4. Mehr Werkzeuge zur Ablaufsteuerung

Neben der `while`-Anweisung, die gerade vorgestellt wurde, kennt Python – abgesehen von wenigen Abweichungen — die üblichen Anweisungen zur Ablaufsteuerung, die von anderen Sprachen bekannt sind.

4.1. `if`-Anweisungen

Ein Beispiel zur `if`-Anweisung

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...      x = 0
...      print('Negative changed to zero')
... elif x == 0:
...      print('Zero')
... elif x == 1:
...      print('Single')
... else:
...      print('More')
...
More
```

`else`-Zweig oder `elif`-Zweige sind optional. Im Unterschied zum `else`-Zweig, der nur einmal vorkommen kann, ist eine Abfolge von mehreren `elif`-Zweigen möglich; dadurch lassen sich verschachtelte Einrückungen vermeiden. Eine Abfolge von `if` ... `elif` ... `elif`-Zweigen ersetzt die `switch`- oder `case`-Konstrukte anderer Programmiersprachen.

4.2. `for`-Anweisungen

Die `for`-Anweisung in Python unterscheidet sich ein wenig von der, die man von C oder Pascal her kennt. Man kann nicht nur über eine Zahlenfolge iterieren (wie in Pascal) oder lediglich Schrittweite und Abbruchbedingung festlegen (wie in C), sondern über eine beliebige Sequenz (also z. B. eine Liste oder Zeichenkette), und zwar in der Reihenfolge, in der die Elemente in der Sequenz vorkommen. Zum Beispiel:

```
>>> # Die Längen einiger Zeichenketten ermitteln:
... a = ['Katze', 'Fenster', 'rauswerfen']
>>> for x in a:
...     print(x, len(x))
...
Katze 5
Fenster 7
rauswerfen 10
```

Vorsicht ist geboten, wenn man versucht, Veränderungen an einer Sequenz vorzunehmen, über die gerade iteriert wird (was natürlich nur bei veränderbaren Sequenztypen, wie etwa Listen, passieren kann). Will man eine Liste verändern, über die man iteriert, um beispielsweise ausgewählte Elemente zu duplizieren, muss man über eine Kopie iterieren. Die Slice-Notation macht dies sehr einfach:

```
>>> for x in a[:]: # benutze eine Kopie der gesamten Liste
...     if len(x) > 7: a.insert(0, x)
...
>>> a
['rauswerfen', 'Katze', 'Fenster', 'rauswerfen']
```

4.3. Die Funktion `range()`

Wenn man wirklich über eine Zahlenfolge iterieren muss, bietet sich die eingebaute Funktion `range()` an, die arithmetische Folgen erzeugt.

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Wird nur ein Argument angegeben, so beginnt der erzeugte Bereich bei Null und endet mit dem um 1 kleineren Wert des angegebenen Arguments. `range(10)` erzeugt die Zahlen von 0 bis einschließlich 9. Das entspricht den gültigen Indizes einer Sequenz mit zehn Elementen. Es ist ebenfalls möglich, den Bereich mit einem anderen Wert als Null zu beginnen oder auch eine bestimmte Schrittweite (step) festzulegen — sogar negative Schrittweiten sind möglich.

```
range(5, 10)
   5 bis 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

Will man über die Indizes einer Sequenz iterieren, kann man `range()` und `len()` wie folgt kombinieren:

```
>>> a = ['Mary', 'hatte', 'ein', 'kleines', 'Lamm']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 hatte
2 ein
3 kleines
4 Lamm
```

Eleganter ist es jedoch, in solchen Fällen die Funktion `enumerate()` zu benutzen, siehe Schleifentechniken.

Etwas Seltsames passiert, wenn man einfach ein range-Objekt ausgeben will:

```
>>> print(range(10))
range(0, 10)
```

Zwar verhält sich das von `range()` zurückgegebene Objekt in etwa wie eine Liste, es ist jedoch in Wahrheit keine Liste. `range()` liefert ein Objekt zurück, das der Reihe nach die einzelnen Zahlen der Folge zurückliefert, die durch die an `range()` übergebenen Argumente festgelegt wurde. Dadurch lässt sich gegenüber der Erzeugung einer Liste Speicherplatz sparen.

Wir nennen solch ein Objekt Iterable, und es kann überall da eingesetzt werden, wo ein Objekt erwartet wird, das eine Folge von Elementen der Reihe nach "produziert", bis sein Vorrat erschöpft ist. Beispielsweise fungiert die `for`-Anweisung als ein solcher Iterator. Auch die Funktion `list()` ist ein solcher Iterator, die als Argument ein Iterable erwartet und eine Liste daraus macht

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Später werden noch weitere Funktionen behandelt, die Iterables zurückgeben und Iterables als Argument aufnehmen.

4.4. `break`- und `continue`-Anweisungen und der `else`-Zweig bei Schleifen

Eine `break`-Anweisung in einem Schleifenrumpf bewirkt — wie in C — dass an dieser Stelle mit sofortiger Wirkung die sie unmittelbar umgebende Schleife verlassen wird.

Auch Schleifen-Anweisungen können einen `else`-Zweig haben. Dieser wird genau dann ausgeführt, wenn die Schleife nicht durch eine `break`-Anweisung abgebrochen wurde. Das folgende Beispiel zur Berechnung von Primzahlen veranschaulicht das.

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # Schleife wurde durchlaufen, ohne dass ein Faktor gefunden wurde
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Ja, dieser Code ist korrekt. Schau genau hin: Die `else` Klausel gehört zur `for`-Schleife, **nicht** zur `if`-Anweisung.)

Wenn die `else` Klausel bei einer Schleife benutzt wird, hat sie mehr mit dem `else` einer `try` Anweisung gemein, als mit dem der `if` Anweisung: Die `else` Klausel eines `try` wird ausgeführt, wenn keine Ausnahme auftritt und die einer Schleife, wenn kein `break` ausgeführt wird. Mehr Informationen zur `try` Anweisung und Ausnahmen gibt es unter Ausnahmen behandeln.

Entsprechend bewirkt die `continue`-Anweisung — ebenso von C entliehen — , dass sofort mit der nächsten Iteration der Schleife fortgefahren wird:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

4.5. `pass`-Anweisungen

Die `pass`-Anweisung tut nichts. Sie wird eingesetzt, wenn syntaktisch eine Anweisung benötigt wird, das Programm jedoch nichts tun soll. Ein Beispiel:

```
>>> while True:
...     pass  # geschäftiges Warten auf den Tastatur-Interrupt (Strg+C)
...
```

Auch bei der Erzeugung einer minimalen Klasse kann `pass` zum Einsatz kommen:

```
>>> class MyEmptyClass:
...     pass
...
```

`pass` lässt sich auch sinnvoll einsetzen als Platzhalter für den Rumpf einer Funktionen oder Schleife bei der “Top-Down”-Programmierung, um so zunächst auf einer abstrakteren Ebene zu denken

```
>>> def initlog(*args):
...     pass   # Implementieren nicht vergessen!
...
```

4.6. Funktionen definieren

Im folgenden Beispiel wird eine Funktion definiert, die die Fibonacci-Folge bis zu einer beliebigen Grenze ausgibt:

```
>>> def fib(n):    # die Fibonacci-Folge bis n ausgeben
...     """Print the Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
```

```
...     print()
...
>>> # Jetzt rufen wir die Funktion auf, die wir gerade definiert haben:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Das Schlüsselwort `def` leitet die Definition einer Funktion ein. Darauf folgt der Funktionsname und eine Auflistung der formalen Parameter, die allerdings auch leer sein kann. Die Anweisungen, die den Funktionskörper bilden, beginnen in der nächsten Zeile und müssen eingerückt sein.

Die erste Anweisung des Funktionskörpers kann auch ein Zeichenkettenliteral sein, ein so genannter Dokumentationsstring der Funktion, auch Docstring genannt. (Mehr zu Docstrings kann im Abschnitt Dokumentationsstrings nachgelesen werden.) Es gibt Werkzeuge, die Docstrings verwenden, um automatisch Online-Dokumentation oder gedruckte Dokumentation zu erzeugen oder es dem Anwender ermöglichen, interaktiv den Code zu durchsuchen. Die Verwendung von Docstrings ist eine gute Konvention, an die man sich bei der Programmierung nach Möglichkeit halten sollte.

Beim Aufruf einer Funktion kommt es zur Bildung eines lokalen Namensraums, der sich auf alle Bezeichner erstreckt, die im Funktionsrumpf (durch Zuweisung oder als Elemente der Parameterliste) neu definiert werden. Diese Bezeichner werden mit den ihnen zugeordneten Objekten in einer lokalen Symboltabelle abgelegt.

Wenn im Funktionsrumpf ein Bezeichner vorkommt, wird der Name zunächst in der lokalen Symboltabelle gesucht, danach in den lokalen Symboltabellen der umgebenden Funktionen, dann in der globalen Symboltabelle und schließlich in der Symboltabelle der eingebauten Namen. Darum ist es ohne weiteres nicht möglich, einer globalen Variablen innerhalb des lokalen Namensraums einer Funktion einen Wert zuzuweisen. Dadurch würde stattdessen eine neue, namensgleiche lokale Variable definiert, die die namensgleiche globale Variable überdeckt und dadurch auch den lesenden Zugriff auf diese globale Variable verhindert. Ein lesender Zugriff auf globale Variablen ist ansonsten immer möglich, ein schreibender Zugriff nur unter Verwendung der `global`-Anweisung.

Die konkreten Parameter (Argumente), die beim Funktionsaufruf übergeben werden, werden den formalen Parametern der Parameterliste zugeordnet und gehören damit zur lokalen Symboltabelle der Funktion. Das heißt, Argumente werden über call by value übergeben (wobei der Wert allerdings immer eine Referenz auf ein Objekt ist, nicht der Wert des Objektes selbst) [1]. Wenn eine Funktion eine andere Funktion aufruft, wird eine neue lokale Symboltabelle für diesen Aufruf erzeugt.

Eine Funktionsdefinition fügt den Funktionsnamen in die lokale Symboltabelle ein. Der Wert des Funktionsnamens hat einen Typ, der vom Interpreter als benutzerdefinierte

Funktion erkannt wird. Dieser Wert kann einem anderen Namen zugewiesen werden, der dann ebenfalls als Funktion genutzt werden kann und so als Möglichkeit zur Umbenennung dient.

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Wer Erfahrung mit anderen Programmiersprachen hat, wird vielleicht einwenden, dass `fib` gar keine Funktion, sondern eine Prozedur ist, da sie keinen Wert zurückgibt. Tatsächlich geben aber auch Funktionen ohne eine **`return`**-Anweisung einen Wert zurück, wenn auch einen eher langweiligen, nämlich den eingebauten Namen `None` ("nichts"). Die Ausgabe des Wertes `None` wird normalerweise vom Interpreter unterdrückt, wenn es der einzige Wert wäre, der ausgegeben wird. Möchte man ihn sehen, kann man ihn mittels **`print()`** sichtbar machen.:

```
>>> fib(0)
>>> print(fib(0))
None
```

Statt eine Abfolge von Zahlen in einer Funktion auszugeben, kann man auch eine Liste dieser Zahlen als Objekt zurückliefern.

```
>>> def fib2(n): # gibt die Fibonacci-Folge bis  n zurück
...     """Return a list containing the Fibonacci series up to n."""
...     result = list()
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # siehe unten
...         a, b = b, a + b
...     return result
...
>>> f100 = fib2(100)    # ruf es auf
>>> f100                # gib das Ergebnis aus
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Dieses Beispiel zeigt einige neue Eigenschaften von Python:

- Die **`return`**-Anweisung gibt einen Wert von einer Funktion zurück. Ohne einen Ausdruck als Argument gibt **`return`** `None` zurück; das gleiche gilt, wenn eine **`return`**-Anweisung fehlt.

- Die Anweisung `result.append(a)` ruft eine Methode des Listenobjektes in `result` auf. Eine Methode ist eine Funktion, die zu einem Objekt 'gehört' und wird mittels Punktnotation (`obj.methodname`) dargestellt. Dabei ist `obj` irgendein Objekt (es kann auch ein Ausdruck sein) und `methodname` der Name einer Methode, die vom Typ des Objektes definiert wird. Unterschiedliche Typen definieren verschiedene Methoden. Methoden verschiedener Typen können denselben Namen haben ohne doppeldeutig zu sein. (Es ist auch möglich, eigene Objekttypen zu erstellen, indem man Klassen benutzt,

siehe Klassen.) Die Methode `append()`, die im Beispiel gezeigt wird, ist für Listenobjekte definiert. Sie hängt ein neues Element an das Ende der Liste an. Im Beispiel ist es äquivalent zu `result = result + [a]`, aber effizienter.

4.7. Mehr zum Definieren von Funktion

Funktionen lassen sich auch mit einer variablen Anzahl von Argumenten definieren. Dabei sind drei Varianten zu unterscheiden, die auch kombiniert werden können.

4.7.1. Standardwerte für Argumente

Die nützlichste Variante ist, einen Standardwert für ein oder mehrere Argumente anzugeben. Das erzeugt eine Funktion, die mit weniger Argumenten aufgerufen werden kann, als sie definitionsgemäß erlaubt. Zum Beispiel:

```
def ask_ok(prompt, retries=4, complaint='Bitte Ja oder Nein!'):
    while True:
        ok = input(prompt)
        if ok in ('j', 'J', 'ja', 'Ja'): return True
        if ok in ('n', 'N', 'ne', 'Ne', 'Nein'): return False
        retries = retries - 1
        if retries < 0:
            raise IOError('Benutzer abgelehnt!')
        print(complaint)
```

Diese Funktion könnte auf mehrere Arten aufgerufen werden:

- Indem man nur das vorgeschriebene Argument übergibt: `ask_ok("Willst du wirklich aufhören?")`
- Indem man zusätzlich ein optionales Argument übergibt: `ask_ok("Willst du die Datei überschreiben?", 2)`
- Oder indem man sogar alle übergibt: `ask_ok("Willst du die Datei überschreiben?", 2, "Komm schon, nur Ja oder Nein")`

Das Beispiel führt auch noch das Schlüsselwort `in` ein. Dieses überprüft ob ein gegebener Wert in einer Sequenz gegeben ist.

Die Standardwerte werden zum Zeitpunkt der Funktionsdefinition im definierenden Gültigkeitsbereich ausgewertet, so dass:

```
i = 5

def f(arg=i):
    print(arg)
```

```
i = 6
f()
```

`5` ausgeben wird.

Wichtige Warnung: Der Standardwert wird nur einmal ausgewertet. Das macht einen Unterschied, wenn der Standardwert veränderbares Objekt, wie beispielsweise eine Liste, ein Dictionary oder Instanzen der meisten Klassen, ist. Zum Beispiel häuft die folgende Funktion alle Argumente an, die ihr in aufeinanderfolgenden Aufrufen übergeben wurden:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

Und sie gibt folgendes aus:

```
[1]
[1, 2]
[1, 2, 3]
```

Wenn man nicht will, dass der Standardwert von aufeinanderfolgenden Aufrufen gemeinsam benutzt wird, kann man die Funktion folgendermaßen umschreiben:

```
def f(a, L=None):
     if L is None:
          L = []
     L.append(a)
     return L
```

4.7.2. Schlüsselwortargumente

Funktionen können auch mit Schlüsselwortargumenten in der Form `Schlüsselwort=Wert` aufgerufen werden. Zum Beispiel die folgende Funktion:

```
def parrot(voltage, state='völlig steif', action='fliegen', type='norwegische
Blauling'):
     print("-- Der Vogel würde selbst dann nicht", action, end=' ')
     print("selbst wenn Sie ihm ", voltage, "Volt durch den Schnabel jagen täten")
     print("-- Ganz erstaunlicher Vogel, der", type, "! Wunderhübsche Federn!")
     print("-- Er is", state, "!")
```

mindestens ein Argument (`voltage`) akzeptiert drei optionale Argumente (`state` , `action` und `type`) und kann mit allen folgenden Varianten aufgerufen werden:

```
parrot(4000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('Viertausend', state = 'an den Gänseblümchen riechen')
parrot('eine Million', 'keine Spur leben', 'springen')
```

die folgenden Aufrufe wären allerdings alle ungültig:

```
parrot()                     # das benötigte Argument fehlt
parrot(voltage=5.0, 'tot')   # auf ein Schlüsselwortargument folgt ein normales
parrot(110, voltage=220)     # doppelter Wert für ein Argument
parrot(actor='John Cleese')  # unbekanntes Schlüsselwort
```

Bei einem Funktionsaufruf müssen Schlüsselwortargumente nach positionsabhängigen Argumenten kommen. Alle übergebenen Schlüsselwortargumente müssen jeweils auf eines der Argumente passen, die die Funktion akzeptiert (beispielsweise ist `actor` kein gültiges Argument für die `parrot` Funktion), wobei ihre Reihenfolge aber unwichtig ist. Das gilt auch für nicht-optionale Argumente (beispielsweise ist `parrot(voltage=1000)` auch gültig). Kein Argument darf mehr als einen Wert zugewiesen bekommen. Ein Beispiel, das wegen dieser Einschränkung scheitert:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
 File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Ist ein Parameter der Form `**name` in der Definition enthalten, bekommt dieser ein Dictionary (siehe Mapping Types), das alle Schlüsselwortargumente enthält, bis auf die, die in der Definition vorkommen. Dies kann mit einem Parameter der Form `*name`, der im nächsten Unterabschnitt beschrieben wird, kombiniert werden. Dieser bekommt ein Tupel, das alle positionsabhängigen Argumente enthält, die über die Anzahl der definierten hinausgehe. (`*name` muss aber vor `**name` kommen.) Wenn wir zum Beispiel eine Funktion wie diese definieren:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Haben sie", kind, "?")
    print("-- Tut mir leid,", kind, "ist leider gerade aus.")
    for arg in arguments:
        print(arg)
    print("-" * 40)
    keys = sorted(keywords.keys())
    for kw in keys:
        print(kw, ":", keywords[kw])
```

könnte sie so aufgerufen werden:

```
cheeseshop("Limburger", "Der ist sehr flüssig, mein Herr.",
           "Der ist wirklich sehr, SEHR flüssig, mein Herr.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

und natürlich würde sie folgendes ausgeben:

```
-- Haben sie Limburger ?
-- Tut mir leid, Limburger ist leider gerade aus.
Der ist sehr flüssig, mein Herr.
Der ist wirklich sehr, SEHR flüssig, mein Herr.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Man beachte, dass die Liste der Schlüsselwortargumente erzeugt wird, indem das Ergebnis der Methode `keys()` sortiert wird, bevor dessen Inhalt ausgegeben wird. Tut man das nicht, ist die Reihenfolge der Ausgabe undefiniert.

4.7.3. Beliebig lange Argumentlisten

Die am wenigsten gebräuchliche Möglichkeit ist schließlich, festzulegen, dass eine Funktion mit einer beliebigen Zahl von Argumenten aufgerufen werden kann, die dann in ein Tupel (siehe Tupel und Sequenzen) verpackt werden. Vor diesem speziellen Argument kann eine beliebige Menge normaler Argumente vorkommen.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Normalerweise wird dieses spezielle Argument an das Ende der Argumentliste gesetzt, weil es alle verbleibenden Argumente, mit denen die Funktion aufgerufen wird, aufnimmt. Alle Argumente, die in der Definition auf ein `*args` folgen, sind nur durch Schlüsselwortargumente zu übergeben ('keyword-only') und nicht durch positionsabhängige.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("Erde", "Mars", "Venus")
'Erde/Mars/Venus'
>>> concat("Erde", "Mars", "Venus", sep=".")
'Erde.Mars.Venus'
```

4.7.4. Argumentlisten auspacken

Die umgekehrte Situation ereignet sich, wenn die Argumente schon in einer Liste oder einem Tupel stecken, aber für einen Funktionsaufruf ausgepackt werden müssen, der separate positionsabhängige Argumente erfordert. Zum Beispiel erwartet die eingebaute Funktion `range()` getrennte Argumente für Start und Stop. Wenn sie aber nicht getrennt vorhanden sind, kann man im Funktionsaufruf den `*`-Operator benutzen, um die Argumente aus einer Liste oder einem Tupel auszupacken.

```
>>> list(range(3, 6))   # normaler Aufruf mit getrennten Argumenten
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))  # Aufruf mit Argumenten, die aus einer Liste ausgepackt werden
[3, 4, 5]
```

Analog können Dictionaries Schlüsselwortargumente mit dem `**`-Operator bereitstellen:

```
>>> def parrot(voltage, state=' völlig steif',
...     action='fliegen', type='norwegische Blauling'):
...     print("-- Der Vogel würde selbst dann nicht", action, end=' ')
...     print("selbst wenn Sie ihm ", voltage, "Volt durch den Schnabel jagen täten.")
...     print("-- Er is", state, "!")

>>> d = {"voltage": "vier Millionen", "state": "verdammt nochmal tot!", "action":
"FLIEGEN"}
>>> parrot(**d)
```

```
-- Der Vogel würde selbst dann nicht FLIEGEN selbst wenn sie ihm vier Millionen Volt
durch den Schnabel jagen täten.
-- Er is verdammt nochmal tot!
```

4.7.5. Lambda-Form - anonyme Funktion

Aufgrund der hohen Nachfrage, haben ein paar Merkmale, die in funktionalen Programmiersprachen wie Lisp üblich sind, Einzug in Python gehalten. Mit dem Schlüsselwort `lambda` können kleine anonyme Funktionen erstellt werden. Hier eine Funktion, die die Summe seiner zwei Argumente zurückgibt: `lambda a, b: a + b`. `lambda` kann überall genutzt werden, wo ein Funktionsobjekt benötigt wird. Semantisch ist es nur syntaktischer Zucker für eine normale Funktionsdefinition. Wie verschachtelte Funktionsdefinitionen, können in einer `lamdba`-Form Variablen der umgebenden Namensräume referenziert werden:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

4.7.6. Dokumentationsstrings

Hier nun ein paar Konventionen zum Inhalt und Formatieren von Dokumentationsstrings.

Die erste Zeile sollte immer eine kurze, prägnante Zusammenfassung des Zwecks des Objekts sein. Wegen der Kürze, sollte es nicht explizit auf den Namen oder den Typ des Objekts hinweisen, da diese durch andere Wege verfügbar sind (es sei denn, wenn der Name ein Verb ist, das den Ablauf der Funktion beschreibt). Dieser Zeile sollte mit einem Großbuchstaben anfangen und mit einem Punkt enden.

Enthält der Dokumentationsstring mehrere Zeilen, dann sollte die zweite Zeile leer sein, um die Zusammenfassung visuell vom Rest der Beschreibung zu trennen. Die folgenden Zeilen sollten aus einem oder mehrere Absätzen bestehen, die die Konventionen zum Aufruf des Objektes erläutern, seine Nebeneffekte etc.

Der Python-Parser entfernt die Einrückung von mehrzeiligen Zeichenkettenliteralen nicht, sodass Werkzeuge, die die Dokumentation verarbeiten, die Einrückung entfernen müssen, sofern das gewünscht ist. Dies geschieht aufgrund folgender Konvention: Die erste nicht-leere Zeile nach der ersten Zeile bestimmt den Umfang der Einrückung für den gesamten Dokumentationsstring. (Die erste Zeile kann man dafür nicht benutzen, da sie normalerweise neben dem öffnenden Anführungszeichen des Zeichenkettenliterals, sodass die Einrückung im Literal nicht erscheint.) Entsprechend dieser Einrückung werden vom Anfang jeder Zeile der Zeichenkette Leerzeichen entfernt. Zeilen, die weniger eingerückt sind, sollten nicht auftauchen, falls doch sollten alle führenden Leerzeichen der Zeile

entfernt werden. Die Entsprechung der Leerzeichen sollte nach der Expansion von Tabs (üblicherweise zu 8 Leerzeichen) überprüft werden.

Hier ein Beispiel eines mehrzeiligen Docstrings:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print(my_function.__doc__)
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.8. Intermezzo: Schreibstil

Jetzt da Du längere, komplexere Stücke in Python schreibst, ist es an der Zeit einmal über den Schreibstil (coding style) zu sprechen. Viele Sprachen können in verschiedenen Stilen geschrieben (präziser: formatiert) werden; davon sind manche lesbarer als andere. Es anderen leichter zu machen Deinen Code zu lesen ist immer eine gute Idee und sich einen schönen Schreibstil anzugewöhnen hilft dabei ungemein.

Für Python hat sich **PEP 8** als der Styleguide herauskristallisiert, dem die meisten Projekte folgen. Es fördert einen sehr lesbaren Schreibstil, der angenehm zu lesen ist. Jeder Pythonentwickler sollte ihn irgendwann einmal lesen, hier jedoch die wichtigsten Punkte:

- Benutze eine Einrückung von 4 Leerzeichen, keine Tabs.

  4 Leerzeichen sind ein guter Kompromiss zwischen geringer Einrückung, die eine größere Verschachtelungstiefe ermöglicht, und größerer Einrückung, die den Code leichter lesbar macht. Tabs führen zu Unordnung und sollten deshalb vermieden werden.
- Breche Zeilen so um, dass sie nicht über 79 Zeichen hinausgehen.

  Das ist hilfreich für Benutzer mit kleinen Bildschirmen und macht es auf größeren möglich mehrere Dateien nebeneinander zu betrachten.
- Benutze Leerzeilen, um Funktion und Klassen, sowie größere Codeblöcke innerhalb von Funktionen zu trennen.
- Verwende eine eigene Zeile für Kommentare, sofern das möglich ist.
- Schreibe Docstrings.
- Benutze Leerzeichen um Operatoren herum und nach Kommas, jedoch nicht direkt innerhalb von Klammerkonstrukten: `a = f(1, 2) + g(3, 4)`.
- Benenne Deine Klassen und Funktionen konsistent: Die Konvention schlägt `CamelCase` für Klassen und `klein_geschrieben_mit_unterstrichen` für Funktionen

und Methoden vor. Benutze immer `self` als Namen für das erste Methoden Argument (mehr zu Klassen und Methoden, siehe Eine erste Betrachtung von Klassen).

- Benutze keine ausgefallenen Dateikodierungen, wenn Dein Code für ein internationales Publikum vorgesehen ist. Pythons Standardkodierung – UTF-8 – oder sogar einfaches ASCII ist in jedem Fall am Besten.
- Benutze auch keine nicht-ASCII-Zeichen in Bezeichnern, wenn es auch nur den Hauch einer Chance gibt, dass der Code von Menschen gelesen oder gewartet wird, die eine andere Sprache sprechen.

Fußnoten

[1] Eigentlich wäre *call by object reference* eine bessere Beschreibung, denn wird ein veränderbares Objekt übergeben, sieht der Aufrufende jegliche Veränderungen, die der Aufgerufene am Objekt vornimmt (beispielsweise Elemente in eine Liste einfügt)