgende Ausgabe:

```
        ********
       ********
      ********
     ********
    ********
   ********
     ********
      ********
       ********
        ********
```

Geben Sie den folgenden Quellcode in den Dateieditor ein und speichern Sie die Datei unter *zigzag.py*:

```
import time, sys
einzug = 0 # Wie viele Leerzeichen einrücken
einzug_erhöhen = True # Ob die Einrückung zunimmt
oder nicht
try:
  while True: # Die Hauptprogrammschleife
    print(' ' * einzug, end='')
    print('********')
    time.sleep(0.1) # Pause für 1/10s.
    if einzug_erhöhen:
      # Erhöhe die Anzahl der Leerzeichen:
      einzug = einzug + 1
      if einzug == 20:
```

```
            # Richtung wechseln:
            einzug_erhöhen = False
        else:
          # Verringern der Anzahl der Leerzeichen:
          einzug = einzug - 1
          if einzug == 0:
            # Richtung wechseln:
            einzug_erhöhen = True
    except KeyboardInterrupt:
        sys.exit()
```

Analysieren wir diesen Code Zeile für Zeile, beginnend ganz oben:

```
import time, sys
einzug = 0 # Wie viele Leerzeichen einrücken
einzug_erhöhen = True # Ob die Einrückung zunimmt
oder nicht
```

Zunächst importieren wir die Module `time` und `sys`. Unser Programm verwendet zwei Variablen: Die Variable `einzug` zeichnet auf, wie viele Leerzeichen als Einrückung vor dem Block aus acht Sternchen stehen. Die Variable `einzug_erhöhen` enthält einen booleschen Wert, der angibt, ob die Einrückung zunimmt oder abnimmt:

```
try:
  while True: # Die Hauptprogrammschleife
    print(' ' * einzug, end='')
    print('********')
    time.sleep(0.1) # Pause für 1/10s.
```

Anschließend fügen wir den restlichen Programmcode in eine `try`-Anweisung ein. Wenn man während der Ausführung eines Python-Programms Strg-C drückt, löst Python die Ausnahme `KeyboardInterrupt` aus. Ist keine `try-except`-Anweisung vorhanden, um diese Ausnahme abzufangen, stürzt das Programm mit einer unschönen Fehlermeldung ab. Wir möchten jedoch, dass unser Programm die Ausnahme `KeyboardInterrupt` sauber behandelt,

indem es `sys. exit()` aufruft. (Den Code, der dies ermöglicht, finden Sie in der `except`-Anweisung am Ende des Programms.)

Die `while True:`-Endlosschleife wiederholt die Anweisungen im Programm unendlich. Dabei verwenden wir `' ' * einzug`, um die richtige Anzahl von Leerzeichen für die Einrückung auszugeben. Damit nach diesen Leerzeichen nicht automatisch eine neue Zeile ausgegeben wird, übergeben wir auch `end=''` beim ersten `print()`-Aufruf. Ein zweiter `print()`-Aufruf gibt das Band aus Sternchen aus. Wir haben die `time.sleep()`-Funktion bisher noch nicht besprochen; hier reicht es zu wissen, dass sie eine Pause von einer Zehntelsekunde in das Programm einfügt:

```
  if einzug_erhöhen:
    # Erhöhe die Anzahl der Leerzeichen
    einzug = einzug + 1
    if einzug == 20:
      # Richtung wechseln:
      einzug_erhöhen = False
```

Als Nächstes passen wir die Anzahl der Einrückungen an, die beim nächsten Ausgeben der Sternchen verwendet wird. Ist `einzug_erhöhen` auf `True`, erhöhen wir `einzug` um `1`. Sobald jedoch die Einrückung `20` erreicht, verringern wir die Einrückung:

```
  else:
    # Verringern der Anzahl der Leerzeichen:
    einzug = einzug - 1
    if einzug == 0:
      # Richtung wechseln:
      einzug_erhöhen = True
```

Ist `einzug_erhöhen` gerade `False`, möchten wir eins von `einzug` subtrahieren. Sobald die Einrückung `0` erreicht, soll die Einrückung wieder erhöht werden. In jedem Fall springt die Programmausführung zurück an den Anfang der Hauptprogrammschleife, um die Sternchen erneut auszugeben.

Drückt man zu irgendeinem Zeitpunkt, während sich die Programmausführung im `try`-Block befindet, Strg-C, löst diese `except`-Anweisung die `KeyboardInterrupt`-Ausnahme aus und verarbeitet sie:

```
except KeyboardInterrupt:
    sys.exit()
```

Die Programmausführung wechselt in den `except`-Block, führt `sys.exit()` aus und beendet das Programm. So hat man, obwohl die Hauptprogrammschleife unendlich ist, eine Möglichkeit, das Programm zu beenden.

Ein kurzes Programm: Spike

Lassen Sie uns ein weiteres Scroll-Animationsprogramm erstellen. Dieses Programm verwendet String-Replikation und verschachtelte Schleifen, um Zacken (engl. *spikes*) zu zeichnen. Öffnen Sie eine neue Datei in Ihrem Code-Editor, geben Sie den folgenden Code ein und speichern Sie diese als *spike.py*:

```
import time, sys
try:
  while True: # Die Hauptprogrammschleife
    # Zeichnen von Linien mit zunehmender Länge:
    for i in range(1, 9):
      print('-' * (i * i))
      time.sleep(0.1)
    # Zeichnen von Linien mit abnehmender Länge:
    for i in range(7, 1, -1):
      print('-' * (i * i))
      time.sleep(0.1)
except KeyboardInterrupt:
  sys.exit()
```

Wenn Sie dieses Programm ausführen, zeichnet es wiederholt die folgende Zacke:

```
-
----
---------
----------------
```

```
------------------------
----------------------------------
--------------------------------------------------
-------------------------------------------------
------------
-----------------------------------------------
----------------------------------
------------------------
---------------
---------
----
```

Wie beim vorherigen Zickzack-Programm muss das Spike-Programm die Funktionen `time.sleep()` und `sys.exit()` aufrufen. Die erste Zeile importiert die Module `time` und `sys`. Ein `try`-Block fängt die `KeyboardInterrupt`-Ausnahme ab, die ausgelöst wird, wenn man Strg-C drückt. Eine Endlosschleife zeichnet dann fortlaufend Zacken.

Die erste `for`-Schleife zeichnet Spitzen mit zunehmender Größe:

```
# Zeichnen von Linien mit zunehmender Länge:
for i in range(1, 9):
  print('-' * (i * i))
  time.sleep(0.1)
```

Da die `i`-Variable nacheinander auf `1`, dann auf `2`, dann auf `3` usw. bis ausschließlich `9` gesetzt wird, vervielfacht der nachfolgende `print()`-Aufruf die `'-'`-Strings mit `1 * 1` (also: 1), dann `2 * 2` (also: 4), dann `3 * 3` (also: 9) und so weiter. Dieser Code erstellt somit Strings, die 1, 4, 9, 16, 25, 36, 49 und schließlich 64 Bindestriche lang sind. Indem wir die Strings quadratisch wachsen lassen, erzeugen wir die obere Hälfte der Spitze.

Um die untere Hälfte der Spitze zu zeichnen, benötigen wir eine weitere `for`-Schleife, die bewirkt, dass `i` bei 7 beginnt und dann bis `1` abnimmt, ohne die `1`:

```
# Zeichnen von Linien mit abnehmender Länge:
for i in range(7, 1, -1):
```

```
    print('-' * (i * i))
    time.sleep(0.1)
```

Sie können die Werte 9 und 7 in den beiden `for`-Schleifen anpassen, wenn Sie die Breite der Spitze verändern möchten. Der übrige Code funktioniert mit diesen neuen Werten weiterhin einwandfrei.

Zusammenfassung

Funktionen sind das wichtigste Mittel, um Ihren Code in logische Gruppen zu gliedern. Da Variablen in Funktionen in ihren eigenen lokalen Gültigkeitsbereichen existieren, kann der Code einer Funktion die Werte lokaler Variablen in anderen Funktionen nicht direkt beeinflussen. Dies begrenzt die Codeteile, die die Werte Ihrer Variablen ändern können, was beim Debuggen hilfreich ist.

Funktionen sind ein hervorragendes Werkzeug, um Ihren Code zu organisieren. Sie können sich Funktionen wie Blackboxen vorstellen: Sie haben Eingaben in Form von Parametern und Ausgaben in Form von Rückgabewerten, und der Code in ihnen beeinflusst keine Variablen in anderen Funktionen.

In den vorherigen Kapiteln konnte ein einziger Fehler dazu führen, dass Ihr Programm abstürzt. In diesem Kapitel haben Sie `try`- und `except`-Anweisungen kennengelernt, mit denen Code ausgeführt werden kann, wenn ein Fehler erkannt wurde. Dadurch werden Ihre Programme widerstandsfähiger gegenüber häufig auftretenden Fehlern.

Übungsfragen

1. Warum sind Funktionen in Ihren Programmen vorteilhaft?
2. Wann wird der Code in einer Funktion ausgeführt: wenn die Funktion definiert wird oder wenn sie aufgerufen wird?
3. Mit welcher Anweisung wird eine Funktion erstellt?
4. Was ist der Unterschied zwischen einer Funktion und einem Funktionsaufruf?
5. Wie viele globale Gültigkeitsbereiche hat ein Python-Programm? Wie viele lokale Gültigkeitsbereiche gibt es?
6. Was geschieht mit Variablen in einem lokalen Gültigkeitsbereich, wenn der Funktionsaufruf beendet wird?

7. Was ist ein Rückgabewert? Kann ein Rückgabewert Teil eines Ausdrucks sein?
8. Wenn eine Funktion keine `return`-Anweisung enthält, welcher Rückgabewert ergibt sich beim Aufruf dieser Funktion?
9. Wie können Sie eine Variable innerhalb einer Funktion dazu bringen, sich auf die globale Variable zu beziehen?
10. Welcher Datentyp besitzt `None`?
11. Was bewirkt die Anweisung `import heissenalleihrehaustiereeric`?
12. Wenn Sie im Modul `spam` eine Funktion namens `bacon()` haben, wie würden Sie diese nach dem Import von `spam` aufrufen?
13. Wie können Sie verhindern, dass ein Programm beim Auftreten eines Fehlers abstürzt?
14. Was gehört in den `try`-Block? Was gehört in den `except`-Block?
15. Schreiben Sie das folgende Programm in eine Datei mit dem Namen *notrandomdice.py* und führen Sie es aus. Warum gibt jeder Funktionsaufruf dieselbe Zahl zurück?

```
import random
zufallszahl = random.randint(1, 6)
def hole_zufallswurf():
    # Gibt eine zufällige Ganzzahl von 1 bis 6
    zurück
    return zufallszahl
print(hole_zufallswurf())
print(hole_zufallswurf())
print(hole_zufallswurf())
print(hole_zufallswurf())
```

Übungsprogramme

Schreiben Sie zur Übung Programme, die die folgenden Aufgaben ausführen.

Die Collatz-Sequenz

Schreiben Sie eine Funktion mit dem Namen `collatz()`, die einen Parameter mit dem Namen `zahl` besitzt. Ist `zahl` gerade, soll `collatz()` den Wert von `zahl // 2` ausgeben und zurückgeben. Ist `zahl` ungerade, soll `collatz()` das Ergebnis von `3 * zahl + 1` ausgeben und diesen Wert zurückgeben.

Schreiben Sie anschließend ein Programm, das den Benutzer eine Ganzzahl eingeben lässt und fortlaufend `collatz()` auf diese Zahl anwendet, bis die Funktion den Wert `1` zurückgibt. (Erstaunlicherweise funktioniert diese Folge tatsächlich für jede Ganzzahl; früher oder später gelangen Sie mit dieser Sequenz immer zur 1! Selbst Mathematiker wissen nicht genau, warum. Ihr Programm erforscht damit die sogenannte Collatz-Sequenz, die manchmal auch als »das einfachste Problem der Mathematik, das niemand lösen kann«, bezeichnet wird, siehe *https://youtu.be/094y1Z2wpJg*, Veritaserum, 2021.)

Denken Sie daran, den Rückgabewert von `input()` mit der Funktion `int()` in eine Ganzzahl umzuwandeln; andernfalls handelt es sich um einen String-Wert. Um die Ausgabe übersichtlicher zu gestalten, sollten die `print()`-Aufrufe, die die Zahlen ausgeben, den benannten Parameter `sep=' '` verwenden, damit alle Werte in einer Zeile erscheinen.

Die Ausgabe dieses Programms könnte beispielsweise wie folgt aussehen:

```
Bitte gib eine Zahl ein:
3
3 10 5 16 8 4 2 1
```

Hinweis

Eine Ganzzahl `zahl` ist gerade, wenn `zahl % 2 == 0`, und ungerade, wenn `zahl % 2 == 1`.

Eingabevalidierung

Fügen Sie dem vorherigen Projekt `try`- und `except`-Anweisungen hinzu, um zu überprüfen, ob man einen Nicht-Ganzzahl-String eingegeben hat. Normalerweise löst die Funktion `int()` einen `ValueError`-Fehler aus, wenn ihr ein Nicht-Ganzzahl-String übergeben wird, wie beispielsweise bei `int('lachen')`. Geben Sie im `except`-Block eine Nachricht für den Benutzer aus, dass eine Ganzzahl eingegeben werden muss.

Stichwortverzeichnis

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z