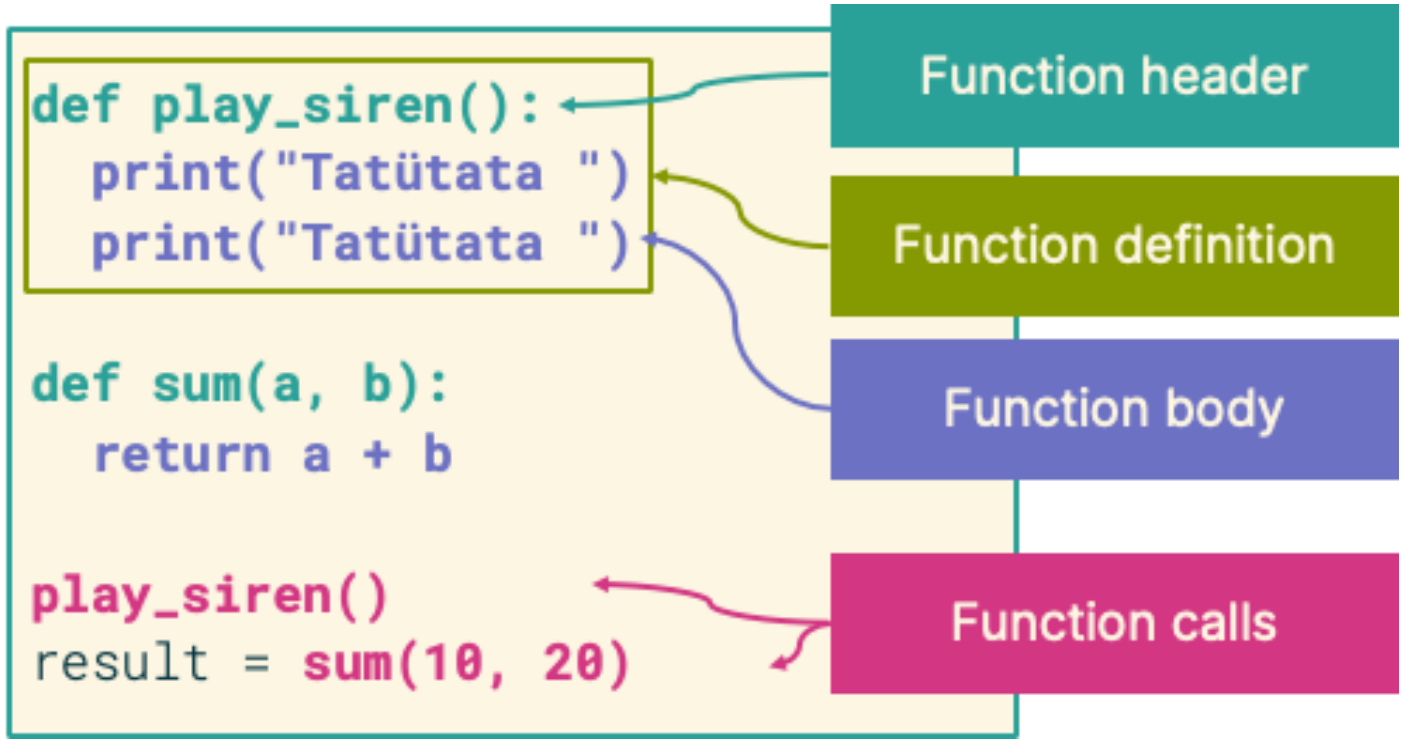
Deep dive Funktionen mit Python

Mehr Modularität in unseren Projekten

Funktionen

In den nächsten Slides werden wir uns nochmals kurz die Funktionen in Erinnerung rufen

Funktionen und Funktionsaufrufe



Funktionen und Funktionsaufrufe

```python
# 12-play-siren.py

def play_siren():
    print("Tatütata")
    print("Tatütata")


def sum(a, b):
    return a + b


play_siren()
result = sum(10, 20)
```

Variable Anzahl von Parametern

Bisher wird darauf geachtet, dass die Anzahl der Parameter einer Funktion bei Definition und Aufruf übereinstimmt.

Sie können aber auch Funktionen mit einer variablen Anzahl von Parametern definieren.

Variable Anzahl von Parametern

Bei der Definition einer solchen Funktion notieren Sie vor dem letzten (gegebenenfalls einzigen) Parameter das Zeichen *.

Dieser Parameter enthält ein **`Tupel`** mit den bis dahin nicht zugeordneten Werten der Parameterkette.

Variable Anzahl von Parametern

Wir erinnern uns nochmals kurz an den logischen Fehler des letzten Kapitels. Diesem möchten wir nun genauer auf den Grund gehen.

```python
# 16-multi-params.py

def sumfunc(*params):
    result = 0
    for s in params:
        result += s
    return result

print("Sum:", sumfunc(3, 4))
print("Sum:", sumfunc(3, 8, 12, -5))
```

Optionale Parameter

Optionale Parameter ermöglichen ebenfalls eine variable Parameterzahl. Sie müssen einen Vorgabewert besitzen und am Ende der Parameterliste stehen. Zusätzlich können benannte Parameter eingesetzt werden.

```
1  # 16-opt-params.py
2
3  def volume(width, length, depth=1, color="black"):
4      print("Values:", width, length, depth, color)
5      print("Volume:", width * length * depth, "Farbe:", color)
6
7
8  volume(4, 6, 2, "red")
9  volume(2, 12, 7)
10 volume(5, 8)
11 volume(4, 7, color="red")
```

Optionale Parameter

Die beiden Parameter **`depth`** und **`color`** sind optional und stehen am Ende der Parameterliste. Es folgen die vier Aufrufe:

- Beim ersten Aufruf werden alle vier Parameter übergeben
- Beim zweiten Aufruf wird nur der erste optionale Parameter übergeben. Der zweite optionale Parameter erhält daher den Vorgabewert
- Beim dritten Aufruf werden beide optionalen Parameter nicht übergeben und erhalten ihren Vorgabewert
- Beim vierten Aufruf wird nur der zweite optionale Parameter übergeben. Da dieser Parameter nicht positionell zugeordnet werden kann, muss er benannt werden

Kopien und Referenzen

Werden Parameter, die an eine Funktion übergeben werden, innerhalb der Funktion verändert, wirkt sich dies unterschiedlich aus:

- Bei der Übergabe eines **`einfachen Objekts (Zahl oder Zeichenkette)`** wird eine Kopie des Objekts angelegt. Eine Veränderung der Kopie **`hat keine Auswirkungen auf das Original`**.
- Bei der Übergabe eines Objekts, zum Beispiel **`des Typs Liste, Dictionary oder Set, wird mit einer Referenz auf das Originalobjekt gearbeitet`**. Eine Veränderung über die Referenz **`verändert`** auch das Original.

Kopien und Referenzen

```python
# 16-ref-and-copy.py

def change(za, tx, li, di, st):
    za = 8
    tx = "ciao "
    li[0] = 7
    di["x"] = 7
    st.discard(3)
    print(f"Function: {za} {tx} {li} {di} {st}")
hza = 3
htx = "hello"
hli = [3,"abc"]
hdi = {"x":3, "y":"abc"}
hst = set([3, "abc"])
print(f"Before:   {hza} {htx} {hli} {hdi} {hst}")
change(hza, htx, hli, hdi, hst)
print(f"After:  {hza} {htx} {hli} {hdi} {hst}")
```

Namespaces

Die Definition einer Funktion in Python erzeugt einen **`lokalen Namensraum`**. In diesem lokalen Namensraum stehen alle Namen der Variablen, denen innerhalb der Funktion ein Wert zugewiesen wird, und die Namen der Variablen aus der Parameterliste.

Rekursive Funktionen

Bestimmte Abläufe lassen sich am besten rekursiv programmieren. Eine rekursive Funktion ruft sich immer wieder selbst auf. Damit dies nicht zu einer endlosen Menge an Funktionsaufrufen führt, muss es eine Bedingung geben, die der Rekursion ein Ende setzt. Zudem wird ein erster Aufruf benötigt, der die Rekursion einleitet.

```
1 # 16-recursive.py
2
3 def make_half(value):
4     print(value)
5     value = value / 2
6     if value > 0.05:
7         make_half(value)
8 make_half(3)
```

Lambda Funktionen

Lambda-Funktionen werden mithilfe das Schlüsselworts lambda erstellt. Sie werden auch **anonyme Funktionen** genannt, im Gegensatz zu den bisher genutzten benannten Funktionen. Sie haben im Vergleich zu einer benannten Funktion eine kürzere Definition.

```
1 # 16-lambda.py
2
3 multiplication = lambda x,y: x*y
4 add = lambda x,y: x+y
5 print(multiplication(5,3))
6 print(add(4,7))
```

Übung: Eine Funktion definieren

Füllen Sie im folgenden Code die Lücken (___) so aus, dass die Funktion das Ergebnis der Multiplikation aller Parameter liefert. Geben Sie nur einen Ausdruck pro Lücke an:

```
def multiplication(a1,__,__):
  result= __* a2 * a3
  return __
```

Testen Sie Ihren Code anschliessend in PyCharm.

Lösung: Eine Funktion definieren

```
1 # 12-define-a-function
2
3 def multiplication(a1, a2, a3):
4     result = a1 * a2 * a3
5     return result
```

Ende

Das war alles für dieses Kapitel