Python Kurs 2019/2020

4: Funktionen

Bernhard Mallinger

`b.mallinger [at] gmx.at`

https://totycro.github.io/python-kurs

Funktionen

Hier muss `a` in einem Wertebereich bleiben, der Code dafür wird wiederholt:

```
MAX = 5.0
MIN = 3.0

a = [...]

if a < MIN:
     a = MIN
elif a > MAX:
    a = MAX

a = [...]

if a < MIN:
     a = MIN
elif a > MAX:
    a = MAX

print(a)
```

Funktionen

- Können selber definiert werden durch beliebigen Code
- Übernehmen Daten via Parameter/Argumente und geben Daten zurück
- Wichtige Werkzeuge im Strukturieren von Code
- Code in Funktionen kann an verschiedenen Stellen aufgerufen werden
- Soll einen Schritt in der Verarbeitung / Programmlogik darstellen
  ⇒ Geben Programmschritten einen Namen

Funktionen

```
def fun(a, b, c):
    d = a + b
    print(d, c)

print("before")
fun(1, 2, "a")
print("middle")
fun(4, 5, "b")
print("after")
```

```
# Output:
before
3 a
middle
9 b
after
```

Parameterübergabe

- Parameterübergabe kann wie Variablendefinition verstanden werden
  - rechte Seite (Werte) kommen von außerhalb der Funktion
  - linke Seite (Variablennamen) wird in Funktion definiert
- Wichtig um zu verstehen welche Änderungen sich außerhalb der Funktion auswirken

Parameterübergabe

Code mit Parameterübergabe:

```
def f(l, a):
   l.append(a)
   a = a + 1
   l.append(a)


my_list = [5]
my_number = 3

f(my_list, my_number)

print(my_list, my_number)
```

Parameterübergabe

Gleicher Code ohne Funktion:

```
my_list = [5]
my_number = 3

l = my_list
a = my_number
l.append(a)
a = a + 1
l.append(a)

print(my_list, my_number)
```

→ Bei Listen und andere Datenstrukturen wird immer auf das gleiche Objekt verwiesen, also wirken sich Änderungen außerhalb der Funktion aus
→ Basistypen wie `str` oder `int` sind hingegen unveränderlich

Funktionen

Funktionen können Werte mit `return` zurückgeben:

```
def even(number):
    return number % 2 == 0

a = 4
if even(a):
    print(a, "is even!")
```

`return` beendet Funktionsaufrufe, egal wo es steht:

```
def safe_divide(a, b):
    if b == 0:
        return 0

    return a / b
```

Funktionen

Endet eine Funktion nicht mit einem `return`, denkt sich Python automatisch ein `return None` dazu.

```
def greeter(name):
    print("Good morning", name)

result = greeter("Boku")
print(result)
```

Output:

```
Good morning Boku
None
```

Funktionen

AUFGABE

Definiere eine Funktion `trim_value`, die im Code auf der Folie 2 eingesetzt werden könnte.

Scope

Wenn eine Variable definiert wird, ist sie nur im aktuellen Scope sichtbar:

```
def fun():
    a = 3
    print(a)

fun()
print(a)
```

Output:

```
3
Traceback (most recent call last):
  File "a.py", line 6, in <module>
    print(a)
NameError: name 'a' is not defined
```

Scope

Dadurch sind Funktionen isoliert voneinander:

```
def fun():
    a = 3
    print(a)

def fun2():
    a = "hi"
    print(a)

fun()
fun2()
```

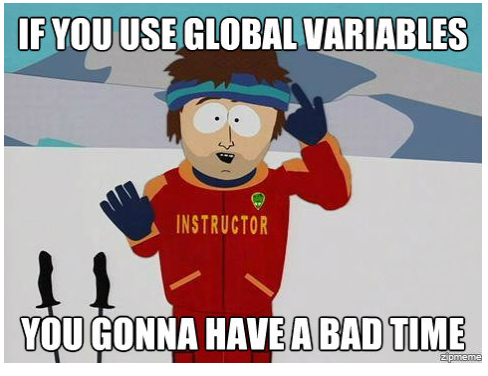
Output:

```
3
hi
```

Scope

- Zugriff auf globale Variablen ist lesend immer möglich
- Verändern ist auch möglich, aber eine schlechte Idee!

```
a = 3
def fun(b):
    return a + b
```



Scope

Namen können auch andere Namen überlagern:

```
a = 3

def f(b):
    a = b + 1
    print(a)

f(5)
print(a)
```

→ Führt eher zu Verwirrung, also vermeiden!

main

Empfehlung: Code immer nur in Funktionen, nie im globalen Scope ausführen:

```
def calculate_foo(x):
    return x + 1

def main():
    bar = 5
    foo = calculate_foo(bar)
    [...]

main()
```

→ Erleichtert Handhabung von Code (Funktionen sind isoliert)
→ Inputs für jede Funktion werden explizit übergeben
→ Keine globalen Variablen

Positional / Keyword arguments

Man kann auch den Namen von Parametern beim Aufruf angeben:

```
def f(a, b): pass

f(a=3, b="foo")
```

- Ist generell empfehlsam zur Lesbarkeit
- Ermöglicht später einfach die Reihenfolge der Parameter zu ändern, bzw. neue hinzuzufügen
- Positional und keyword arguments können gemischt werden, dürfen sich aber nicht überschneiden:

```
Traceback (most recent call last):
  File "a.py", line 2, in <module>
    f(3, a=5, b="foo")
TypeError: f() got multiple values for argument 'a'
```

Default arguments

Man kann auch Werte für Funktionsaufrufe vordefinieren, die beim Aufruf überschrieben werden können:

```
def f(a, b=3):
    return a * b

print(f(a=3))
print(f(a=3, b=2))
```

Output:

```
9
6
```

Funktionen als Variablen

Eine Funktion kann auch wie eine Variable übergeben werden:

```
def check_greater_3(x):
    return x > 3

def print_filtered(l, check):
    print([x for x in l if check(x)])

print_filtered([3, 4, 5], check=check_greater_3)
```

Output:

```
[4, 5]
```

`float` equal

AUFGABE

Implementiere die Funktion `epsilon_equals`, die überprüft, ob die Differenz zwischen zwei Fließkommazahlen höchstens `epsilon` ist. Überprüfe anhand dieser Eingaben:

```
epsilon_equals(0.4, 0.4, 1e-12) == True
epsilon_equals(0.4, 0.5, 1e-12) == False
epsilon_equals(0.400001, 0.4, 1e-6) == True
epsilon_equals(0.1 + 0.2, 0.3, 1e-10) == True
epsilon_equals(0.3, 0.1 + 0.2, 1e-10) == True
epsilon_equals(0.3000001, 0.1 + 0.2, 1e-9) == False
```

Definiere auch einen default value für `epsilon`.

Überprüfter Listenindexzugriff

AUFGABE

Implementiere die Funktion `safe_list_get`, die ein Item einer Liste an einem Index zurückgibt.

Falls dieser Index nicht im Bereich der Liste ist, gibt einen definierbaren Defaultwert zurück. Wenn dieser nicht definiert ist, soll "not found" zurückgegeben werden.

```
>>> l = [7, 8, 9]
>>> safe_list_get(l, 1, 404)
8
>>> safe_list_get(l, 99, default_value=404)
404
>>> safe_list_get(l, 99)
"not found"
>>> safe_list_get(l=l, index=99, default_value=404)
404
```

Liste von Studierenden

AUFGABE

Schreibe eine Funktion, die eine Liste von Vornamen alphabetisch sortiert ausgibt. Optional kann eine Nachricht dazu angegeben werden.

```
>>> print_students(students=["Emilia", "Maximilian", "Anna"])
3 Studierende:
* Anna
* Emilia
* Maximilian
>>> print_students(message="Auszeichnungen 2019", students=["David"])
Auszeichnungen 2019
* David
```

Wechselgeld

AUFGABE

Schreibe eine Funktion für einen Automat, der Wechselgeld berechnet: `calculate_change(price, inserted_money)`. Verwende Strings zur Beschreibung der Geldstücke und Listen zur Rückgabe.

Verwende Cent statt Euro als Geldeinheit, um Geldberechnungen mit ungenauen `float`s zu vermeiden. Beschränke dich auf Beträge unter 1 Euro.

`calculate_change(price=15, inserted_money=20)`
→ `["5 cent"]`
`calculate_change(price=16, inserted_money=50)`
→ `['20 cent', '10 cent', '2 cent', '2 cent']`
`calculate_change(price=61, inserted_money=100)`
→ `["20 cent", "10 cent", "5 cent", "2 cent", "2 cent"]`

Hinweis: Jede Münze kann höchstens 2x vorkommen.

Lohnverrechnung revisited

AUFGABE

Löse die Aufgabe "Lohnverrechnung" aus Kapitel 2 (Folie 19) mittels Funktionen.

Dabei könntest du etwa folgende Funktionen definieren:
`worked_overtime`: Gibt einen `bool` zurück, je nach dem ob Überstunden geleistet wurden.
`calculate_amount_overtime`: Anzahl der Überstunden
`calculate_amount_non_overtime`: Anzahl der regulären Stunden
`calculate_non_overtime_pay`: Lohn nur für reguläre Stunden
`calculate_overtime_pay`: Lohn nur für Überstunden

Welche dieser Funktionen sind nützlich und führen zu einem einfachen Programm?
Welche Daten sollen als Konstanten gespeichert werden und welche werden über Parameter übergeben?

Einzeiler

- `lambda` ermöglicht das Definieren von Funktionen, die aus einem Ausdruck bestehen.
- Lambda hätte "make_function" heißen sollen.
- Möglichst selten verwenden!

```
>>> third_element = lambda x: x[2]
>>> third_element([7, 8, 9, 10])
9
```