6 Funktionen

In den vorherigen Kapiteln haben wir immer wieder auf Funktionen zurückgegriffen, welche uns von Python zur Verfügung gestellt wurden. Wenn du beispielsweise den Befehl len("Haus") eingibst, bekommen wir von der Funktion len() den Rückgabewert 4 zurück. Dieser Wert ändert sich natürlich, wenn ein anderes Argument als "Haus"der Funktion übergeben wird.
In diesem Kapitel besprechen wir, wie man selber solche Funktionen definieren kann. Es gibt verschiedene Gründe, warum es Sinn macht Funktionen zu definieren.

1. Mehrer hintereinander ausgeführte Anweisungen können unter einem Namen zusammengefasst werden. Es kann also als Strukturierungselement angesehen werden, das eine Menge von Anweisungen gruppiert.

2. Ein längeres Programm erhält durch Funktionen eine Struktur, welche helfen kann, den Code besser lesen und verstehen zu können.

3. Ein Funktionsname kann dabei helfen zu verstehen, was das Unterprogramm berechnet oder ausführt.

4. Muss eine Codesequenz mehr als einmal ausgeführt werden, so braucht man nur den Funktionsnamen aufzurufen (Vermeidung von Codeduplizität).

Betrachten wir das folgende Beispiel:

```
1  from random import randint
2
3  eingabe = int(input("Gib eine positive ganze Zahl an: "))
4
5  liste = []
6  for i in range(eingabe):
7      liste.append(randint(0,100))
8
9  result = 0
10 for i in range(eingabe):
11     result = result + liste[i]
12
13 result = result / len(liste)
14 print("Das Ergebnis lautet " + str(result) + ".")
```

Es benötigt eine Weile um zu verstehen, was das Programm genau macht. Bei genauerem Hinsehen sieht man, dass die paar Codezeilen aus zwei Hauptteilen besteht: Zuerst wird eine zufällig, ganzzahlige Liste erstellt und danach deren arithmetischen Mittelwert berechnet.

Lagern wir diese zwei Hauptteile in Funktionen mit geeigneten Namen aus, so wird das Programm verständlicher zu lesen sein, ohne sich um programmtechnische Details kümmern zu müssen:

```
1  from random import randint
2  def zufallsliste_erstellen(eine_zahl):
3      liste = []
4      for i in range(eine_zahl):
5          liste.append(randint(0,100))
6      return liste
7  def berechne_mittelwert(eine_liste):
8      result = 0
9      for i in range(len(eine_liste)):
10         result = result + eine_liste[i]
11     result = result / len(eine_liste)
12     print("Das Ergebnis lautet " + str(result) + ".")
13
14 # Hauptprogramm
15 eingabe = int(input("Gib eine positive ganze Zahl an: "))
16 zufallsliste = zufallsliste_erstellen(eingabe)
17 berechne_mittelwert(zufallsliste)
```

Das Programm besteht nun wesentlich aus den Zeilen 15-17. Nur das Lesen dieser 3 Zeilen reicht aus, um zu verstehen, was das gesamte Programm macht. Die Anweisungen für das Erstellen einer Zufallsliste und die Berechnung des arithmetischen Mittelwertes wurden in Funktionen ausgelagert (siehe Zeile 1-12).
Zusätzlich stehen die Möglichkeiten der beiden Funktionen zu jedem beliebigen Zeitpunkt später im Programm wieder zur Verfügung. D.h. alleine durch den Funktionsaufruf zufallsliste_erstellen() kann jederzeit im Programm wieder eine Zufallsliste erstellt werden, ohne die ganzen Anweisungen nochmals aufschreiben zu müssen.
Betrachten wir nun im Detail, wie Funktionen in Python erstellt werden können.

6.1 Eine Funktion ohne Rückgabewert definieren

Mit dem Keyword def führen wir eine neue Funktion ein. Nach der Anweisung def steht der Name der Funktion, gefolgt von runden Klammern (). In der Klammer () werden die Argumente, falls welche verlangt, aufgelistet. Zum Schluss kommt noch der obligate Doppelpunkt :. Die darauffolgende Zeilen müssen wie üblich eingerückt sein, ansonsten gehören sie nicht mehr zur Funktion.
Hier ein Beispiel einer Funktion, welche eine Zahl als Übergabeparameter erwartet. Die Funktion selber multipliziert die Eingabe mit 2 und gibt das Resultat auf der Konsole wieder aus.

```
1  def mit_zwei_multiplizieren(eingabe):
2      eingabe = 2*eingabe
3      print("Verdopple ich diese Zahl, so erhalte ich ",eingabe)
4  zahl = int(input("Gib eine ganze Zahl ein: "))
5  mit_zwei_multiplizieren(zahl)
6  zahl = int(input("Gib eine weitere ganze Zahl ein: "))
7  mit_zwei_multiplizieren(zahl)
8  print("Danke für die Eingabe.")
```

Jedes Mal wenn im Programm die Funktion mit_zwei_multiplizieren() aufgerufen wird, wird der Codeblock bei der Definition der Funktion (in unserem Beispiel Zeile 2 und 3) ausgeführt.
Natürlich können der Funktion auch mehr als nur ein Argument übergeben werden.

```
1 # Eingabe: Längen des Rechtecks
2 # Ausgabe in der Konsole: Fläche des Rechtecks
3 def flaeche_rechteck(a,b):
4     print("Die Fläche des Rechtecks ist "+str(a*b)+".")
```

Eine mögliche Ausgabe könnte dann folgendermassen aussehen:

```
>>> flaeche_rechteck(3,7)
Die Fläche des Rechtecks ist 21.
>>> flaeche_rechteck(2.3,5.1)
Die Fläche des Rechtecks ist 11.729999999999999.
```

6.1.1 Aufgaben

1. Erstelle eine Funktion, welche einen String als Argument erwartet und den ersten und letzten Buchstaben des Strings ausgibt.

2. Schreibe eine Funktion summe(), welche für die Eingabe einer Zahl n folgendes Resultat ausgibt:

   $$summe := 1 + 2 + \cdots + n$$

   Es sollte dann z.B. folgendermaßen aussehen:

   ```
   >>> summe(4)
   10
   >>> summe(100)
   5050
   ```

3. Definiere eine Funktion teilermenge(zahl), welche die Menge der Teiler von zahl ausgibt.

   Zum Beispiel:

   ```
   >>> teilermenge(24)
   [1, 2, 3, 4, 6, 8, 12, 24]
   ```

6.2 Eine Funktion mit Rückgabewert definieren

Unsere selbst geschriebenen Funktionen von oben haben bisher die Resultate lediglich auf der Konsole ausgegeben. Jedoch kann es sein, dass die Ergebnisse für den weiteren Programmverlauf gebraucht und weiter verarbeitet werden müssen. In solchen Fällen macht es Sinn Funktionen zu definieren, welche mir ein Ergebnis zurückgeben, wie z.B.

```
>>> len("Haus")
4
```

Nehmen wir das gleiche Beispiel von oben:

```
1 # Eingabe: Längen des Rechtecks
2 # Ausgabe in der Konsole: Fläche des Rechtecks
3 def flaeche_rechteck(a,b):
4     print("Die Fläche des Rechtecks ist "+str(a*b)+".")
```

Diese Funktion gibt auf der Konsole die Fläche des Rechtecks mit Längen a und b aus. Möchte man das Ergebnis nicht ausgeben, sondern z.B. für eine Weiterverarbeitung zurückgeben, so kann dies mit der return Anweisung gemacht werden:

```
1 def flaeche_rechteck(a,b):
2     return a*b
```

Nun kann das Ergebnis der Funktion flaeche_rechteck() in einer Variable gespeichert und weiter verwendet werden.

```
>>> a = flaeche_rechteck(2,5)
>>> a
10
```

Bemerkung
Eine Funktion mit einem return Statement kommt dem Konzept einer Funktion im Sinne der Mathematik sehr nahe:

$$y = f(x)$$

wobei

f : Funktionsname
x : Argument
y : Rückgabewert

Mehr zum Thema Funktionen findest du in der Dokumentation unter
`https://docs.python.org/3.2/tutorial/controlflow.html#defining-functions`

6.2.1 Aufgaben

1. In den vorherigen Kapiteln hast du die Fakultät in den Aufgaben bereits kennengelernt.

$$n! := n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

   (a) Definiere eine Funktion fakultaet(zahl), welche die Fakultät von zahl berechnet und als Ergebnis zurückgibt.

   (b) Schreibe nun ein kleines Programm, welches zwei natürliche Zahlen einliest und jeweils deren Fakultät berechnet und ausgibt.

2. Schreibe eine Funktion quersumme(zahl), welche die Quersumme von zahl berechnet und zurückgibt.

3. Schaue dir die folgende Funktion an und überlege, was das Ergebnis sein könnte.

```
1 def meine_funktion(a,b):
2     while b != 0:
3         t = a%b
4         a = b
5         b = t
6     return a
```

   Hast du eine Vermutung? Teste sie an einigen Beispielen:

```
>>> meine_funktion(12,16)
>>> meine_funktion(1,16)
>>> meine_funktion(8,16)
>>> meine_funktion(8,21)
>>> meine_funktion(18,21)
>>> meine_funktion(27,36)
```

4. Welche der folgenden Definitionen sind zulässig? Welche nicht und wieso? Überlege zuerst und tippe es danach zur Kontrolle ein.

   Definition 1:

```
def print():
    print("Hallo Welt")
```

   Definition 2:

```
def print1()
    print("Hallo Welt")
```

   Definition 3:

```
def print1():
    print("Hallo Welt")
```

   Definition 4:

```
def print1():
print("Hallo Welt")
```

5. Schreibe zwei Funktionen mit folgender Funktionalität:

   (a) dez_to_bin(dezzahl): Die Funktion wandelt die Dezimalzahl dezzahl in die entsprechende Binärzahl um. [7]

   (b) bin_to_dez(binzahl): Die Funktion wandelt die Binärzahl binzahl in die entsprechende Dezimalzahl um.

   (c) Teste deine beiden Funktionen indem du sie mit den von Python zur Verfügung gestellten Funktionen vergleichst (bin() und int()):

```
>>> bin(24)# Konvertiert dezimal zu binär
'0b11000'
>>> int('11000',2)# Konvertiert binär zu dezimal
24
```

6.3 Mehrere Rückgabewerte

Bis jetzt haben unsere Funktionen immer nur einen Wert zurückgeben. Mit Python ist es überhaupt kein Problem, auch Funktionen zu definieren, welche mehr als einen Wert zurückgeben. Hier ein Beispiel dazu:

```
1 def add_multiply(zahl1, zahl2):
2     return zahl1+zahl2, zahl1*zahl2
3
4 a = add_multiply(4,10)
5 print(a, a[0], a[1])
6
7 a, b = add_multiply(4,10)
8 print(a, b)
```

Dieses Programm gibt die Summe und das Produkt zweier Zahlen (hier zahl1 und zahl2) zurück. Folgende Ausgabe erhalten wir auf der Konsole:

```
(14, 40) 14 40
14 40
```

Wir sehen bei der Zuweisung in Zeile 4, dass die mehreren Rückgabewerte als Tupel der einen Variable a übergeben werden. Man kann aber auch jeden Rückgabewert einer einzigen Variablen zuweisen, so wie es in Zeile 7 gemacht wurde.

[7] Das Binärsystem (auch Dualsystem genannt) ist ein Zahlensystem, welches zur Darstellung von Zahlen nur zwei verschiedene Ziffern (0 und 1) benutzt. Jede Zahl kann somit nur mit Hilfe von 0 und 1 dargestellt werden. Für mehr Informationen siehe unter `https://de.wikipedia.org/wiki/Dualsystem`

6.4 lambda-Operator

Der lambda-Operator bietet eine Möglichkeit anonyme Funktionen, also Funktionen ohne Namen, zu schreiben und zu benutzen. Anstelle des Keywords def gebrauchen wir das Keyword lambda. Sie können, wie normal definierte Funktionen, eine beliebe Anzahl von Parametern haben, führen Befehle aus und können einen Rückgabewert liefern. Die Syntax sieht folgendermassen aus:

```
lambda [arg1 [,arg2,.....argn]]:expression
```

Das nächste Beispiel zeigt den Unterschied zwischen einer normalen Funktion und einer lambda Funktion:

```
>>> def f(x): return x**2
>>> print(f(8))
64
>>> g = lambda x: x**2
>>> print(g(8))
64
```

Wir sehen, dass die beiden Funktionen f und g genau das gleiche ausführen und auf die gleiche Weise benutzt werden können. Sie unterscheiden sich hier lediglich in der Definition. Wie aber der Name (anonyme Funktionen) schon sagt, müssen wir einem lambda-Ausdruck keinen Namen zuweisen, was wir am folgenden Beispiel gut sehen:

```
>>> (lambda x: x**2)(8)
64
```

Der lambda-Operator kann dann gut gebraucht werden, wenn beispielsweise eine Funktion als Argument einer weiteren Funktion übergeben werden soll und man diese erste Funktion nachher nicht mehr braucht. Schauen wir dazu folgendes Beispiel an:

```
1  def temp_funktion(x):
2      return x + 42
3
4  def make_list(f,groesse):
5      ergebnis = []
6      for i in range(groesse):
7          ergebnis.append(f(i))
8      return ergebnis
9
10 a = make_list(temp_funktion, 5)
11     print(a)
```

Diese Programm liefert dann folgenden Output:

```
[42, 43, 44, 45, 46]
```

Nehmen wir an, dass die Funktion temp_funktion() in einem späteren Programmverlauf nie mehr gebraucht wird. So können wir uns die Namensvergabe sparen und ändern denn Code folgendermassen:

```
def make_list(f,groesse):
    ergebnis = []
    for i in range(groesse):
        ergebnis.append(f(i))
    return ergebnis

a = make_list(lambda x: x + 42, 5)
    print(a)
```

Dieses Programm hat den gleichen Output wie das erste, jedoch mussten wir nicht eine Funktion definieren, welche wir später sowieso nicht gebraucht hätten.

6.4.1 Aufgaben

1. Welche der folgenden Eingaben sind zulässig? Welche nicht und wieso? Wie müsste es richtig sein? Überlege zuerst und tippe es danach zur Kontrolle ein.

```
>>> a = lambda arg1, arg2: arg1 + arg2
>>> print(a(0,2))
>>> a = (lambda arg1, arg2: arg1 + arg2)(0,2)
>>> print(a(0,2))
>>> a = lambda arg1, arg2: arg1 + arg2
>>> print(a(2+2))
>>> a = lambda arg1, arg2: arg1 + arg2
>>> b = lambda x: a(2, x)
>>> print(b(3,4))
```

2. Gegeben ist folgendes Programm:

```
def f2(d):
    return 2**d
def f3(d):
    return 3**d
def f5(d):
    return 5**d
def f7(d):
    return 7**d
def add_function(f,g):
    return f(2) + g(2)

print(add_function(f2, f3))
print(add_function(f5, f7))
```

Schreibe das Programm mit dem lambda-Operator so um, dass die Zeilen 1-12 weggelassen werden können, jedoch der gleiche Output produziert wird.

Abgewandelt von `https://pythonbuch.com/funktion.html`