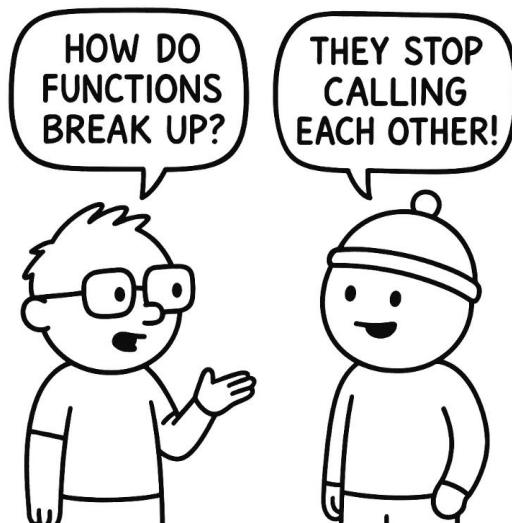


Lösungsalgorithmen & Programmieren

- Block 05 /Abend 05-

Funktionen



🎯 Leistungs-/Lernziele dieses Blocks:

Die Studierenden ...

kennen den Aufbau und die Syntax von Funktionen in Python (def, Parameterliste, Rumpf, return) und können einfache Funktionen selbst definieren.
können gegebenen Programmcode in Funktionen mit Parameterübergabe und Rückgabewert auslagern, um Duplikate zu vermeiden und Programme zu strukturieren
können Programme mit Hilfe von Funktionen übersichtlicher gestalten und wiederverwendbaren Programmcode erstellen (Modularisierung).
verstehen den Unterschied zwischen lokalen und globalen Variablen, können globale Variablen gezielt nutzen und kennen die Risiken übermässiger Global-Usage.
kennen die Parameterübergabearten „call by value“ und „call by reference“ aus anderen Sprachen und können erklären, wie sich Pythons Call-by-Object-Reference / Call-by-Sharing davon unterscheidet.
können Funktionen mit Parametern, optionalen Parametern und variabler Parameterliste (*args) schreiben und korrekt aufrufen.
können Funktionen mit einem Rückgabewert formulieren und den Rückgabewert sinnvoll weiterverwenden (Zuweisung, verschachtelte Aufrufe, Ausdrücke).
kennen die grundlegenden Konzepte Dokstring/Codedokumentation, anonyme Funktionen (lambda) und Closures (Funktionen in Funktionen) und können einfache Beispiele nachvollziehen.

Inhaltsverzeichnis:

1	Übersicht über Subroutinen	2
1.1	Subroutinen ohne Parameter und ohne Rückgabewert	4
1.2	Subroutinen mit Parameter aber ohne Rückgabewert	6
1.3	Subroutinen mit Parameter und Rückgabewert	7
1.4	Wiederholungsfragen:	11

1 Übersicht über Subroutinen

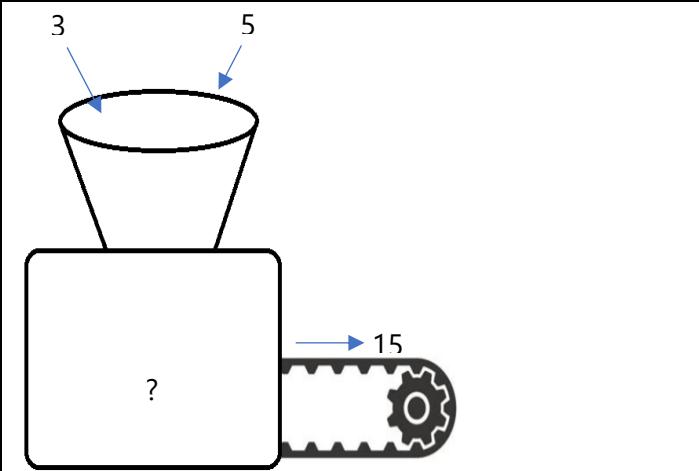
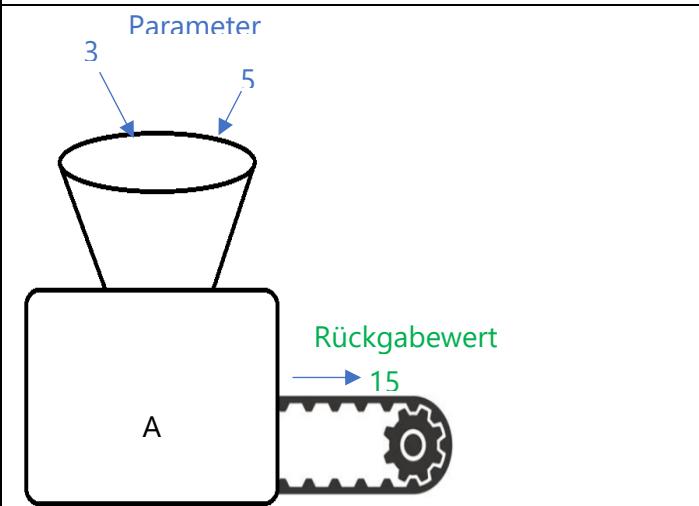
Bei Subroutinen werden verschiedene Begriffe verwendet, welche wiederum über Synonyme verfügen.

Für Subroutinen: Funktion, Unterprogramm, Methode, Prozedur oder Routine sind einige Synonyme für eine Subroutine.

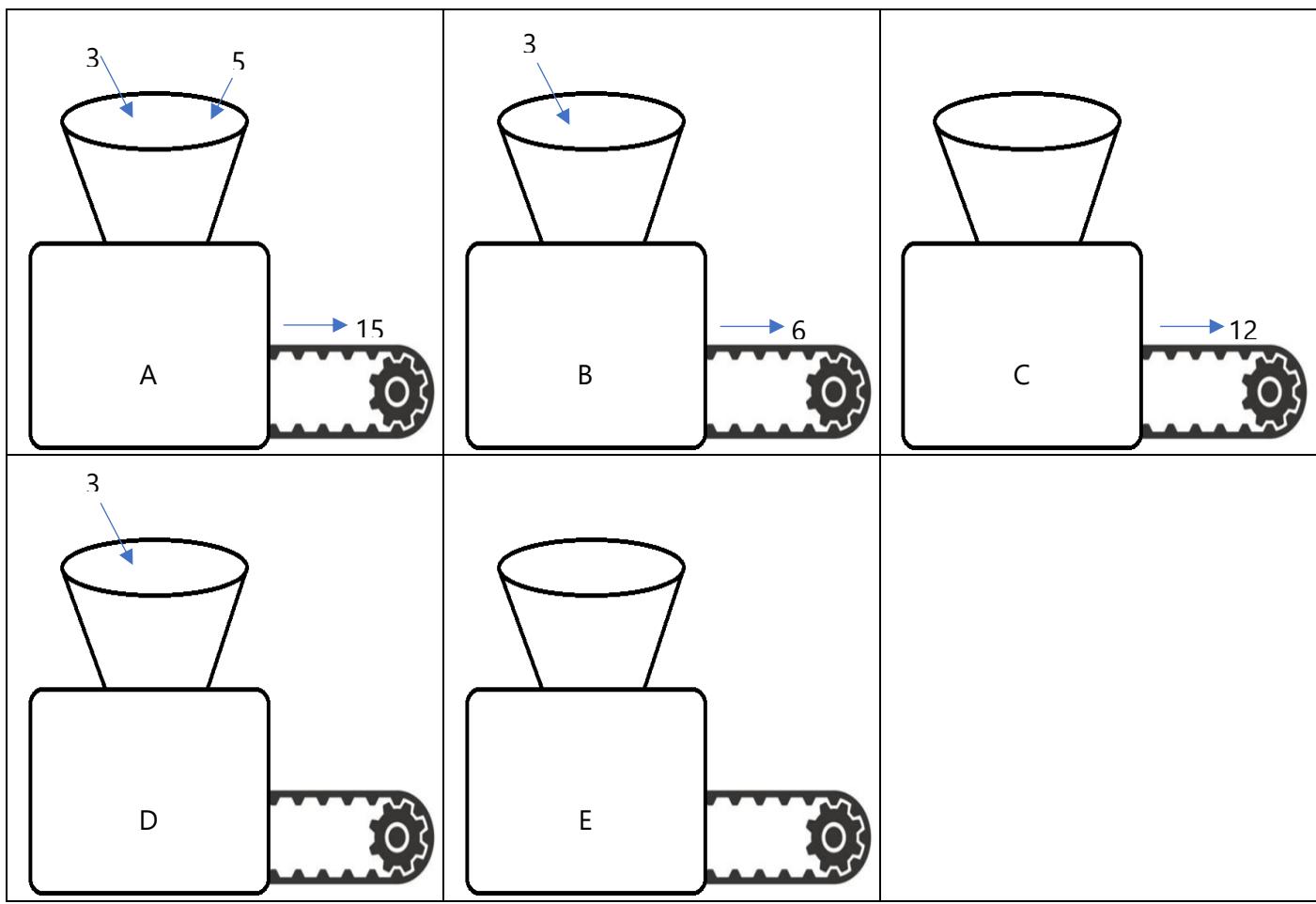
Für Parameter: Argument, Eingabewert, Übergabewert, Übergabeparameter

Für Rückgabewert: Returnwert, Funktionswert, Rückgabeparameter

Um die verschiedenen Arten und den Grundaufbau von Subroutinen zu erklären, werden wir uns einer Metapher bedienen.

	<p>Man kann schnell erkennen, was diese «Maschine» macht. Diese Maschine multipliziert zwei Zahlen. Man muss der Maschine zwei Zahlen übergeben und die Maschine gibt das Produkt aus.</p>
	<p>Die Maschine entspricht einem Unterprogramm. Sie hat eine bestimmte Funktionalität, die von einem Softwareentwickler programmiert wurde. Wenn eine solche Maschine / eine solche Subroutine zwingend eine Übergabe von irgendwelchen Werten benötigt, um überhaupt funktionieren zu können, nennen wir dies Parameter. Wenn eine solche Maschine / eine solche Subroutine einen errechneten / ermittelten Wert zurückgibt, nennen wir dies Rückgabewert.</p>

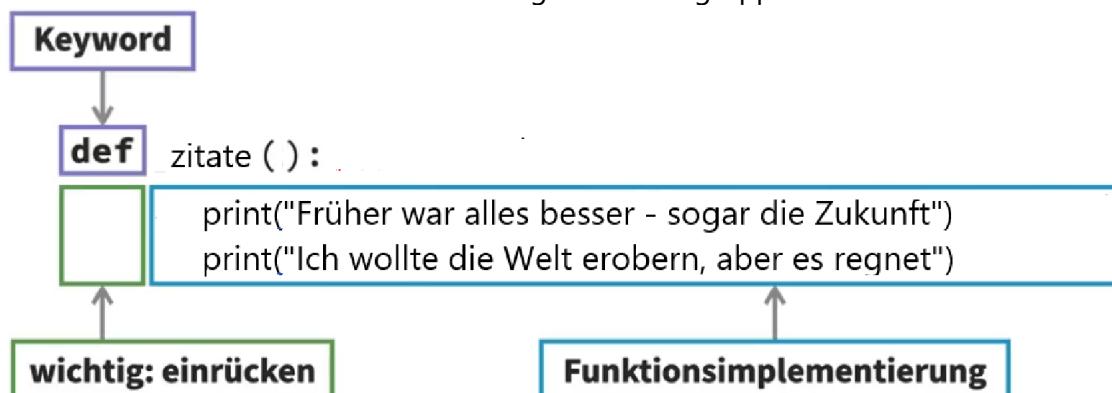
Nun kann man sich vorstellen, dass solche Maschinen ganz unterschiedlich aufgebaut sein können. Die einen verlangen z.B. zwei Parameter, andere nur einen oder sogar gar keinen Parameter. Die Maschine C im untenstehenden Beispiel könnte z.B. immer den aktuellen Monat ausgeben (12 = Dezember). Da diese Subroutine die aktuelle Systemzeit abfragt kann, benötigt es keine Parameter. Auch kann es Maschinen / Subroutinen geben, die einen Parameter verlangen, aber keinen Wert zurückgeben (Maschine D) oder auch weder Parameter verlangen noch einen Rückgabewert liefern (Maschine E).

**Wichtig!**

Unter Rückgabewert versteht man nicht eine Ausgabe an einen User (z.B. mit `print()`). Was ein Rückgabewert ist, wird später genauer beleuchtet.

1.1 Subroutinen ohne Parameter und ohne Rückgabewert

Mit dem Schlüsselwort def kann man Programmcode gruppieren und benennen:



<pre> def zitate(): print("Früher war alles besser - sogar die Zukunft") print("Ich wollte die Welt erobern, aber es regnet") print("Ich habe keine Macken, das sind Special Effects") zitate() zitate() </pre>	<p>Ausgabe:</p> <pre> 42 42 </pre>
--	--------------------------------------

Der zweimalige Aufruf der Subroutine (Funktion) zitate() liefert:

```

Früher war alles besser - sogar die Zukunft
Ich wollte die Welt erobern, aber es regnet
Ich habe keine Macken, das sind Special Effects
Früher war alles besser - sogar die Zukunft
Ich wollte die Welt erobern, aber es regnet
Ich habe keine Macken, das sind Special Effects
PS C:\Users\rolan\OneDrive - sluz\Unterricht SJ25-26\07 TEKO\03 Projekte\Block 01\03 Code &
  
```

Im folgenden Quellcode haben wir eine Variable a die ausserhalb der Funktion deklariert wurde. Sie ist damit eine globale Variable. Kann eine Funktion direkt auf die globale Variable zugreifen?

Der folgende Programmcode soll dies prüfen: Die Ausgabe zweimal die 42 überrascht uns nicht.

<pre> a=42 def ausgabe(): print (a) ausgabe() print(a) </pre>	<p>Ausgabe:</p> <pre> 42 42 </pre>
--	--------------------------------------

Wenn wir hier kompletten Zugriff auf die globale Variable hätten, dann müsste der nachfolgende Programmcode funktionieren, auch wenn wir lesend und schreibend auf die globale Variable zugreifen.

Tippe den Programmcode ab und führe ihn aus. Was passiert?

<pre> a=42 def ausgabe(): print (a) a += 1 ausgabe() print(a) </pre>	<p>Ausgabe:</p> <p>Ausnahmebehandlung UnboundLocalError</p>
---	--

Es erfolgt eine Fehlermeldung, die besagt, dass wir nicht auf eine lokale Variable zugreifen können, ohne dass ihr vorab ein Wert zugewiesen wurde.

UnboundLocalError: cannot access local variable 'a' where it is not associated with a value

Unsere Programmcodezeile `a += 1` greift also auf eine lokale Variable "a", Das ist somit nicht unsere globale Variable a.

In unserer Funktion steht also bei der "print()"-Ausgabe bei einem lesenden Zugriff zwar der Wert, der globale Variable zur Verfügung, aber nicht die Variable selbst!

Wir deklarieren nun eine "lokale" Variable auf der ersten Zeile der Funktion. Was passiert nun?

```
a=42  
def ausgabe():  
    a = 1  
    print(a)  
    a += 1  
  
ausgabe()  
print(a)
```

Ausgabe:
1
42

Es gibt keine Ausnahmebehandlung mehr. Durch die Deklaration einer lokalen Variable a, wird die globale Variable a verdeckt.

Fazit: Wenn wir in einer Funktion auf eine globale Variable zugreifen möchten, ist das nur im Lesemodus (... = a) und nicht im Schreibmodus (a= ...) möglich.

Mit dem Schlüsselwort `global` bekommen wir aber kompletten Zugriff auf die globale Variable (lesend und schreibend):

```
a=42  
def ausgabe():  
    global a  
    a += 1  
    print(a)  
  
ausgabe()  
print(a)
```

Ausgabe:
43
43

Wir haben nun die Möglichkeit, mit Funktionen unseren Programmcode modular und damit wiederverwendbar zu machen. Was stört uns aber diesbezüglich? Warum sollte man möglichst auf die Verwendung des Schlüsselwortes «`global`» verzichten?

1.2 Subroutinen mit Parameter aber ohne Rückgabewert

Wir schreiben ein Programm, das zwei Parameter a und b hat. Die Parameter a und b werden zu lokalen Variablen während der Ausführung der Funktion. Warum nennt man sie Parameter?

Weil man den Variablen Startwerte mitteilen kann. Wenn ich die Funktion z.B: mit den Werten multi(4,5) aufrufe, geht der erste Wert (4) als Inhalt/Startwert in den ersten Parameter (a), der zweite Wert (5) dient als Inhalt des zweiten Parameter, etc.

```
def multi(a, b):
    print(a * b)

multi(4, 5)
var1 = 2
multi(var1, 5)
multi(5, "abc")
```

Ausgabe:

```
20
10
abcabcabcabcabc
```

Wenn du die Funktion multi() mit den Parametern 2.5 und «abc» aufrufst, erfolgt eine Ausnahmebehandlung: «*TypeError: can't multiply sequence by non-int of type 'float'*» Warum?

Betrachte den Programmcode.

```
def multi(a, b):
    a+=1
    print(a * b)

multi(4, 5)
var1 = 2
multi(var1, 5)
print (var1)
```

Ausgabe:

```
25
15
2
```

Interessant ist die dritte Ausgabe (2). Hättest du damit gerechnet, dass 3 ausgegeben wird?

Der Grund ist, dass die Parametervariable a eine autonome Kopie der Variablen var1 ist. Lediglich wird der Inhalt als Startwert übernommen. Sonst gibt es keine Verbindung der Parametervariablen a zur Variablen var1.

Man unterscheidet in der Regel zwischen zwei Arten von Parameterübergabe:

- CallByValue (Wertübergabe, der Inhalt der Variablen wird übergeben)
oder
- CallByReference (Referenzübergabe, die Adresse der Variablen wird übergeben)

Bei CallByValue arbeiten wir mit einer autonomen Kopie der Variablen, die lediglich den Inhalt der Variablen als Startwert erhält. Änderungen gehen nach dem Unterprogrammaufruf also verloren.

Bei CallByReference arbeiten wir mit denselben Variablen, Änderungen bleiben somit auch nach dem Unterprogrammaufruf erhalten.

Bei Python ist das etwas speziell, denn Python hat weder CallByValue, noch CallByObjectRefrence. Python verwendet das Prinzip CallByObjectRefrence, auch CallBySharing genannt. Vereinfacht haben wir bei Python ein CallByRefrence, dass sich wie CallByValue verhält.

1.3 Subroutinen mit Parameter und Rückgabewert

```
def multi(a, b):
    return a*b

multi(4, 5)
```

Ausgabe:
[redacted]

Der ganze Unterprogrammaufruf wird ersetzt durch den Rückgabewert.
Das heisst an die Stelle von `multi(4, 5)` kommt 20. Wenn wir damit nichts machen, verpufft es ins Nirvana.

```
def multi(a, b):
    return a*b

print(multi(4, 5))
x=multi(4,5)
print(x)
```

Ausgabe:
20
20

unreachable Code:

```
def multi(a, b):
    return a*b
    print ("Das steht hinter der Sprunganweisung")

print(multi(4, 5))
```

Ausgabe:
20

Die Printanweisung ist unreachable Code (unerreichbarer Code), da `return` eine Sprunganweisung ist. Python generiert in solchen Fällen kein Fehler und keine Warnung.

Man kann auch nur `return` (ohne einen Wert) zurückgeben. Wie bei dem `Break` bei Schleifen wird die Funktion verlassen, aber nichts zurückgegeben.

```
def multi(a, b):
    return
    print ("Das steht hinter der Sprunganweisung")

print(multi(4, 5))
```

Ausgabe:
None

Was passiert, wenn man ein Funktion aufruft, bevor sie deklariert wurde?

```
print(multi(4, 5))

def multi(a, b):
    return a*b
```

Ausgabe:
Fehlermeldung:
`NameError: name 'multi' is not defined`

Was passiert, wenn die Anzahl der Parameter beim Aufrufen der Funktion nicht exakt übereinstimmen?

```
def multi(a, b):
    return a*b

x= multi(5)
print(x)

y= multi(2, 5, 8)
```

Ausgabe:
Fehlermeldung:
`TypeError: multi() missing 1 required positional argument: 'b'`

`TypeError: multi() takes 2 positional arguments but 3 were given`

Optionale Parameter:

```
def addiere(a ,b=8):  
    return a+b
```

```
x= addiere(4,9)  
print(x)  
x= addiere(4)  
print(x)
```

Ausgabe:

```
13  
12
```

Warum führt das zu einer Fehlermeldung?

```
def addiere(a=8 ,b):  
    return a+b
```

Variable Parameter:

Die folgende Funktion addiere() hat als zweiten Parameter eine Liste. Damit kann man beim Aufruf beliebig viele Parameterwerte übergeben:

```
def addiere(a, *b):  
    resultat = a  
    for i in b:  
        resultat+=i  
    return resultat
```

```
x= addiere(4, 9)  
print(x)  
x= addiere(4)  
print(x)  
x= addiere(4, 1, 3, 6, 7)  
print(x)
```

Ausgabe:

```
13  
4  
21
```

Warum führt das zur folgenden Fehlermeldung?

```
TypeError: addiere() missing 1 required keyword-only argument: 'b'
```

```
def addiere(*a, b):  
    resultat = a
```

Codedokumentation:

Mit drei Hochkomma ("") leitet man die Beschreibung eines Unterprogramms ein.

Betrachten Sie das folgende Beispiel:

```
def addiere(a, *b):
    """
    addiere(a, *b) --> Addiert die per Parameter übergebenen Werte und gibt das Resultat zurück
    Parameter:
    :param a: Eine Zahl die addiert werden soll
    :param b: Eine Liste von Zahlen addiert werden sollen
    Rückgabewert:
    :return: Die Summe aller übergebenen Werte
    """
    resultat = a
    for i in b:
        resultat+=i
    return resultat

print(addiere(5, 4, 56))
print(addiere.__doc__)
```

Ausgabe:

```
65
addiere(a, *b) --> Addiert die per Parameter übergebenen Werte und gibt das Resultat zurück
Parameter:
:param a: Eine Zahl die addiert werden soll
:param b: Eine Liste von Zahlen addiert werden sollen
Rückgabewert:
:return: Die Summe aller übergebenen Werte
```

Anonyme Funktionen (Operator lambda):

Anonyme Funktionen sind Funktionen bei deren Deklaration man keinen Bezeichner angibt. Die aber eine Funktionsreferenz auf sich selbst zurückliefert. In Python werden diese anonymen Funktionen über sogenannte Lambda-Ausdrücke umgesetzt. Dazu existiert eine Operator mit der Bezeichnung `lambda`.

Betrachten wir mal den Ausdruck: `lambda x: x*2`

Nach dem Lambdaoperator kann man einen Parameter deklarieren (hier `x`). Nach dem Doppelpunkt kommt der eigentliche Funktionskörper (hier `x*2`). Was hier steht wird auch standardmäßig als Funktionsrückgabewert zurückgegeben.

<code>lbd= lambda x: x*2</code> #lbd ist ein Zeiger auf die Funktion <code>z= lbd(21)</code> #via Funktionszeiger kann ich die Funktion aufrufen <code>print (z)</code>	Ausgabe: 42
---	---------------------------

Das kann auch komplett anonym (also ohne Funktionszeiger) erfolgen:

<code>def lanwenden(x, liste):</code> <code>for i in liste:</code> <code> print (x(i))</code> <code>lanwenden(lambda x: x*10, [2,3,5,7,11])</code>	Ausgabe: 20 30 50 70 110
--	--

Closures (Funktionen innerhalb von Funktionen)

Python gestattet die Deklaration von Funktionen innerhalb von Funktionen. Man nennt dies «Closure»

```
def rechnen(a):
```

```
    pot=3
```

```
    def innen():
```

```
        text = "Das Ergebnis ist"
```

```
        print (text, a**pot)
```

```
    innen()
```

```
rechnen(4)
```

```
rechnen(5)
```

```
rechnen(6)
```

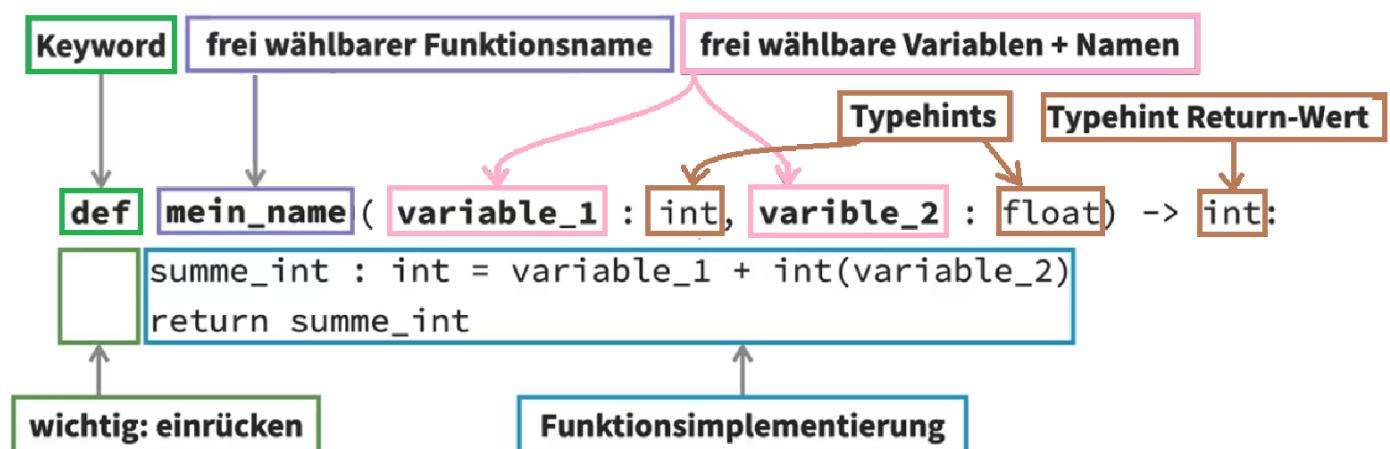
Ausgabe:

Das Ergebnis ist 64

Das Ergebnis ist 125

Das Ergebnis ist 216

Der Parameter a wird beim Aufrufen mit einem Startwert versehen (z.B. 4). Er steht dann als lokale Variable mit dem Bezeichner a zur Verfügung. Die Variable pot wird als lokale Variable innerhalb der Funktion rechnen() deklariert. Innerhalb der Funktion innen() gibt es eine lokale Variable text, die auch innerhalb von innen() zur Verfügung steht. Die innere Funktion kann aber auf die Variablen der äusseren Funktion zugreifen. Die Funktion innen() ist aber von ausserhalb nicht zugänglich. Dies kann als funktionale Alternative zur Datenkapselung (analog der objektorientierten Programmierung) verwendet werden.



1.4 Wiederholungsfragen:

01) Was ist zu beachten, wenn optionale Parameter oder Listen an eine Funktion übergeben werden?

- sie müssen immer am Ende der Parameterliste stehen
- sie dürfen entweder am Anfang oder am Ende der Parameterliste stehen, nicht aber zwischen zwei regulären Parametern
- sie müssen immer am Anfang der Parameterliste stehen
- sie müssen nach der eigentlichen Parameterliste und vor dem Doppelpunkt stehen

02) Eine verschachtelte Funktion kann die Variablen des umgebenden Gültigkeitsbereichs ____.

- weder lesen noch verändern
- gar nicht ansprechen
- lesen und verändern
- nur lesen

03) Eine globale Variable kann in einer Funktion normalerweise ____ werden.

- nur gelesen, aber nicht verändert
- gelesen und verändert
- gar nicht angesprochen
- weder gelesen noch verändert

04) Die Anzahl der Parameter beim Aufruf einer Funktion ____.

- muss kleiner oder gleich der Anzahl der deklarierten Parameter sein
- muss nicht mit der Anzahl der deklarierten Parameter übereinstimmen
- muss grösser oder gleich der Anzahl der deklarierten Parameter sein
- muss der Anzahl der deklarierten Parameter entsprechen

05) In Python wird das Ergebnis einer Funktion ____ zurückgegeben.

- mithilfe einer globalen Variablen
- über die an die Funktion übergebenen Parameter
- automatisch am Ende der Funktion
- mit der Anweisung return

06) Mit welchem Schlüsselwort definieren Sie eigene Funktionen?

- def
- func
- sub
- ret