8
Funktionen

In diesem Kapitel lernen Sie, *Funktionen* zu schreiben. Dabei handelt es sich um benannte Codeblöcke, die für jeweils eine ganz bestimmte Aufgabe vorgesehen sind. Wenn Sie diese Aufgabe ausführen lassen möchten, rufen Sie die entsprechende Funktion auf. Muss eine Aufgabe mehrmals in einem Programm erledigt werden, brauchen Sie den Code dafür nicht wiederholt zu schreiben, sondern können einfach die dafür vorgesehene Funktion aufrufen. Python führt dann den Code in der Funktion aus. Wie Sie noch sehen werden, erleichtern Funktionen es Ihnen, Programme zu schreiben, zu lesen, zu testen und zu korrigieren.

Außerdem lernen Sie in diesem Kapitel Möglichkeiten kennen, um Informationen an Funktionen zu übergeben. Sie erfahren, wie Sie verschiedene Funktionen schreiben – etwa um Informationen anzuzeigen, Daten zu verarbeiten oder Werte zurückzugeben –, und wie Sie Funktionen in eigenen Dateien – sogenannten *Modulen* – speichern, um Ihre Hauptprogrammdateien besser zu gliedern.

Funktionen definieren

Die folgende einfache Funktion namens `greet_user()` gibt einen Gruß aus:

greeter.py

```
❶ def greet_user():
❷     """Display a simple greeting."""
❸     print("Hello!")

❹ greet_user()
```

Dieses Beispiel zeigt die einfachste Struktur einer Funktion. Mit dem Schlüsselwort `def` (❶) leiten Sie eine *Funktionsdefinition* ein, in der Sie Python den Namen der Funktion mitteilen und in den Klammern gegebenenfalls auch die Informationen angeben, die die Funktion benötigt, um ihre Aufgabe zu erledigen. Hier lautet der Name der Funktion `greet_user()`, und da sie keine weiteren Angaben für ihre Arbeit braucht, sind die Klammern leer (aber trotzdem erforderlich). Die Definition endet mit dem Doppelpunkt.

Alle eingerückten Zeilen hinter `def greet_user():` bilden den *Rumpf* der Funktion. Der Text bei ❷ ist ein *Docstring* (Dokumentationsstring), der beschreibt, was die Funktion tut. Docstrings sind in drei Paare von doppelten Anführungszeichen eingeschlossen. Python sucht nach diesen Zeichen, wenn es eine Dokumentation für die Funktionen in Ihren Programmen erstellt.

Die Zeile `print("Hello!")` bei ❸ ist die einzige echte Codezeile im Rumpf dieser Funktion. Dies ist die Aufgabe, die `greet_user()` erledigen soll.

Wenn Sie diese Funktion verwenden möchten, müssen Sie sie *aufrufen*. Dabei weisen Sie Python an, den Code in der Funktion auszuführen. Für einen solchen Funktionsaufruf müssen Sie lediglich den Namen der Funktion hinschreiben und gegebenenfalls alle erforderlichen zusätzlichen Informationen in den Klammern angeben. Da hier keine weiteren Angaben benötigt werden, rufen wir die Funktion bei ❹ einfach mit `greet_user()` auf. Wie erwartet gibt sie `Hello!` aus:

```
Hello!
```

Informationen an eine Funktion übergeben

Wir können unsere Funktion `greet_user()` so abändern, dass sie nicht nur ein einfaches `Hello!` ausgibt, sondern den Benutzer mit Namen begrüßt. Damit sie das tun kann, geben Sie bei der Funktionsdefinition `def greet_user()` in den Klammern `username` an. Das führt dazu, dass die Funktion bei jedem Aufruf erwartet, dass Sie in den Klammern einen Namen für `username` übergeben:

```
def greet_user(username):
    """Display a simple greeting."""
    print(f"Hello, {username.title()}!")

greet_user('jesse')
```

Durch `greet_user('jesse')` wird die Funktion `greet_user()` aufgerufen und ihr die Information übergeben, die sie benötigt, um die `print`-Anweisung auszuführen. Sie nimmt diesen Namen entgegen und zeigt den entsprechenden Gruß an:

```
Hello, Jesse!
```

Wenn Sie `greet_user('sarah')` schreiben, wird ebenfalls `greet_user()` aufgerufen, diesmal aber der Name `'sarah'` übergeben, weshalb die Ausgabe `Hello, Sarah!` lautet. Sie können `greet_user()` so oft aufrufen, wie Sie wollen, und dabei jeden beliebigen Namen übergeben und erzeugen dadurch jedes Mal eine vorhersagbare Ausgabe.

Argumente und Parameter

Im vorherigen Beispiel haben wir die Funktion `greet_user()` so definiert, dass sie einen Wert für die Variable `username` benötigt. Wenn wir die Funktion aufrufen und die entsprechende Information (einen Namen) übergeben, gibt sie die entsprechende Begrüßung aus.

Die Variable `username` in der Definition von `greet_user()` ist ein Beispiel für einen *Parameter*, also eine Information, die die Funktion benötigt, um ihre Aufgabe zu erfüllen. Der Wert `'jesse'` im Aufruf `greet_user('jesse')` wird als *Argument* bezeichnet. Dies ist die Information, die wir bei dem Aufruf an die Funktion in den Klammern übergeben. In diesem Fall übergeben wir das Argument `'jesse'` an die Funktion `greet_user()`, woraufhin dieser Wert dem Parameter `username` zugewiesen wird.

Hinweis

Die Begriffe Argument und Parameter werden manchmal auch synonym gebraucht. Seien Sie daher nicht überrascht, wenn die Variablen in einer Funktionsdefinition als Argumente oder die Variablen in einem Funktionsaufruf als Parameter bezeichnet werden.

Probieren Sie es selbst aus!

8-1 Nachricht: Schreiben Sie die Funktion `display_message()`, die einen Satz über das ausgibt, was Sie in diesem Kapitel lernen. Rufen Sie die Funktion auf und vergewissern Sie sich, dass die Nachricht richtig angezeigt wird.

8-2 Lieblingsbuch: Schreiben Sie die Funktion `favorite_book()` mit dem Parameter `title`. Diese Funktion soll eine Aussage wie *One of my favorite books is Alice in Wonderland* ausgeben. Rufen Sie die Funktion auf und übergeben Sie dabei einen Buchtitel als Argument.

Argumente übergeben

Eine Funktionsdefinition kann mehrere Parameter enthalten, und daher kann auch ein Funktionsaufruf mehrere Argumente benötigen. Um diese Argumente zu übergeben, gibt es verschiedene Methoden: Sie können *positionsabhängige Argumente* verwenden, die in einer bestimmten Reihenfolge stehen müssen, *Schlüsselwortargumente*, die aus einem Variablennamen und einem Wert bestehen, sowie Listen und Dictionaries mit Werten. Diese Vorgehensweisen sehen wir uns im Folgenden genauer an.

Positionsabhängige Argumente

Wenn Sie eine Funktion aufrufen, muss Python die übergebenen Argumente den Parametern in der Funktionsdefinition zuordnen. Die einfachste Möglichkeit dazu besteht darin, die Argumente in eine bestimmte Reihenfolge zu stellen. Bei dieser Vorgehensweise sprechen wir von *positionsabhängigen Argumenten*.

Betrachten wir zur Veranschaulichung eine Funktion, die verschiedene Informationen über Haustiere anzeigt, und zwar die Art des Tieres und seinen Namen:

pets.py

```
❶ def describe_pet(animal_type, pet_name):
      """Display information about a pet."""
      print(f"\nI have a {animal_type}.")
      print(f"My {animal_type}'s name is {pet_name.title()}.")

❷ describe_pet('hamster', 'harry')
```

Die Definition bei ❶ zeigt, dass die Funktion eine Tierart und den Namen des Tieres erwartet. Wenn wir `describe_pet()` aufrufen, müssen wir diese Argumente in der angegebenen Reihenfolge übergeben. In dem Funktionsaufruf bei ❷ wird das Argument `'hamster'` daher dem Parameter `animal_type` zugewiesen und `'harry'` dem Parameter `pet_name`. Diese beiden Parameter werden dann im Rumpf der

Funktion dazu genutzt, die Informationen über das Tier auszugeben, also einen Hamster namens Harry zu beschreiben:

```
I have a hamster.
My hamster's name is Harry.
```

Mehrere Funktionsaufrufe

Eine Funktion können Sie so oft aufrufen, wie es erforderlich ist. Um ein weiteres Tier zu beschreiben, brauchen Sie nur einen zweiten Aufruf von `describe_pet()`:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

In diesem zweiten Funktionsaufruf übergeben wir `describe_pet()` die Argumente `'dog'` und `'willie'`. Wie im vorherigen Beispiel ordnet Python `'dog'` dem Parameter `animal_type` und `'willie'` dem Parameter `pet_name` zu. Die Funktion gibt daraufhin nach dem Hamster Harry eine Aussage über den Hund Willie aus:

```
I have a hamster.
My hamster's name is Harry.

I have a dog.
My dog's name is Willie.
```

Der mehrfache Aufruf einer Funktion stellt eine effiziente Arbeitsweise dar, denn der Code zur Beschreibung eines Haustieres muss nur ein einziges Mal in der Funktion geschrieben werden. Immer wenn Sie dann ein Haustier beschreiben müssen, können Sie einfach die Funktion mit den Angaben über das jeweilige Tier aufrufen. Selbst wenn der Beschreibungscode zehn Zeilen umfasst, brauchen Sie für ein neues Tier immer nur eine einzige Zeile mit dem Funktionsaufruf.

In Funktionen können Sie so viele positionsabhängige Argumente verwenden, wie Sie brauchen. Python arbeitet die beim Funktionsaufruf übergebenen Argumente ab und ordnet sie den entsprechenden Parametern in der Funktionsdefinition zu.

Es kommt auf die Reihenfolge an

Wenn Sie positionsabhängige Argumente verwenden und ihre Reihenfolge bei einem Funktionsaufruf durcheinanderbringen, können Sie unerwartete Ergebnisse erhalten:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('harry', 'hamster')
```

In diesem Funktionsaufruf geben wir erst den Namen und dann die Tierart an. Da `'harry'` an erster Stelle steht, wird dieses Argument dem ersten Parameter zugewiesen, nämlich `animal_type`, und `'hamster'` dem Parameter `pet_name`. Dadurch erhalten wir einen »Harry« namens Hamster:

```
I have a harry.
My harry's name is Hamster.
```

Sollten Sie komische Ergebnisse wie dieses hier bekommen, vergewissern Sie sich, dass die Reihenfolge der Argumente im Funktionsaufruf mit der Reihenfolge der Parameter in der Funktionsdefinition übereinstimmt.

Schlüsselwortargumente

Ein *Schlüsselwortargument* ist ein Name-Wert-Paar, das Sie einer Funktion übergeben. Dabei verknüpfen Sie den Argumentwert unmittelbar mit dem Namen des Parameters, sodass es keine Verwechslungen (also keinen Harry namens Hamster) geben kann. Bei Schlüsselwortargumenten müssen Sie sich keine Gedanken über ihre Reihenfolge im Funktionsaufruf machen, denn sie geben deutlich an, welche Rolle die einzelnen Werte spielen.

Mit Schlüsselwortargumenten sieht unser Aufruf der Funktion `describe_pet()` wie folgt aus:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(animal_type='hamster', pet_name='harry')
```

Die Funktion `describe_pet()` ist unverändert geblieben, aber beim Funktionsaufruf teilen wir Python ausdrücklich mit, zu welchem Parameter die einzelnen Argu-

mente jeweils gehören. Dadurch weiß Python, dass das Argument `'hamster'` zu dem Parameter `animal_type` gehört und `'harry'` zu `pet_name`. In der Ausgabe haben wir jetzt auch tatsächlich einen Hamster namens Harry.

Da Python weiß, was die einzelnen Werte bedeuten, spielt die Reihenfolge bei Schlüsselwortargumenten keine Rolle. Die folgenden Funktionsaufrufe sind gleichwertig:

```
describe_pet(animal_type='hamster', pet_name='harry')
describe_pet(pet_name='harry', animal_type='hamster')
```

Hinweis

Bei der Verwendung von Schlüsselwortargumenten müssen Sie darauf achten, die Namen der Parameter genau so zu schreiben wie in der Funktionsdefinition.

Standardwerte

Wenn Sie eine Funktion schreiben, können Sie für jeden Parameter einen *Standardwert* (oder *Vorgabewert*) festlegen. Wird später im Funktionsaufruf ein Argument für diesen Parameter übergeben, verwendet Python den Argumentwert, anderenfalls den Standardwert. Ist für einen Parameter ein Standardwert definiert, können Sie also das zugehörige Argument im Funktionsaufruf weglassen. Das vereinfacht nicht nur Funktionsaufrufe, sondern macht auch deutlich, wie die Funktion üblicherweise benutzt werden sollte.

Nehmen wir an, Sie haben festgestellt, dass die meisten Aufrufe von `describe_pet()` zur Beschreibung von Hunden verwendet werden, und entscheiden sich daher, `'dog'` als Standardwert für `animal_type` vorzugeben. Wer `describe_pet()` für einen Hund aufrufen möchte, kann die entsprechende Angabe nun einfach weglassen:

```
def describe_pet(pet_name, animal_type='dog'):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(pet_name='willie')
```

Hier haben wir in die Definition von `describe_pet()` den Standardwert `'dog'` für `animal_type` aufgenommen. Wird die Funktion jetzt ohne Angabe von `animal_type` aufgerufen, verwendet Python den Wert `'dog'` für diesen Parameter:

```
I have a dog.
My dog's name is Willie.
```

Beachten Sie, dass wir außerdem die Reihenfolge der Parameter in der Funktionsdefinition ändern mussten. Da wir dank des Standardwertes kein Argument für die Tierart mehr angeben müssen, ist im Funktionsaufruf nur noch der Name als Argument übrig. Für Python ist dies aber nach wie vor ein positionsabhängiges Argument. Wenn wir im Aufruf also nur den Tiernamen angeben, wird dieses Argument mit dem ersten Parameter in der Definition verknüpft. Daher muss `pet_name` der erste Parameter sein.

Die einfachste Möglichkeit zur Verwendung der Funktion, besteht jetzt darin, im Funktionsaufruf lediglich den Namen des Hundes anzugeben:

```
describe_pet('willie')
```

Das führt zu derselben Ausgabe wie im vorherigen Beispiel. Hier ist `'willie'` das einzige Argument und wird daher dem ersten Parameter in der Definition zugeordnet, also `pet_name`. Da für `animal_type` kein Argument angegeben wird, greift Python auf den Standardwert `'dog'` zurück.

Um ein anderes Tier zu beschreiben, müssen Sie den Funktionsaufruf wie folgt schreiben:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Da jetzt ausdrücklich ein Argument für `animal_type` bereitgestellt wird, ignoriert Python den Standardwert für diesen Parameter.

Hinweis

Parameter mit Standardwerten müssen in der Definition hinter den Parametern ohne Standardwerte aufgeführt werden, damit Python positionsabhängige Argumente korrekt zuordnen kann.

Verschiedene Formen für Funktionsaufrufe

Da positionsabhängige Argumente, Schlüsselwortargumente und Vorgabewerte kombiniert verwendet werden können, gibt es mehrere Möglichkeiten, um eine Funktion aufzurufen. Betrachten Sie die folgende Definition von `describe_pet()` mit einem Standardwert:

```
def describe_pet(pet_name, animal_type='dog'):
```

Für `pet_name` muss immer ein Argument angegeben werden, und das kann ein positionsabhängiges oder ein Schlüsselwortargument sein. Wenn das zu beschreibende Tier kein Hund ist, muss auch ein Argument für `animal_type` übergeben

werden, und auch dafür können Sie sowohl ein positionsabhängiges als auch ein Schlüsselwortargument verwenden.

Alle folgenden Aufrufe sind für diese Funktion geeignet:

```
# Ein Hund namens Willie.
describe_pet('willie')
describe_pet(pet_name='willie')

# Ein Hamster namens Harry.
describe_pet('harry', 'hamster')
describe_pet(pet_name='harry', animal_type='hamster')
describe_pet(animal_type='hamster', pet_name='harry')
```

Alle Funktionsaufrufe für dasselbe Tier führen jeweils zu derselben Ausgabe.

Hinweis

Für welche Form von Aufruf Sie sich entscheiden, spielt wirklich keine Rolle. Solange Sie die gewünschte Ausgabe erhalten, können Sie einfach den Stil verwenden, mit dem Sie am besten zurechtkommen.

Argumentfehler vermeiden

Wenn Sie beim Aufruf einer Funktion versehentlich weniger oder mehr Argumente angeben, als die Funktion benötigt, erhalten Sie eine entsprechende Fehlermeldung. Das folgende Beispiel zeigt, was passiert, wenn Sie versuchen, `describe_pet()` ohne Argumente aufzurufen:

```
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"\nI have a {animal_type}.")
    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet()
```

Python bemerkt, dass in dem Aufruf eine Information fehlt, und teilt uns das im Traceback mit:

```
Traceback (most recent call last):
❶  File "pets.py", line 6, in <module>
❷    describe_pet()
❸ TypeError: describe_pet() missing 2 required positional arguments: 'animal_
type' and 'pet_name'
```

Bei ❶ im Traceback erfahren wir, wo sich das Problem befindet, sodass wir an der entsprechenden Stelle nachsehen können, warum unser Funktionsaufruf nicht

funktioniert. Der fehlerhafte Funktionsaufruf wird bei ❷ angezeigt. In der Zeile bei ❸ ist angegeben, welche zwei Argumente in dem Funktionsaufruf fehlen. Wenn sich die Funktion in einer getrennten Datei befindet, können wir dank dieser Angaben den Aufruf direkt korrigieren, ohne die Datei zu öffnen und den Funktionscode zu lesen.

Python unterstützt uns, da es den Funktionscode liest und uns die Namen der anzugebenden Argumente mitteilt. Das ist ein weiterer Grund dafür, Variablen und Funktionen beschreibende Namen zu geben, denn dadurch werden die Fehlermeldungen von Python viel aussagekräftiger – sowohl für Sie selbst als auch für andere Personen, die Ihren Code nutzen.

Wenn Sie zu viele Argumente angeben, erhalten Sie ein ähnliches Traceback, das Ihnen helfen kann, den Funktionsaufruf zu korrigieren, sodass er zu der Funktionsdefinition passt.

Probieren Sie es selbst aus!

8-3 T-Shirt: Schreiben Sie die Funktion `make_shirt()`, die eine Kleidergröße und den Text des auf dem T-Shirt abzudruckenden Spruchs entgegennimmt und einen Satz mit einer Zusammenfassung der Bestellung ausgibt.

Rufen Sie die Funktion einmal mit positionsabhängigen und einmal mit Schlüsselwortargumenten auf.

8-4 Große T-Shirts: Ändern Sie die Funktion `make_shirt()` so ab, dass standardmäßig T-Shirts der Größe L mit dem Text *I love Python* hergestellt werden. Bestellen Sie ein T-Shirt der Größe L und eines der Größe M jeweils mit dem Standardspruch sowie ein T-Shirt beliebiger Größe mit einem anderen Text.

8-5 Städte: Schreiben Sie die Funktion `describe_city()`, die den Namen einer Stadt und des zugehörigen Landes entgegennimmt und einen einfachen Satz wie *Reykjavik is in Iceland* ausgibt. Sehen Sie für den Landesparameter einen Standardwert vor. Rufen Sie die Funktion für drei verschiedene Städte auf, darunter mindestens einmal für eine Stadt, die sich nicht in dem vorgegebenen Land befindet.

Rückgabewerte

Eine Funktion muss ihre Ausgabe nicht direkt anzeigen. Sie kann auch Daten verarbeiten und einen Wert oder eine Menge von Werten zurückgeben. Dabei sprechen wir von einem sogenannten *Rückgabewert*. Die Anweisung `return` sendet einen Wert, der innerhalb der Funktion ermittelt wurde, in die Zeile zurück, in der die Funktion aufgerufen wurde. Mithilfe von Rückgabewerten können Sie einen Großteil der Routinearbeit Ihres Programms in Funktionen auslagern. Das vereinfacht den Hauptteil des Programms.

Einen einfachen Wert zurückgeben

Die folgende Funktion nimmt einen Vor- und Nachnamen entgegen und gibt einen sauber formatierten vollständigen Namen zurück:

formatted_name.py

```
❶ def get_formatted_name(first_name, last_name):
      """Return a full name, neatly formatted."""
❷     full_name = f"{first_name} {last_name}"
❸     return full_name.title()

❹ musician = get_formatted_name('jimi', 'hendrix')
  print(musician)
```

In der Definition von `get_formatted_name()` sind ein Vor- und ein Nachname als Parameter vorgesehen (❶). Die Funktion kombiniert die beiden Namen mit einem Leerzeichen dazwischen und weist das Ergebnis `full_name` zu (❷). Der Wert in dieser Variablen wird dann bei ❸ in das Format mit großen Anfangsbuchstaben umgewandelt und an die aufrufende Zeile zurückgegeben.

Wenn Sie eine Funktion aufrufen, die einen Rückgabewert hat, müssen Sie eine Variable angeben, der dieser Rückgabewert zugewiesen werden kann. In diesem Fall verwenden wir dafür `musician` (❹). Die Ausgabe zeigt den formatierten, aus Vor- und Nachname zusammengesetzten Gesamtnamen an:

```
Jimi Hendrix
```

Das scheint übermäßig viel Arbeit für ein solches Ergebnis zu sein, das wir auch mit folgender einfacher Zeile erhalten könnten:

```
print("Jimi Hendrix")
```

Stellen Sie sich aber ein umfangreiches Programm vor, in dem viele Vor- und Nachnamen getrennt gespeichert werden. In einem solchen Fall ist eine Funktion wie `get_formatted_name()` sehr praktisch. Sie müssen dann immer nur diese Funktion aufrufen, wenn Sie den vollständigen Namen anzeigen lassen möchten.

Optionale Argumente

Manchmal ist es sinnvoll, einzelne Argumente nur als eine Option vorzusehen, sodass die entsprechenden Informationen nur dann angegeben werden müssen, wenn es nötig ist. Um das zu erreichen, können Sie Standardwerte verwenden.

Nehmen wir an, Sie möchten die Funktion `get_formatted_name()` so erweitern, dass sie auch mit einem zweiten Vornamen umgehen kann. Das könnten Sie durchaus wie folgt schreiben:

```
def get_formatted_name(first_name, middle_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {middle_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')
print(musician)
```

Wenn Sie dieser Funktion einen Vornamen, einen zweiten Vornamen und einen Nachnamen übergeben, baut sie alle drei zu einem String zusammen, fügt an den Trennstellen Leerzeichen ein und sorgt dafür, dass alle Bestandteile mit einem Großbuchstaben beginnen:

```
John Lee Hooker
```

Aber ein zweiter Vorname ist nicht immer erforderlich, und wenn Sie versuchen, die Funktion nur mit einem Vor- und einem Nachnamen aufzurufen, erhalten Sie eine Fehlermeldung. Um die Angabe des zweiten Vornamens optional zu machen, geben wir dem Argument `middle_name` einen leeren Standardwert und ignorieren es, sofern der Benutzer keinen ausdrücklichen Wert angibt. Außerdem müssen wir den Parameter `middle_name` ans Ende der Liste setzen:

```
❶ def get_formatted_name(first_name, last_name, middle_name=''):
      """Return a full name, neatly formatted."""
❷     if middle_name:
          full_name = f"{first_name} {middle_name} {last_name}"
❸     else:
          full_name = f"{first_name} {last_name}"
      return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')
print(musician)

❹ musician = get_formatted_name('john', 'hooker', 'lee')
print(musician)
```

Hier wird der Name aus drei möglichen Bestandteilen zusammengebaut. Da es immer einen Vor- und einen Nachnamen gibt, stehen diese Parameter in der Funktionsdefinition an erster Stelle. Der zweite Vorname dagegen ist optional. Daher wird er mit einem leeren String als Standardwert am Ende der Liste aufgeführt (❶).

Im Rumpf der Funktion prüfen wir, ob ein zweiter Vorname angegeben wurde. Da Python nicht leere Strings als `True` interpretiert, wird die Bedingung zu `True` ausgewertet, wenn in dem Funktionsaufruf ein Argument für `middle_name` angegeben wurde (❷). In diesem Fall werden alle drei Bestandteile zum Gesamtnamen

zusammengebaut. Dieser Name wird dann mit großen Anfangsbuchstaben versehen, an die aufrufende Zeile zurückgegeben, der Variablen `musician` zugewiesen und ausgegeben. Wurde dagegen kein zweiter Vorname angegeben, so wird der `if`-Test zu `False` ausgewertet und der `else`-Block ausgeführt (❸). In diesem Fall wird der Gesamtname nur aus Vor- und Nachname zusammengebaut, zurückgegeben, `musician` zugewiesen und angezeigt.

Der Aufruf dieser Funktion nur mit einem Vor- und Nachnamen ist ganz einfach. Wenn Sie jedoch auch einen zweiten Vornamen angeben, müssen Sie darauf achten, dass Sie dieses Argument als letztes übergeben, damit Python die positionsabhängigen Argumente korrekt zuordnet (❹).

Diese geänderte Version unserer Funktion ist für Personen mit und ohne zweiten Vornamen geeignet:

```
Jimi Hendrix
John Lee Hooker
```

Mithilfe von optionalen Argumenten können Funktionen eine große Menge an verschiedenen Fällen abdecken, wobei die Funktionsaufrufe so einfach bleiben wie möglich.

Ein Dictionary zurückgeben

Funktionen können beliebige Arten von Werten zurückgeben, auch kompliziertere Datenstrukturen wie Listen oder Dictionaries. Die folgende Funktion nimmt die Bestandteile eines Namens entgegen und gibt ein Dictionary zurück, das für eine Person steht:

person.py

```
    def build_person(first_name, last_name):
        """Return a dictionary of information about a person."""
❶       person = {'first': first_name, 'last': last_name}
❷       return person

    musician = build_person('jimi', 'hendrix')
❸   print(musician)
```

Die Funktion `build_person()` nimmt einen Vor- und einen Nachnamen entgegen und platziert diese Werte bei ❶ in einem Dictionary. Dabei wird der Wert von `first_name` unter dem Schlüssel `first` gespeichert und der von `last_name` unter dem Schlüssel `last`. Bei ❷ geben wir das komplette Dictionary für die Person zurück. Der Rückgabewert wird bei ❸ ausgegeben, wobei die ursprünglichen Informationen jetzt in einem Dictionary gespeichert sind:

```
{'first': 'jimi', 'last': 'hendrix'}
```

Diese Funktion nimmt zwei einfache Informationen in Textform entgegen und stellt sie in eine aussagekräftigere Datenstruktur, mit der Sie sie auf anspruchsvollere Weise verarbeiten können, als sie lediglich auszugeben. Die Strings `'jimi'` und `'hendrix'` sind jetzt als Vor- und Nachname gekennzeichnet. Diese Funktion können Sie leicht erweitern, sodass sie auch noch optionale Werte wie einen zweiten Vornamen, das Alter, den Beruf oder sonstige Angaben über die Person entgegennimmt, die Sie speichern möchten. Mit folgender Änderung können Sie beispielsweise das Alter festhalten:

```
def build_person(first_name, last_name, age=None):
    """Return a dictionary of information about a person."""
    person = {'first': first_name, 'last': last_name}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

Hier haben wir der Funktionsdefinition den optionalen Parameter `age` hinzugefügt und ihm den besonderen Wert `None` zugewiesen, der verwendet wird, wenn eine Variable keinen Wert erhalten hat. Sie können sich `None` als einen Platzhalter vorstellen. Beim Überprüfen von Bedingungen wird `None` zu `False` ausgewertet. Wird im Funktionsaufruf ein Wert für `age` angegeben, so wird er in dem Dictionary gespeichert. Die Funktion speichert immer den Namen der Person, kann aber so erweitert werden, dass sie auch noch andere Informationen über die Person festhält.

Funktionen in einer while-Schleife

Funktionen können Sie in allen Python-Strukturen verwenden, die Sie bis jetzt kennengelernt haben. Beispielsweise können Sie die Funktion `get_formatted_name()` in einer `while`-Schleife einsetzen, um Benutzer förmlicher zu begrüßen. Eine erste Version könnte wie folgt aussehen:

greeter.py

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

# Dies ist eine Endlosschleife!
while True:
    print("\nPlease tell me your name:")
    f_name = input("First name: ")
    l_name = input("Last name: ")
```

❶

```
    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

Hier verwenden wir die einfache Version von `get_formatted_name()` ohne zweiten Vornamen. Die `while`-Schleife bittet den Benutzer, seinen Namen einzugeben, wobei wir getrennt nach Vor- und Nachnamen fragen (❶).

Es gibt jedoch ein Problem bei dieser `while`-Schleife: Wir haben keine Beendigungsbedingung definiert. Wo bringen Sie eine solche Bedingung unter, wenn Sie nach einer Folge von Eingaben fragen? Die Benutzer sollen das Programm so einfach wie möglich abbrechen können, weshalb jede Eingabeaufforderung auch eine Möglichkeit dazu angeben sollte. Mit der Anweisung `break` können wir die Schleife bei jeder Eingabeaufforderung ganz einfach abbrechen:

```
def get_formatted_name(first_name, last_name):
    """Return a full name, neatly formatted."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

while True:
    print("\nPlease tell me your name:")
    print("(enter 'q' at any time to quit)")

    f_name = input("First name: ")
    if f_name == 'q':
        break

    l_name = input("Last name: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\nHello, {formatted_name}!")
```

Wir fügen eine Meldung hinzu, die den Benutzern mitteilt, wie sie das Programm beenden können, und brechen die Schleife ab, wenn an einer der Eingabeaufforderungen der Beendigungswert eingegeben wird. Jetzt fährt das Programm mit der Begrüßung der Benutzer so lange fort, bis jemand statt eines Namens den Wert `'q'` eingibt:

```
Please tell me your name:
(enter 'q' at any time to quit)
First name: eric
Last name: matthes

Hello, Eric Matthes!
```