KAPITEL 3

Funktionen

Im vorherigen Kapitel haben wir verschiedene Python-eigene Funktionen wie `int` und `float` benutzt. Außerdem kamen einige vom `math`-Modul bereitgestellte Funktionen wie `sqrt` und `pow` zum Einsatz. In diesem Kapitel lernen Sie, Ihre eigenen Funktionen zu erstellen und auszuführen. Darüber hinaus werden wir sehen, wie sich Funktionen gegenseitig aufrufen können. Als Beispiele verwenden wir Texte von Monty-Python-Songs, um Ihnen ein wichtiges Leistungsmerkmal von Python zu zeigen – die Möglichkeit, eigene Funktionen zu schreiben, ist das Fundament der Programmierung.

In diesem Kapitel stellen wir außerdem eine neue Anweisung vor: die `for`-Schleife, mit der Berechnungen wiederholt werden können.

Neue Funktionen definieren

Eine **Funktionsdefinition** legt den Namen einer Funktion und die Reihenfolge der Anweisungen fest, die beim Aufruf der Funktion ausgeführt werden sollen. Hier ein Beispiel:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

`def` ist ein Schlüsselwort, das anzeigt, dass es sich hier um eine Funktionsdefinition handelt. Der Name der Funktion lautet `print_lyrics` (»Songtext ausgeben«). Für Funktionen gelten die gleichen Namensregeln wie für Variablennamen.

Die leeren runden Klammern hinter dem Namen bedeuten, dass diese Funktion keine Argumente übernimmt.

Die erste Zeile der Funktion wird **Header** (Kopfteil) genannt – der Rest wird als **Body** (Funktionskörper) bezeichnet. Der Header muss mit einem Doppelpunkt abgeschlossen und der Body muss eingerückt werden. Per Konvention wird um vier Leerzeichen eingerückt. Der Funktionskörper besteht aus zwei `print`-Anweisungen. Grundsätzlich kann der Body einer Funktion eine beliebige Anzahl von Anweisungen unterschiedlicher Art enthalten.

Die Definition einer Funktion erzeugt ein Funktionsobjekt, das Sie wie folgt anzeigen können:

```
print_lyrics
```

```
<function __main__.print_lyrics()>
```

Das bedeutet, dass `print_lyrics` eine Funktion (engl. *Function*) ist, die keine Argumente übernimmt (runde Klammern ohne Inhalt nach dem Funktionsnamen). `__main__` gibt den Namen des Moduls an, das `print_lyrics` enthält.

Nachdem wir eine Funktion definiert haben, können wir sie auf die gleiche Weise aufrufen wie die eingebauten Funktionen:

```
print_lyrics()
```

```
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Läuft die Funktion, werden die Anweisungen im Body ausgeführt, wodurch die ersten zwei Zeilen des Lumberjack-Songs ausgegeben werden.

Parameter

Sie wissen bereits, dass für einige Funktionen Argumente angegeben werden müssen. Beim Aufruf von `abs` übergeben Sie beispielsweise eine Zahl als Argument. Manche Funktionen können auch mehr als ein Argument übernehmen. `math.pow` benötigt zum Beispiel zwei Argumente: die Basis und den Exponenten.

Hier die Definition einer Funktion, die ein Argument übernimmt:

```
def print_twice(string):
    print(string)
    print(string)
```

Der Variablenname innerhalb der runden Klammern wird **Parameter** genannt. Beim Aufruf der Funktion wird der Wert des Arguments diesem Parameter zugewiesen. So können wir `print_twice` (»zweimal ausgeben«) etwa so aufrufen:

```
print_twice('Dennis Moore, ')
```

```
Dennis Moore,
Dennis Moore,
```

Die Ausführung dieser Funktion hat die gleichen Auswirkungen wie die Zuweisung des Arguments an eine Variable und die Ausführung des Bodys der Funktion, wie hier gezeigt:

```
string = 'Dennis Moore, '
print(string)
print(string)
```

```
Dennis Moore,
Dennis Moore,
```

Auch Variablen können als Argument verwendet werden:

```
line = 'Dennis Moore, '
print_twice(line)
```

```
Dennis Moore,
Dennis Moore,
```

In diesem Beispiel wird der Wert von `line` dem Parameter `string` zugewiesen.

Funktionen aufrufen

Sobald eine Funktion definiert ist, kann sie innerhalb einer anderen Funktion eingesetzt werden. Um das zu demonstrieren, schreiben wir Funktionen, die den Text von »The Spam Song« (*https://www.songfacts.com/lyrics/monty-python/the-spam-song*) ausgeben:

Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,

Wir beginnen mit der folgenden Funktion, die zwei Parameter übernimmt:

```
def repeat(word, n):
    print(word * n)
```

Mit dieser Funktion können wir die erste Zeile des Songs ausgeben, wie hier gezeigt:

```
spam = 'Spam, '
repeat(spam, 4)
```

```
Spam, Spam, Spam, Spam,
```

Um die ersten zwei Zeilen auszugeben, definieren wir eine neue Funktion, die `repeat` (»wiederholen«) verwendet:

```
def first_two_lines():
    repeat(spam, 4)
    repeat(spam, 4)
```

Diese können wir dann wie folgt aufrufen:

```
first_two_lines()
```

```
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
```

Um die letzten drei Zeilen auszugeben, können wir eine weitere Funktion definieren, die ebenfalls `repeat` verwendet:

```
def last_three_lines():
    repeat(spam, 2)
    print('(Lovely Spam, Wonderful Spam!)')
    repeat(spam, 2)
```

```
last_three_lines()
```

```
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,
```

Zum Schluss kombinieren wir die Einzelteile zu einer Funktion, die die gesamte Strophe ausgibt:

```
def print_verse():
    first_two_lines()
    last_three_lines()
```

```
print_verse()
```

```
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,
```

Bei der Ausführung von `print_verse` (»Strophe ausgeben«) wird die Funktion `first_two_lines` (»erste zwei Zeilen«) aufgerufen, die wiederum `repeat` aufruft, die ihrerseits `print` aufruft. Das sind ganz schön viele Funktionen.

Natürlich hätten wir die gleiche Sache auch mit weniger Funktionen erledigen können. In diesem Beispiel ging es aber darum, zu zeigen, wie Funktionen zusammenarbeiten können.

Wiederholung

Wenn Sie mehr als eine Zeile ausgeben wollen, können Sie auch eine `for`-Anweisung verwenden. Hier ein einfaches Beispiel:

```
for i in range(2):
    print(i)
```

```
0
1
```

Die erste Zeile der Funktion ist ein Header, der mit einem Doppelpunkt abgeschlossen wird.

Die folgende Zeile beginnt mit dem Schlüsselwort `for`, einer neuen Variablen namens `i` und einem weiteren Schlüsselwort namens `in`. Es verwendet die `range`-Funktion, um eine Folge von zwei Werten zu erzeugen: `0` und `1`. Wenn wir in Python zählen, beginnen wir in der Regel bei `0`.

Bei der Ausführung der `for`-Anweisung wird der erste von `range` erzeugte Wert der Variablen `i` zugewiesen. Danach wird die `print`-Funktion im Body der Schleife aufgerufen, wodurch `0` ausgegeben wird.

Am Ende des Bodys angekommen, springt Python wieder zurück zum Header, um den Code erneut auszuführen, was auch der Grund dafür ist, dass diese Anweisung als **Schleife** bezeichnet wird. Im zweiten Durchlauf wird `i` der nächste

Wert von `range` zugewiesen und ausgegeben. Weil dies der letzte von `range` erzeugte Wert ist, wird die Schleife danach beendet.

Hier sehen Sie, wie wir eine `for`-Schleife verwenden können, um zwei Strophen des Songs auszugeben:

```
for i in range(2):
    print("Verse", i)
    print_verse()
    print()
```

```
Verse 0
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,

Verse 1
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,
```

Eine `for`-Schleife kann auch innerhalb einer Funktion eingesetzt werden. Die Funktion `print_n_verses` übernimmt beispielsweise einen ganzzahligen Parameter namens `n` und gibt daraufhin die angegebene Anzahl von Strophen aus:

```
def print_n_verses(n):
    for i in range(n):
        print_verse()
        print()
```

In diesem Beispiel benutzen wir `i` nicht im Body der Schleife. Trotzdem muss der Header der Schleife einen Variablennamen enthalten.

Variablen und Parameter sind lokal

Wenn Sie innerhalb einer Funktion eine Variable anlegen, ist sie **lokal**. Das heißt, sie existiert nur innerhalb der Funktion. Die folgende Funktion übernimmt beispielsweise zwei Argumente, verkettet sie und gibt das Ergebnis zweimal aus:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

Hier ein Beispiel für die Verwendung der Funktion:

```
line1 = 'Always look on the '
line2 = 'bright side of life.'
cat_twice(line1, line2)
```

```
Always look on the bright side of life.
Always look on the bright side of life.
```

Bei der Ausführung von `cat_twice` (»zweimal verketten«) wird eine lokale Variable namens `cat` angelegt, die bei Beendigung der Funktion wieder gelöscht wird. Versuchen wir, sie danach auszugeben, erhalten wir einen `NameError`:

```
print(cat)
```

```
NameError: name 'cat' is not defined
```

Außerhalb der Funktion ist `cat` nicht definiert.

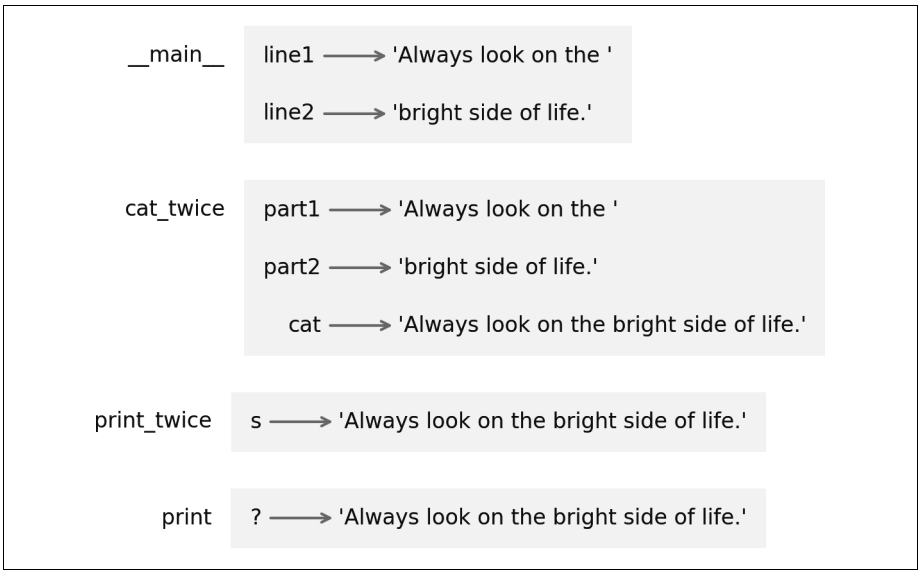
Auch Parameter sind lokal. So gibt es außerhalb von `cat_twice` nichts, worauf sich die Namen `part1` und `part2` beziehen.

Stack-Diagramme

Um mitzuverfolgen, welche Variablen wo verwendet werden können, kann es manchmal helfen, ein **Stack-Diagramm** zu zeichnen. Wie Zustandsdiagramme (siehe den Abschnitt »Zustandsdiagramme« auf Seite 38) zeigen Stack-Diagramme sowohl den Wert jeder Variablen als auch die Funktion, zu der sie gehören.

Dabei wird jede Funktion durch einen **Frame** (»Rahmen«) dargestellt. Das ist ein Kasten, der außen mit dem Funktionsnamen markiert wird und die Parameter und lokalen Variablen enthält.

Hier sehen Sie das Stack-Diagramm für das vorherige Beispiel:



Die Frames werden so angeordnet, dass erkennbar wird, welche Funktion eine andere aufruft. Lesen wir das Diagramm von unten nach oben, sehen wir, dass `print` von `print_twice` aufgerufen wurde, das seinerseits von `cat_twice` aufgerufen wurde, das wiederum von `__main__` aufgerufen wurde. Dabei ist `__main__` ein spezieller Name für den obersten Frame. Erstellen Sie eine Variable außerhalb einer Funktion, gehört sie standardmäßig zu `__main__`.

Das Fragezeichen (`?`) im Frame für `print` gibt an, dass wir den Namen des Parameters nicht kennen. Wenn Sie neugierig sind, fragen Sie einen virtuellen Assistenten: »Wie heißen die Parameter von Pythons `print`-Funktion?«

Tracebacks

Tritt zur Laufzeit ein Fehler in einer Funktion auf, gibt Python, dem Stack folgend, die Namen der ausgeführten Funktion, der von ihr aufgerufenen Funktion und so weiter aus.

Um Ihnen ein Beispiel zu geben, definiere ich hier eine fehlerhafte Version von `print_twice`. Sie versucht, `cat` auszugeben, das aber eine lokale Variable in einer anderen Funktion ist:

```
def print_twice(string):
    print(cat)          # NameError
    print(cat)
```

Hier sehen Sie, was passiert, wenn wir jetzt versuchen, `cat_twice` auszuführen:

```
cat_twice(line1, line2)

Traceback (most recent call last):

    File <string>:2

    Cell In[21], line 3 in cat_twice
      print_twice(cat)

    Cell In[26], line 2 in print_twice
      print(cat)            # NameError

NameError: name 'cat' is not defined
```

Die Fehlermeldung enthält einen **Traceback** (engl. für »Rückverfolgung«). Dieser gibt an, welche Funktion gerade lief, als der Fehler auftrat, die Funktion, die sie aufgerufen hat, und so weiter. In diesem Beispiel sieht man, dass `cat_twice` die Funktion `print_twice` aufgerufen hat und dass der Fehler in `print_twice` auftrat.

Die Reihenfolge der Funktionen im Traceback entspricht der Reihenfolge im Stack-Diagramm. Die ausgeführte Funktion befindet sich am unteren Ende.

Warum Funktionen?

Eventuell ist Ihnen nicht klar, warum es sich lohnt, ein Programm in Funktionen aufzuteilen. Hierfür gibt es mehrere Gründe:

- Die Erstellung einer neuen Funktion gibt Ihnen die Möglichkeit, eine Gruppe von Anweisungen mit einem Namen zu versehen, wodurch sie leichter lesbar und einfacher zu debuggen sind.
- Funktionen können ein Programm verkleinern, weil Code nicht wiederholt werden muss. Spätere Änderungen müssen nur an einem Ort vorgenommen werden.
- Durch die Unterteilung eines langen Programms in Funktionen können Sie die Einzelteile nacheinander debuggen und sie dann zu einem funktionierenden Ganzen zusammenfügen.
- Gut entworfene Funktionen können oft in mehreren Programmen genutzt werden. Ist eine Funktion einmal geschrieben und von Fehlern befreit, kann sie immer wieder eingesetzt werden.

Debugging

Das Debugging, also das Aufspüren und Beseitigen von Programmierfehlern, kann ziemlich frustrierend, aber auch herausfordernd und interessant sein. Manchmal macht es sogar Spaß. Und es ist eine der wichtigsten Fähigkeiten, die Sie erlernen sollten.

Auf gewisse Weise hat das Debugging etwas von Detektivarbeit. Sie bekommen Hinweise und müssen daraus die Ereignisse ableiten, die zu den Ergebnissen geführt haben, die Sie sehen.

Debugging ist außerdem eine experimentelle wissenschaftliche Arbeit. Sobald Sie eine Idee davon haben, was falsch gelaufen ist, passen Sie Ihr Programm an und versuchen es noch einmal. War Ihre Hypothese korrekt, können Sie das Ergebnis der Änderung vorhersagen und kommen so einem funktionierenden Programm einen Schritt näher. War Ihre Hypothese falsch, müssen Sie sich eine neue überlegen.

Für manche Menschen sind Programmierung und Debugging untrennbar verbunden. Man könnte sagen, Programmierung ist der Prozess, ein Programm nach und nach zu debuggen, bis es tut, was Sie wollen. Hierbei ist es sinnvoll, mit einem funktionierenden Programm zu starten. Daran nehmen Sie kleine Änderungen und Anpassungen vor, die Sie in kleinen Schritten debuggen können.

Wenn Sie feststellen, dass Sie viel Zeit mit Debugging verbringen, ist das oft ein Signal dafür, dass Sie zu viel Code schreiben, bevor Sie mit dem Testen beginnen. Es kann gut sein, dass Sie mit kleineren Schritten schneller vorankommen.

Glossar

Funktionsdefinition
: Eine Anweisung, die eine Funktion erstellt.

Header (Kopfteil)
: Die erste Zeile einer Funktionsdefinition.

Body (Körper)
: Die Folge der Anweisungen innerhalb einer Funktionsdefinition.

Funktionsobjekt
: Ein Wert, der von einer Funktionsdefinition erzeugt wird. Der Name der Funktion ist eine Variable, die auf das Funktionsobjekt verweist.

Parameter
: Ein innerhalb einer Funktion verwendeter Name, der auf den Wert verweist, der als Argument übergeben wurde.

Schleife
: Eine Anweisung, die eine oder mehrere Anweisungen einmal oder mehrmals ausführt.

Lokale Variable
: Eine innerhalb einer Funktion definierte Variable, auf die nur innerhalb dieser Funktion zugegriffen werden kann.

Stack-Diagramm
: Eine grafische Darstellung eines Stacks (Stapels) von Funktionen, ihrer Variablen und der Werte, auf die sie sich beziehen.

Frame (Rahmen)
: Ein Kasten in einem Stack-Diagramm, der für einen Funktionsaufruf steht. Er enthält die lokalen Variablen und Parameter der Funktion.

Traceback (Rückverfolgung)
: Eine Liste der ausgeführten Funktionen; sie wird ausgegeben, wenn eine Ausnahme auftritt.

Übungen

Fragen Sie einen virtuellen Assistenten

Per Konvention werden die Anweisungen in einer Funktion oder `for`-Schleife um jeweils vier Leerzeichen eingerückt. Allerdings ist nicht jeder mit dieser Konvention einverstanden. Wenn Sie mehr über die Geschichte der »großen Diskussion« erfahren möchten, fordern Sie einen virtuellen Assistenten auf: »Erzähle mir mehr über Leerzeichen und Tabulatoren in Python.«

Virtuelle Assistenten sind ziemlich gut darin, kleine Funktionen zu schreiben:

1. Weisen Sie Ihren Lieblingsassistenten an: »Schreibe eine Funktion namens `repeat`, die einen String und einen Integer-Wert übernimmt und den String so oft ausgibt, wie durch den Integer angegeben.«
2. Wenn das Ergebnis eine `for`-Schleife verwendet, können Sie fragen: »Kannst du das auch ohne eine `for`-Schleife?«

3. Suchen Sie sich eine beliebige andere Funktion aus diesem Kapitel aus und weisen Sie den virtuellen Assistenten an, sie zu erstellen. Ihre Herausforderung besteht darin, die Funktion genau genug zu beschreiben, um das gewünschte Ergebnis zu erhalten. Verwenden Sie hierfür das Vokabular, das Sie bisher in diesem Buch gelernt haben.

Virtuelle Assistenten sind auch ziemlich gut im Debuggen von Funktionen:

1. Fragen Sie einen virtuellen Assistenten, was mit dieser Version von `print_twice` nicht stimmt:

```
def print_twice(string):
    print(cat)
    print(cat)
```

Und wenn Sie bei einer der folgenden Übungen nicht weiterkommen, könnten Sie ebenfalls einen virtuellen Assistenten um Unterstützung bitten.

Übung 3-1

Schreiben Sie eine Funktion mit dem Namen `print_right` (»rechtsbündig ausgeben«). Sie soll einen String mit dem Namen `text` als Parameter übernehmen und den Text mit genügend vorangestellten Leerzeichen ausgeben, sodass der letzte Buchstabe jedes Strings jeweils auf der 40. Spalte des Terminals steht, wie unten gezeigt. Tipp: Verwenden Sie die `len`-Funktion, den String-Verkettungsoperator (+) und den String-Wiederholungsoperator (*).

Hier ein Beispiel, das zeigt, wie so etwas funktionieren könnte:

```
print_right("Monty")
print_right("Python's")
print_right("Flying Circus")
```

```
                                   Monty
                                Python's
                           Flying Circus
```

Übung 3-2

Schreiben Sie eine Funktion namens `triangle` (»Dreieck«), die einen String und einen Integer übernimmt und ein Dreieck mit der angegebenen Höhe zeichnet, das aus Kopien des Strings besteht. Hier ein Beispiel für ein Dreieck mit fünf Ebenen, das den String `'L'` verwendet:

```
triangle('L', 5)
```

```
L
LL
LLL
LLLL
LLLLL
```

Übung 3-3

Schreiben Sie eine Funktion mit dem Namen `rectangle` (»Rechteck«), die einen String und zwei Integer als Argumente übernimmt und ein Rechteck mit der angegebenen Breite und Höhe zeichnet, das aus Kopien des Strings besteht. Hier ein Beispiel für ein Rechteck der Breite 5 und der Höhe 4 unter Verwendung des Strings `'H'`;

```
rectangle('H', 5, 4)
```

```
HHHHH
HHHHH
HHHHH
HHHHH
```

Übung 3-4

Der Song »99 Bottles of Beer« beginnt mit diesen Zeilen:

> *99 bottles of beer on the wall,*
> *99 bottles of beer.*
> *Take one down, pass it around,*
> *98 bottles of beer on the wall.*

Die zweite Strophe folgt dem gleichen Muster. Allerdings beginnt sie mit 98 Flaschen Bier (*bottles of beer*) und endet mit 97. Der Song geht – für eine lange Zeit – weiter, bis die Zahl der Flaschen 0 (*zero*) erreicht.

Schreiben Sie eine Funktion mit dem Namen `bottle_verse`, der eine Zahl als Parameter übergeben wird und die eine Strophe ausgibt, die mit der angegebenen Zahl von Flaschen beginnt.

Tipp: Es könnte sinnvoll sein, zuerst eine Funktion zu schreiben, die die erste, zweite oder letzte Zeile ausgeben kann. Diese könnten Sie dann benutzen, um `bottle_verse` zu erstellen.

Benutzen Sie diesen Funktionsaufruf, um die erste Strophe auszugeben:

```
bottle_verse(99)
```

```
99 bottles of beer on the wall
99 bottles of beer
Take one down, pass it around
98 bottles of beer on the wall
```

Um den gesamten Text auszugeben, können Sie diese `for`-Schleife verwenden, die, bei 99 beginnend, bis 1 herunterzählt. Sie müssen dieses Beispiel nicht vollständig verstehen – später werden Sie mehr über `for`-Schleifen und die `range`-Funktion lernen.

```
for n in range(99, 0, -1):
    bottle_verse(n)
    print()
```

Inhalt