Programmierung mit Python für Programmierer

Kapitel 3 - Funktionen, Module, Pakete

Autor: Dr. Christian Heckler

Vorbemerkungen

- Verwendete Literatur: siehe Referenzen
- Verwendete Symbole:
  - ❗: Beispielprogramm
  - ❗: Weitere Erläuterungen im Kurs
  - ❓: Übung

Funktionen - der einfachste Fall

Definition einer Funktion:

```
def funktionsname(): 1
    anweisung(en)    2
```

1. *Funktionskopf*: Schlüsselwort `def` gefolgt von einem beliebigen *Funktionsnamen* und `()` und `:`.
   Für den Funktionsnamen gelten dieselben Regeln und Konvetionen wie für Variablennamen.
2. *Funktionsrumpf*: Eingerückter *Codeblock* (vgl. Schleifen)
   Im Funktionsrumpf können beliebige Anweisungen stehen, also auch If-Anweisungen, Schleifen, Funktionsaufrufe.

Aufruf einer Funktion

- Analog zu den bisher verwendeten Funktionen: Angabe des Funktionsnamens gefolgt von Klammern.
- Beim Aufruf werden die Anweisungen des Funktionsrumpfes durchlaufen. Danach kehrt der Programmfluß an die Aufrufstelle zurück.

Eine einfache Funktion

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

sage_hallo()
```

Funktionsparameter

- Wir haben gesehen, dass Funktionen *Argumente* haben können (z.B. `abs(5)`).
- Diese werden beim Aufruf innerhalb der runden Klammern angegeben.
- Bei der **Funktionsdefinition** werden Variable für die erwarteten Parameter ebenfalls in den runden Klammern angegeben.
- Eine Funktion kann beliebig viele Parameter haben.
- Die Variablen für die Parameter sind nur in der Funktion bekannt (*lokale Variable*) und bekommen die beim Aufruf übergebenen Werte.

Funktionsparameter - Beispiel

```
def berechne_umfang(laenge, breite): (1)
    umfang = 2 * (laenge + breite)
    print("Der Umfang ist", umfang)

x = 4
berechne_umfang(x, 7) (2)
berechne_umfang(2*5, 1+x) (3)
```

(1) Die Funktion `berechne_umfang` [Kle] erwartet zwei *Argumente*.

- Das erste Argument wird während der Funktionsausführung in der (*lokalen*) Variablen `laenge` gespeichert.
- Das zweite Argument wird während der Funktionsausführung in der (*lokalen*) Variablen `breite` gespeichert.

(2) Aufruf der Funktion mit den Werte `4` und `7`. Während der Funktionsausführung hat also die Variable `laenge` den Wert `4` und die Variable `breite` den Wert `7`.

(3) Analog: `laenge` erhält den Wert `10` und `breite` den Wert `5`.

Return und Rückgabewert

- Eine Funktion wird nach der letzten Anweisung des Funktionsrumpfes verlassen.
- Eine Funktion kann explizit mit der Return-Anweisung verlassen werden.
- Mit der Return-Anweisung kann ein Rückgabewert an die Aufrufstelle zurückgegeben werden:

```
return Ausdruck
```

- Es können beliebige Werte zurückgegeben werden, also auch Listen, Dictionaries ...
- Die Auswertung des Funktionsaufrufes ergibt den Rückgabewert, der an der Aufrufstelle wie gehabt verwendet werden kann.
- Wird kein Wert explizit zurückgegeben, so ist der Rückgabewert das `None`-Objekt (Bsp: `print`-Funktion).

Rückgabewert - Beispiel

```
def berechne_umfang(laenge, breite):
    umfang = 2 * (laenge + breite)
    return umfang (1)

print(berechne_umfang(7,5)) (2)
doppelter_umfang = 2 * berechne_umfang(3,2)
```

1. Beim Erreichen dieser Anweisung wird die Funktion verlassen und der Wert der Variablen `umfang` zurückgegeben.
2. Der Rückgabewert kann an der Aufrufstelle beliebig verwendet werden.

Parameterübergabe

- Es können beliebige Objekte beliebigen Typs übergeben werden.
- **Es werden Referenzen auf die Objekte übergeben.** ℹ
- Konsequenzen (siehe die folgenden Beispiele):
  - Weist man in der Funktion einem Parameter einen anderen Wert zu, bleibt der Wert an der Aufrufstelle unverändert.
  - Ändert man in der Funktion über den Parameter das übergebene Objekt (bei veränderlichen Objekten), ist es auch an der Aufrufstelle verändert.

> Das nennt man *Seiteneffekt* des Funktionsaufrufs und sollte vermieden werden.
> Die Idee einer Funktion ist, für Eingaben (Parameter) ein Ergebnis zu liefern - und nicht, sonstige Auswirkungen auf den Programmverlauf zu haben. Das vereinfacht das Verständnis und die Wartbarkeit (und vereinfacht das Erstellen „paralleler“ Programme).

Beispiel: Änderung des Wertes eines Parameters

```
def meine_funktion(b):
    print(b)
    b = 2
    print(b) (1)

a = 1
meine_funktion(a)
print(a) (2)
```

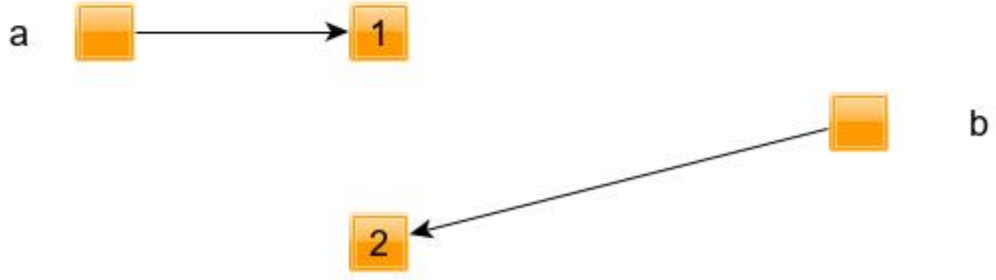
(1) Die lokale Variable `b` hat nun den Wert `2`.

(2) Die Variable `a` im Hauptprogramm hat nach wie vor den Wert `1`

Veranschaulichung

Hauptprogramm meine_funktion

b=2



Beispiel: Änderung des übergebenen Objektes

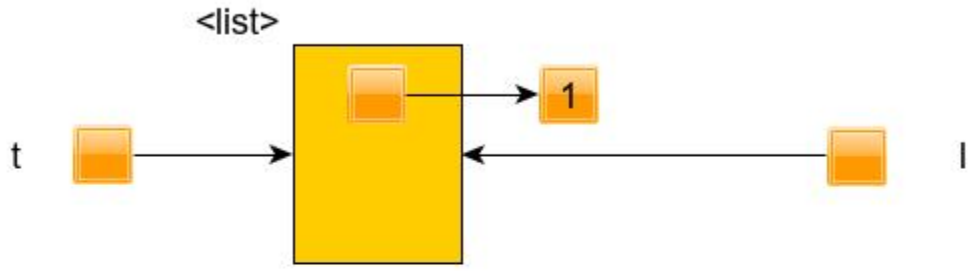
```
def meine_funktion(l):
    l[0] = 2 (1)

t = [1]
meine_funktion(t)
print(t[0]) (2)
```

(1) Das `0.` Element der übergebenen Liste wird verändert.

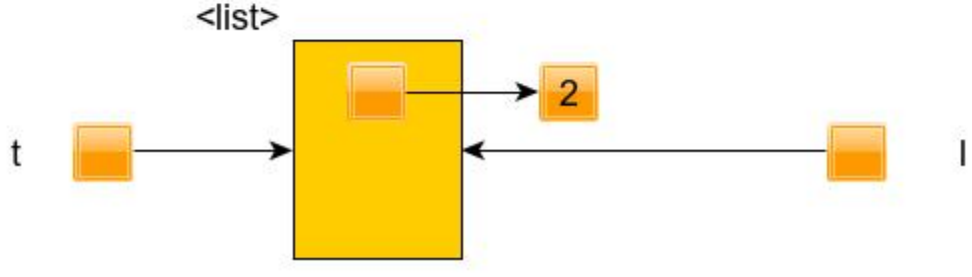
(2) Damit ist auch die Liste an der Aufrufstelle verändert. Ausgabe ist also `2`.

Veranschaulichung

Hauptprogramm meine_funktion



l[0]=2



Rückgabewert

- Analog zur Parameterübergabe wird bei der Rückgabe eine Referenz auf das Rückgabeobjekt übergeben, siehe `rueckgabe.py` ❗
- Es können beliebige Objekte zurückgegeben werden, also auch Tupel:

```
def berechne_umfang_flaeche(laenge, breite):
    umfang = 2 * (laenge + breite)
    flaeche = laenge * breite
    return (umfang, flaeche)

print(berechne_umfang_flaeche(2,3))
umfang, flaeche = berechne_umfang_flaeche(4,8)
print("Umfang:", umfang, ", Flaeche:", flaeche)
```

Lokale Variable

- Variablen, die innerhalb einer Funktion „deklariert“ sind:
  - Die Funktionsparameter.
  - Variablen, denen innerhalb der Funktion ein Wert zugewiesen wird.
- Sind nur innerhalb der zugehörigen Funktion *sichtbar* (d.h. „zugreifbar“).

```
def erhoehe(x):
    y = x + 1
    return y

print(x) # Fehler: x ist hier nicht bekannt
print(y) # Fehler: y ist hier nicht bekannt
```

Globale Variable

- Sind außerhalb von Funktionen definiert.
- Sind in Funktionen *sichtbar* („zugreifbar“), wenn sie vor der Funktionsdefinition erzeugt wurden.

```
x = 1

def erhoehe():
    return x + 1

print(erhoehe())
```

Überdeckung globaler Variablen

- Eine globale Variable kann durch eine lokale Variable *überdeckt* werden.
- In diesem Fall wird innerhalb der Funktion auf die lokale Variable zugegriffen und außerhalb auf die Globale.

```
s = "global"

def meine_funktion():
    s = "lokal"
    print(s)

meine_funktion() # lokal
print(s) # global
```

Ändernder Zugriff auf eine globale Variable

- Durch die Zuweisung eines Wertes an die Variable `s` innerhalb der Funktion wurde eine neue lokale Variable `s` angelegt.
- Wie kann man in einer Funktion den Wert einer globalen Variablen ändern? Deklaration als `global`!

```
s = "global"

def meine_funktion():
    global s
    s = "lokal"
    print(s)

meine_funktion() # lokal
print(s) # lokal
```

Empfehlung zu globalen Variablen

- Vermeiden Sie den Zugriff auf globale Variable in Funktionen.
- Benennen Sie globale und lokale Variable unterschiedlich, so dass es gar nicht erst zu Namenskonflikten kommen kann.

Standardwerte für Funktionsparameter

- Bei der Definition einer Funktion kann man einem Parameter einen Wert zuweisen (*Standardwert*, *Default*). Wird das Argument beim Aufruf nicht angegeben, so wird der Standardwert genommen . Der Parameter kann beim Aufruf also weggelassen werden (*optionaler Parameter*). Bsp. [Kle]:

```
def umfang(laenge, breite=1):
    return 2 * (laenge + breite)

u = umfang(4) # breite hat in der Funktion den Wert 1
```

- Standardwerte können nur „von hinten“ vergeben werden. (Damit beim Aufruf die Zuordnung möglich ist.)

```
def f1(x,y=2,z=3):  # moeglich
    return

def f2(x=1,y,z=3):  # verboten
    return
```

Schlüsselwortparameter

- Bisher: Übergabe der Parameter an Hand der Position (*Positionsparameter*)

```
def umfang(laenge=1, breite=2):
    return 2 * (laenge + breite)

print(umfang(3,7)) # laenge ist 3, breite ist 7
print(umfang(3))   # laenge ist 3, breite ist 2
```

- Es ist auch möglich, die Parameter per Schlüsselwort zu übergeben. (*Schlüsselwortparameter*)

```
u = umfang(laenge=3, breite=7)
u = umfang(breite=7, laenge=3)
u = umfang(laenge=3)
u = umfang(breite=7) # das geht nicht über Positionsparameter
```

Schlüsselwortparameter 2

```
def umfang(laenge, breite):
    return 2 * (laenge + breite)

laenge = 1
print(umfang(1,2))
print(umfang(laenge = 2 * 5, breite = 7))
print(umfang(laenge = laenge, breite = laenge + 2)) 1
```

1 In der Funktion hat der Parameter `laenge` den Wert 1 und der Parameter `breite` den Wert 3.

- „Links“ vom Gleichheitszeichen steht der Parametername, wie er in der Funktion definiert ist (`laenge` bzw. `breite`)
- „Rechts“ vom Gleichheitszeichen steht ein **Ausdruck** (z.B. die (globale) Variable `laenge`). Dieser wird ausgewertet. Der Wert wird dem Parameter in der Funktion übergeben.

Funktionsparameter - was noch

Kommandozeilenparameter

docstring

Funktionsreferenzen

- Funktionen sind Objekte.
- Der Funktionsname ist eine Referenz auf das Funktionsobjekt.
- Eine Funktion kann also auch einer Variablen zugewiesen werden:

```
def f(x):
    print(x)

def g(y):
    print(y+1)

meine_funktion = f
f(4)
meine_funktion = g
g(4)
```

- Oder als Parameter einer anderen Funktion übergeben werden.

Funktionsreferenzen 2

- Unterscheide:

```
x = f      (1)
x = f()    (2)
```

(1) Die Funktion `f` wird der Variablen `x` zugewiesen.

(2) Die Funktion `f` wird aufgerufen und der Rückgabewert der Variablen `x` zugewiesen.

Anwendungsbeispiel: Sortierung von Listen

- Erinnerung: Eine Liste ist eine Sequenz von Objekten

```
l = []
l = [1, "Hallo"]
l.append("Welt")
```

- `l.sort()`: Die Liste wird sortiert (`l` wird verändert)
- `s = sorted(l)`: `s` ist eine neue sortierte Liste. `l` ist unverändert.
- `l.sort(reverse=True)`: Umgekehrte Sortierreihenfolge
- Sortierreihenfolge?
  Man kann einer Sortierfunktion eine Funktion mitgeben, die auf die Elemente angewendet wird. Verglichen werden dann nicht die Elemente selbst, sondern die Funktionswerte.

Sortierung von Listen mit `key` -Funktion

- Beispiel:

```
namen = [("Anton", "Wunderlich"),
         ("Berta","Müller"),
         ("Thomas", "Schmitz")]
```

- Sortierreihenfolge?
  - nach Vornamen
  - nach Nachnamen
- Sortierung nach Vornamen: ℹ

```
def vorname(t):
    return t[0]
namen.sort(key=vorname)
```

- siehe `funktions_ref.py` ❗

Anonyme Funktionen

Closures

Generatoren

Generator-Ausdrücke

Die Funktion `map`

- Beispiel: Funktion `map`
- `map(f,s)` wendet die Funktion f auf jedes Element der Sequenz `s` an und gibt einen Iterator zurück
  - (den man in eine Liste umwandeln kann oder
  - über den man mit for schleifen kann).
- Beispiel:
  - Sei `l` eine Liste von Zahlen, dann liefert
  - `list(map(math.sqrt, l))` Liste der Wurzeln der Zahlen

Die Funktion `reduce`

- Reduziert eine Sequenz auf einen Wert durch fortwährende Anwendung einer Funktion auf je zwei Sequenzelemente.
- Muss importiert werden:

```
from functools import reduce
```

- Beispiel:
  - Sei `add` eine Funktion, die zwei Werte addiert und `l` eine Liste von Zahlen, dann liefert
  - `reduce(add,l)` die Summe aller Listenelemente
- Die Funktion `add` muss man nicht definieren:
  - `reduce(lambda x,y: x+y, l)`
  - Der *lambda-Ausdruck* definiert eine *anonyme Funktion*

Einschub: Funktionale Programmierung statt Schleifen

- Funktionen sind also „normale“ Objekte, die man Variablen zuweisen kann und die man als Parameter anderen Funktionen übergeben kann.
- In der „funktionalen Programmierung“ macht man sich dies zu Nutze.
- Beispielsweise Verwendung von Funktionen statt Schleifen
- Motivation: In manchen Fällen sind Schleifen in Python zu langsam (z.B. Datenanalyse großer Datenmengen).
- Die entsprechenden Bibliotheken stellen dann andere Mechanismen bereit, z.B.
  - Anwendung einer Funktion auf alle Elemente einer Liste (ohne Schleife): `map`
  - Anwendung einer Operation auf je zwei Elemente einer Liste: `reduce`

Übungen Funktionen

- Binäre Suche
- Merge Sort
- Damenproblem
- Bestimmung der maximalen Wortlänge in einer Zeichenkette, die einen Satz darstellt, mit Hilfe von map / reduce.

Referenzen

- [Ste] Ralph Steyer: Programmierung Grundlagen, Herdt-Verlag
- [Kle] Bernd Klein: Einführung in Python 3, Hanser-Verlag
- [Kof] Kofler: Python – Der Grundkurs, Rheinwerk Computing
- [EK] Johannes Ernesti, Peter Kaiser: Python 3 – Das umfassende Handbuch, Rheinwerk Computing
- [Mat] Eric Matthes: Python Crashkurs – Eine praktische, projektbasierte Programmiereinführung, dpunkt.verlag
- [Swe] Sweigart: Eigene Spiele programmieren: Python lernen