

Programmierung mit Python für Einsteiger:: Kapitel 4 - Funktionen

Inhaltsverzeichnis

Funktionen

Funktionen - der einfachste Fall

Aufruf einer Funktion

Eine einfache Funktion

Funktionsparameter

Funktionsparameter - Beispiel

Return und Rückgabewert

Rückgabewert - Beispiel

Funktionen und EVA

🔗 Übung 08A: Funktion Wortlänge

🔗 Übung 08

Parameterübergabe

Beispiel: Änderung des Wertes eines Parameters

Veranschaulichung

Beispiel: Änderung des übergebenen Objektes

Veranschaulichung

🔗 Übung 09 - Parameterübergabe

Lokale Variable

Globale Variable

Überdeckung globaler Variablen

Ändernder Zugriff auf eine globale Variable

Empfehlung zu globalen Variablen

Standardwerte für Funktionsparameter

Schlüsselwortparameter

Schlüsselwortparameter 2

🔗 Weitere Übungen zu Funktionen [Kle]

Der Datentyp Tupel

Sortierung von Listen

Sortierung von Listen mit `key`-Funktion

Funktionsreferenzen

🔗 Übung 10 - Römische Zahlen sortieren

🔗 Weitere Übungen Funktionen [Kle]

Einschub: Funktionale Programmierung statt Schleifen

Funktionale Programmierung: `map`

Funktionale Programmierung: `reduce`

🔗 Übung 11 - Funkt. Progr.: Wortlänge

Referenzen

Funktionen

- Erinnerung: Aufruf von Funktionen:

```
y = int(input("Geben Sie eine Zahl ein"))
x = abs(y)
print(x)
```

- Funktionen können vielfach und an verschiedenen Stellen aufgerufen werden.
- Sie dienen als der Wiederverwendung von Code.
- Können aufgerufen werden, ohne dass man die Implementierung kennt.
- Inhalt des Kapitels: Eigene Funktionen programmieren.

Funktionen - der einfachste Fall

Definition einer Funktion:

```
def funktionsname(): ❶
    anweisung(en)    ❷
```

- ❶ *Funktionskopf*: Schlüsselwort `def` gefolgt von einem beliebigen *Funktionsnamen* und `()` und `:`. Für den Funktionsnamen gelten dieselben Regeln und Konventionen wie für Variablennamen.
- ❷ *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): ❶
    umfang = 2 * (laenge + breite)
    print("Der Umfang ist", umfang)

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

- ❶ 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.
- ❷ 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.
- ❸ 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 ❶  
  
print(berechne_umfang(7,5)) ❷  
doppelter_umfang = 2 * berechne_umfang(3,2)
```

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

Funktionen und EVA

Auch Funktionen genügen dem EVA-Prinzip:

- **Eingabe:** Parameter
- **Verarbeitung:** Im Funktionsrumpf
- **Ausgabe:** Rückgabewert per `return`

❓ Übung 08A: Funktion Wortlänge

- Schreiben Sie eine Funktion `berechne_max_wortlaenge`
 - Eingabeparameter: Eine Zeichenkette, die einen Satz darstellt.
 - Rückgabewert: Die Länge des längsten Wortes
- siehe Ordner `uebungen\08A_uebung_funktion_wortlaenge`
- **Zusatzaufgabe:** Geben Sie auch die Nummer des längsten Wortes zurück.

❓ Übung 08

- Wandeln Sie das Programm zur Berechnung der Anzahl eines Zeichens in einer Zeichenkette in eine Funktion um.

- Eingabe der Funktion:
 - Zeichenkette
 - Zeichen
- Ausgabe der Funktion:
 - Anzahl der Vorkommen des Zeichens in der Zeichenkette
- s. uebungen\08_uebung_funktion_buchstaben

Parameterübergabe

- Es können beliebige Objekte beliebigen Typs übergeben werden.
- **Es werden Referenzen auf die Objekte übergeben.** ⓘ
- 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) ❶  
  
a = 1  
meine_funktion(a)  
print(a) ❷
```

- ❶ Die lokale Variable `b` hat nun den Wert `2`.
- ❷ Die Variable `a` im Hauptprogramm hat nach wie vor den Wert `1`

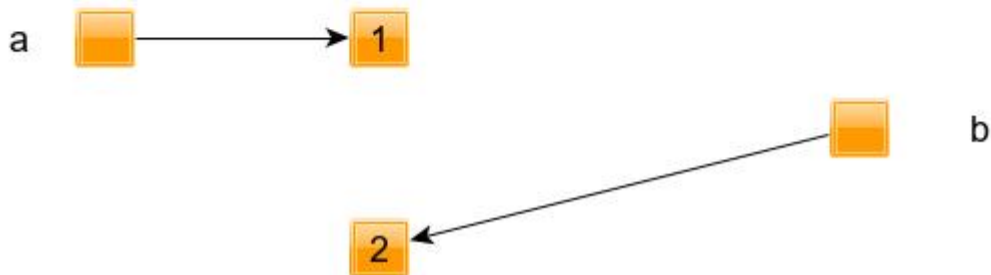
Veranschaulichung

Hauptprogramm

meine_funktion



b=2



Beispiel: Änderung des übergebenen Objektes

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

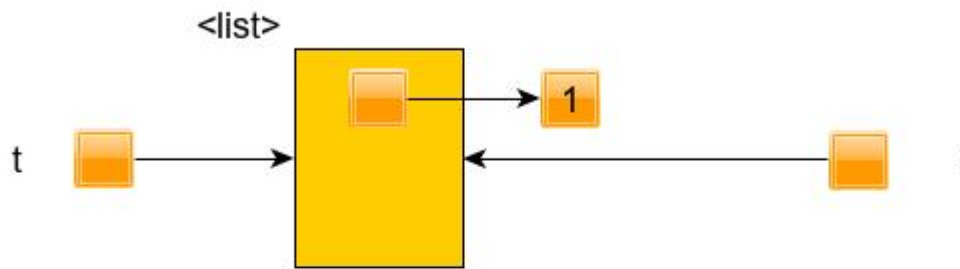
```
t = [1]  
meine_funktion(t)  
print(t[0]) ❷
```

- ❶ Das 0. Element der übergebenen Liste wird verändert.
- ❷ Damit ist auch die Liste an der Aufrufstelle verändert. Ausgabe ist also 2.

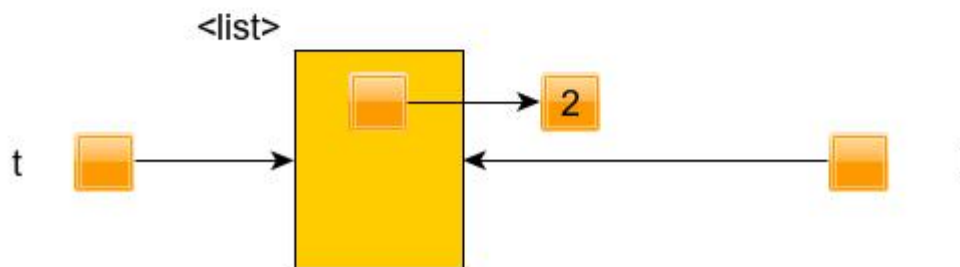
Veranschaulichung

Hauptprogramm

meine_funktion



`l[0]=2`



❓ Übung 09 - Parameterübergabe

- Schreiben Sie eine Funktion die in einer Liste alle Vorkommen von 8 durch 16 ersetzt.
- Die Funktion soll die Anzahl der ersetzten Elemente zurückgeben.
- siehe Ordner `uebungen/09_uebung_parameteruebergabe`



Es handelt sich nur um eine Übung. Wie beschrieben, handelt es sich bei der Änderung einer übergebenen Liste in einer Funktion um einen *Seiteneffekt*, der vermieden werden sollte. In dem Beispiel könnte man z.B. eine Kopie der Liste zurückgeben (was aber entsprechenden Speicherplatz beansprucht).

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)) ❶
```

- ❶ In der Funktion hat der Parameter `laenge` den Wert 1 und der Parameter `breite` den Wert 3.
- „Links“ vom Gleichheitszeichen steht der Parameternamen, 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.

❶ Weitere Übungen zu Funktionen [Kle]

- Schreiben Sie eine Funktion, die überprüft, ob es sich bei einer übergebenen Zeichenkette um ein *Palindrom* handelt (ein Wort, das von hinten und vorne gelesen gleich ist, z.B. „Anna“).
- Schreiben Sie eine Funktion, die die Zeichenkette, die sie als Eingabe erhält, in die sogenannte „Löffelsprache“ übersetzt. Die Codierung erfolgt nach folgenden Regeln: Immer, wenn ein Vokal kommt, wird dieser verdoppelt und dazwischen ein vorher festgelegter String gesetzt. Häufig wird für diesen String „lew“ oder „lef“ verwendet. Diphthonge (ei, au, ie, eu) werden wie ein Vokal betrachtet.


Der Datentyp Tupel

- Analog Liste: Ein Tupel enthält eine Sequenz beliebiger Werte
- Unterschied zu Liste: Ein Tupel ist unveränderlich.
- Definition mit runden Klammern.
- Beispiel:

```
name = ("Christian", "Heckler")
print(name[0]) # Christian
name[0]="Christoph" # Fehler
```

- Wird oft verwendet, um zusammengehörige Daten zu „bündeln“ (vgl. „Rekord“), z.B. Adresse:

```
("Max", "Mustermann", "Irgendwo 42", "42666 Nirgendwo")
```

- siehe `tupel.py` 

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: ⓘ

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

- siehe `funktions_ref.py` ⓘ

Funktionsreferenzen

- Man sieht: Man kann eine Funktion als Parameter an eine andere Funktion übergeben
- Der Parameter `key` der `sort`-Funktion referenziert die Funktion `vorname`
- Eine Funktion kann also auch einer Variablen zugewiesen werden:

```
meine_funktion = vorname
meine_funktion("Christian", "Heckler")
```

- 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.

❓ Übung 10 - Römische Zahlen sortieren

Schreiben Sie ein Programm, das eine Liste von römischen Zahlen nach ihrem Wert sortiert.

siehe Ordner `10_uebung_roemische_zahl_sortieren`

❓ Weitere Übungen Funktionen [Kle]

- **Aufgabe 1:** Schreiben Sie eine Funktion, die die Reihenfolge einer Liste umkehrt. Verzichten Sie aber auf die Benutzung der Listen-Methode `reverse` und versuchen Sie stattdessen nur die Methoden `pop` und `append` zu benutzen.

- **Aufgabe 2:** Schreiben Sie mit Hilfe der Methoden `extend` und `append` (`type` wird auch benötigt) eine Funktion `flatten`, die eine verschachtelte Liste (oder ein Tupel) in eine flache Liste überführt. Hinweis: Es handelt sich um eine rekursive 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`
-

Funktionale Programmierung: `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
-

Funktionale Programmierung: `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*
-

🔗 Übung 11 - Funkt. Progr.: Wortlänge

- Schreiben Sie ein Programm, das für einen Satz (gegeben in einer Zeichenkette) die maximale Länge eines Wortes ausgibt.
- Benutzen Sie keine Schleifen, sondern `map` und `reduce`.
- Hinweis: Für eine Zeichenkette `s` gibt `s.split()` eine Liste der Wörter an, z.B.:

```
s = "Dies ist ein Satz".  
wortliste = s.split() # ["Dies", "ist", "ein", "Satz"]
```

- siehe Ordner `11_uebung_satz_map_reduce`
-

Referenzen

- [Kle] Bernd Klein, Einführung in Python 3