

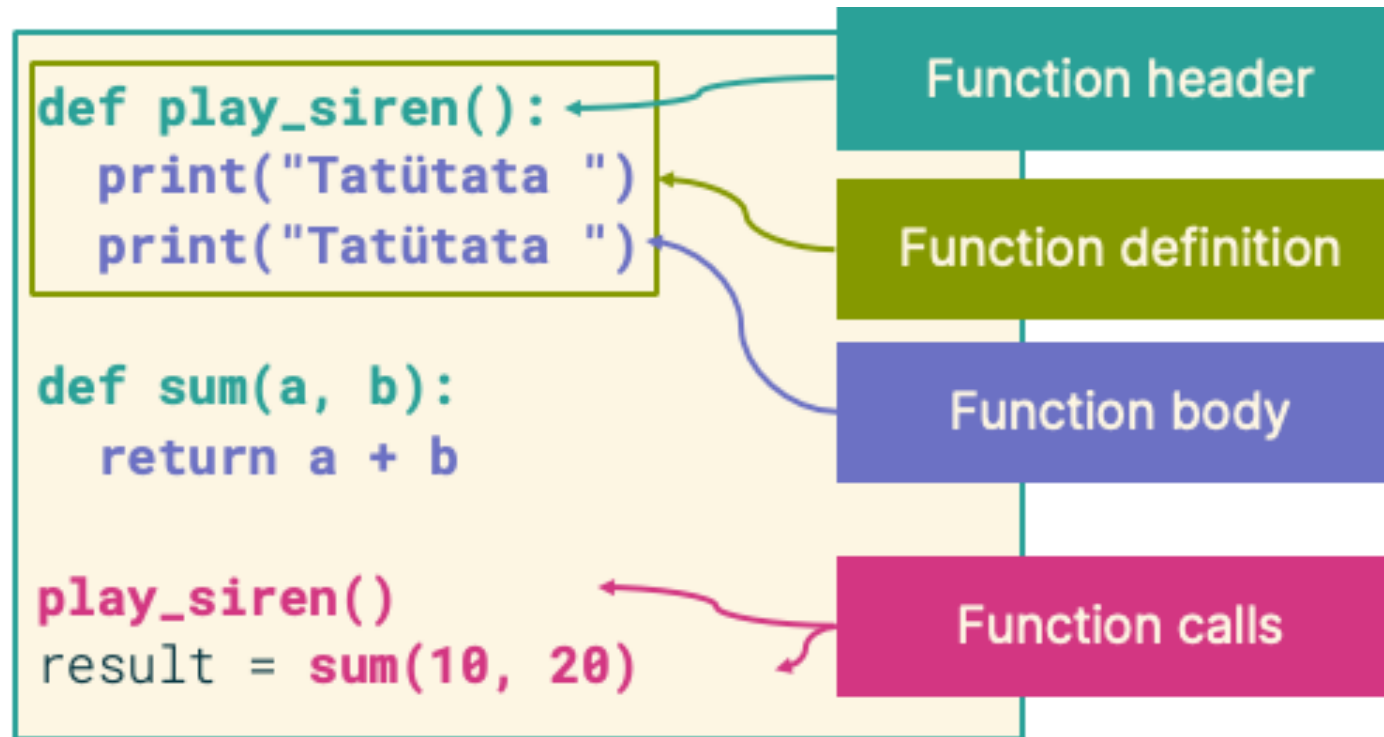
Funktionen mit Python

Mehr Modularität in unseren Projekten

Bank Management Tool

Es folgt ein kurzes Beispiel, welches wiederholte Eingaben verwendet und Bedingungen überprüft

Funktionen und Funktionsaufrufe



Funktionen und Funktionsaufrufe

```
1 # 12-play-siren.py
2
3 def play_siren():
4     print("Tatütata")
5     print("Tatütata")
6
7
8 def sum(a, b):
9     return a + b
10
11
12 play_siren()
13 result = sum(10, 20)
```

Funktionen

- Kombinieren mehrere Anweisungen in einem **Block**
- Können **Argumente** empfangen
- Kann **Werte zurückgeben**
- Ist eine Art "**Mini-Programm**"

Vorteile von Funktionen

- Programmteile, die mehrmals benötigt werden, müssen nur einmal definiert werden
- Nützliche Programmteile können in mehreren Programmen verwendet werden
- Umfangreiche Programme können in übersichtliche Teile zerlegt werden
- Pflege und Wartung von Programmen wird erleichtert

Parameter und Argumente

```
def hello(Parameterfirst_name, last_name):  
    print("Hello", first_name, last_name, "!")  
  
Arguments  
hello("Monty", "Python")
```

- **Parameter**: Variable, die in einer Funktion verwendet werden kann
- **Argument**: Wert, der dem Parameter beim Funktionsaufruf zugewiesen wird

Beim Aufruf der Funktion kommt es zu folgenden Parameterzuweisungen:

`first_name="Monty"` und `last_name="Python"`

Single Parameter Function



```
1 # 12-single-parameter.py
2
3 def square(x):
4     q = x * x
5     print("Number:", x, "Square:", q)
6
7 square(10)
```


Multi Parameter Function



```
1 # 12-multi-parameter.py
2
3 def calculation(x, y, z):
4     result = (x + y) * z
5     print("Result:", result)
6
7
8 calculation(5, 10, 20)
```

Quiz: Ordnung der Parameter

```
1 # 12-multi-parameter.py
2
3 def calculation(x, y, z):
4     result = (x + y) * z
5     print("Result:", result)
6
7
8 calculation(5, 10, 20)
```

Was denken Sie, was würde passieren, wenn man die Zahlen auf Linie 8 **vertauschen** würde? Ist die **Ordnung** wichtig? **Ja** oder **Nein**?

Keyword Argument

```
1 # 12-keyword-arguments.py
2
3 def hello(first_name, last_name):
4     print("Hello", first_name, last_name)
5
6 hello(last_name="Python", first_name="Monty")
```

Mittels **Keyword Argument** ist es nicht nötig die Ordnung der Argumente / Parameter einzuhalten

Funktionen mit Rückgabewerten



```
1 # 12-sum-product-division.py
2
3 def smd(a, b):
4     return a+b, a*b, a/b
5
6 sum, product, division = smd(10, 20)
```

Tipp: Versuchen Sie bestmöglich, ihre Funktionen so zu gestalten, dass sie nur **jeweils für eine Sache** zuständig ist

Funktionen mit Rückgabewerten

Die Anzahl der Rückgabewerte ist ihnen überlassen. Oft dient es jedoch der Übersichtlichkeit, sich auf einen Rückgabewert zu beschränken.

Funktionen mit Rückgabewerten

Die Anzahl der Rückgabewerte ist ihnen überlassen. Oft dient es jedoch der Übersichtlichkeit, sich auf einen Rückgabewert zu beschränken.



```
1 def s(a, b):  
2     return a+b  
3  
4 sum = s(10, 20)
```



```
1 def sm(a, b):  
2     return a+b, a*b  
3  
4 sum, product = smd(10, 20)
```



```
1 def smd(a, b):  
2     return a+b, a*b, a/b  
3  
4 sum, product, division = smd(10, 20)
```

Document Functions (Docstring)

DocStrings bieten eine ideale Möglichkeit, um Funktionen oder Klassen zu beschreiben. DocStrings können mehrzeilig sein.

```
1 # 12-docstrings.py
2
3 def sumfunc(a, b):
4     """Calculates the sum of a and b"""
5     return a + b
6
7 help(sum)
```

Document Functions (Docstring)

Doch DocStrings haben zusätzliche Funktionalitäten – die bei normalen Strings (Zeichenketten) nicht vorhanden sind.

Eine dieser Funktionalitäten ist der magischen Werte `__doc__`, diese beinhaltet den DocString vom Anfang der Funktion oder auch Klasse.

```
1 # 12-docstrings.py
2
3 def sumfunc(a, b):
4     """Calculates the sum of a and b"""
5     return a + b
6
7 help(sum)
```


Geltungsbereich (Function-Scope)

Der Geltungsbereich (Function-Scope) bestimmt die Sichtbarkeit/Verwendbarkeit einer Variablen innerhalb eines Blocks

Funktionen haben nur Zugriff auf Variablen*, die als Parameter übergeben werden, in der Funktion selbst definiert sind oder global sind

```
1 a, b = 40, 50
2 def sum(a, b):
3     return a + b
4
5 print(sum(10, 20))
6 print("a =", a)
7 print("b =", b)
```

Quiz: Anzeige

Was wird auf dem Bildschirm angezeigt, wenn der folgende Code ausgeführt wird?



```
1 def sumf(a, b):  
2     return a + b  
3  
4 result = sumf(10, 20)  
5 print(result)
```

Quiz: Anzeige

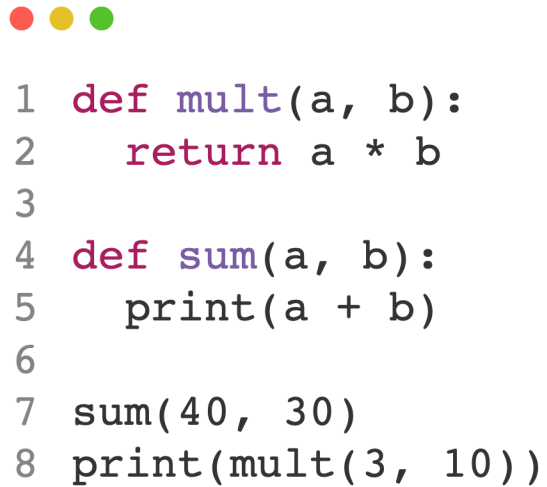
Was wird auf dem Bildschirm angezeigt, wenn der folgende Code ausgeführt wird?



```
1 def sumf(a, b):  
2     return a + b  
3  
4 result = sumfunc(10, 20)  
5 print(result)
```

Quiz: Anzeige


Was wird auf dem Bildschirm angezeigt, wenn der folgende Code ausgeführt wird?

A code block with a light gray background and rounded corners, featuring three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in a monospaced font with syntax highlighting: keywords are in purple, function names in blue, and other tokens in black.

```
1 def mult(a, b):  
2     return a * b  
3  
4 def sum(a, b):  
5     print(a + b)  
6  
7 sum(40, 30)  
8 print(mult(3, 10))
```

Quiz: Anzeige (Scope)

Was wird auf dem Bildschirm angezeigt, wenn der folgende Code ausgeführt wird?



```
1 a, b, c = 30, 20, 10
2
3 def sum(a,b,c):
4     a = 10
5     return a + b + c
6
7 print(sum(a, b, c))
8 print(a)
```

Parametertyp

Um den Typ eines Parameters zu bestimmen kann ganz einfach `isinstance` verwendet werden:

```
1 # 12-isinstance.py
2
3 def sum(a, b):
4     if not isinstance(a, float):
5         raise TypeError("Error: a is not a number")
6
7     if not isinstance(b, float):
8         raise TypeError("Error: b is not a number")
9
10    return a + b
```

Parametertyp

Um den Typ eines Parameters zu bestimmen kann ganz einfach `isinstance` verwendet werden:

```
1 x = 10
2 y = 10.10
3 z = "Monty"
4
5 print(type(x)) # <class 'int'>
6 print(type(y)) # <class 'float'>
7 print(type(z)) # <class 'str'>
```

Type Hints

Wichtig: Python erzwingt keine Type Hints, aber sie erhöhen die Lesbarkeit

Type Checker, IDEs und Linter können Type Hints nutzen, um Fehler frühzeitig zu erkennen

```
1 # 12-type-hints
2
3 def sum(a: float, b: float) -> float:
4     if not (isinstance(a, float) and isinstance(b, float)):
5         raise TypeError("Error: a or b is not a number")
6
7     return a + b
```


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
