Programmieren mit Python

Teil 8: Funktionen

Dr. Aaron Kunert
aaron.kunert@salemkolleg.de

23. November 2022

Funktionen

Wie man Code wiederverwerten kann

Problemstellung

Es sei eine Liste mit Temperaturen gegeben:

```
temperatures = [22, 18, 20, 15, 12, 7, 5, -2, 4]
```

Es sollen zunächst Durchschnitte von jeweils der folgenden Grundgesamtheit genommen werden:

- Alle Werte
- Die ersten drei Werte
- Jeder zweite Wert

Statt durch eine Zahl, soll das Ergebnis jedoch mit den Worten

- `"Mild"` für Durchschnitte $\geq$ 15 Grad
- `"Zu kalt"` für Durschnitte $<$ 15 Grad

abgespeichert werden. Wie macht man das *elegant*?

Lösung (Hauptsache es funktioniert)

```
average = sum(temperatures) / len(temperatures)
if average >= 5:
  average = "Mild"
else:
  average = "Zu kalt"

average_2 = sum(temperatures[:3]) / len(temperatures[:3])
if average_2 >= 5:
  average_2 = "Mild"
else:
  average_3 = "Zu kalt"

average_3 = sum(temperatures[::2]) / len(temperatures[::2])
if average_3 < 5:
  average_3 = "Zu kalt"
else:
  average_3 = "Mild"
#
```

Nachteile dieser Lösung

- Viel Schreibarbeit, viel Wiederholung
- Der Code ist schwierig zu lesen. Man sieht vor lauter Wiederholungen nicht, was passiert.
- Jedes Mal, wenn man diese „Berechnungslogik“ verwendet, könnte man einen (Tipp-)Fehler machen.
- Wenn man das Anforderungsprofil minimal ändert, muss diese „Logik“ bei *jedem* Auftreten im Code geändert werden (z.B. statt `"Mild"` soll das Ergebnis `"Warm"` heißen). In echten Projekten, kann das schnell ein paar Hundert Male sein.

Bessere Lösung

```
def compute_average(temp_list):
  result = sum(temp_list) / len(temp_list)
  if result >= 5:
    result = "Mild"
  else:
    result = "Zu kalt"
  return result

average = compute_average(temperatures)
average_2 = compute_average(temperatures[:3])
average_3 = compute_average(temperatures[::2])
```

Definition: Funktion

Eine Funktion ist ein Codeblock, der nur ausgeführt wird, wenn die Funktion *aufgerufen* wird. Man kann der Funktion Werte als *Parameter* übergeben. Sie kann auch einen Wert als Ergebnis *zurückgeben*.

Man kann sich eine Funktion wie eine Maschine vorstellen, wo man oben Dinge (=Parameter) hineinfüllt und unten ein Ergebnis (=Rückgabewert) herausbekommt. Unabhängig von dem Eingabe-Ausgabe-Prinzip, kann solch eine Maschine auch Nebeneffekte (z.B. Krach) produzieren.

Man unterscheidet zwischen *Definition* und *Ausführung* einer Funktion.

Struktur der Funktions-Definition

`def` *Funktionsname*(*Parameter_0*, *Parameter_1*, ..., *Parameter_n*):

␣␣ *Codezeile1*

␣␣ *Codezeile2*

⋮

␣␣ `return` *Ergebnis*

Struktur eines Funktionsaufrufs

result = *Funktionsname*(*Argument_0*, *Argument_1*, ..., *Argument_n*)

Good to know

- Eine Funktion muss schon *vor* dem ersten Aufruf definiert worden sein (das ist nicht in allen Sprachen so).
- Die Eingabwerte nennt man in der Funktionsdefintion *Parameter*, beim Aufruf der Funktion nennt man sie jedoch *Argumente*.
- Nicht jede Funktion braucht Eingangsdaten. Die Liste von Parametern einer Funktion kann daher leer sein.
- Beim Aufruf spielt die Reihenfolge der angegebenen Argumente eine entscheidene Rolle. Sie werden entsprechend der Reihenfolge den Parametern in der Definition zugeordnet.
- Eine Funktion muss nicht unbedingt etwas zurückgeben, d.h. das `return`-Statement ist optional.
- Das `return`-Statement muss nicht unbedingt am Schluss der Funktion stehen. Jedoch wird Code, der nach dem `return`-Statement kommt, nicht mehr ausgeführt.

Übungen

Funktion ohne Parameter

Schreibe eine Funktion, die Deinen Namen auf der Konsole ausgibt.

Funktion mit einem Parameter

Schreibe eine Funktion, die die übergebene Zahl verdoppelt und das Ergebnis zurückgibt.

Funktion mit zwei Parametern

Schreibe eine Funktion, die die beiden übergebenen Zahlen multipliziert und das Ergebnis zurückgibt.

Funktion ohne Rückgabewert

Was gibt eine Funktion zurück, die kein `return`-Statement enthält?

Lösungen

Funktion ohne Parameter

```
def my_name():
  print("Aaron Kunert")
```

Funktion mit einem Parameter

```
def double(number):
  return number * 2
```

Funktion mit zwei Parametern

```
def multiply(number1,number2):
  return number1 * number2
```

Übung

Aggregatzustand von Wasser

Schreibe eine Funktion, die die Temperatur als Paramter erwartet und abhängig von der Temperatur den Aggregatzustand von Wasser (`"fest"`,`"flüssig"`,`"gasförmig"`) als String zurückgibt.

Zusatz: Schaffst Du es ohne die Schlüsselwörter `elif` und `else`?

Lösung

Aggregatzustand von Wasser

```
def get_state(temp):
  if temp < 0:
    return "fest"
  if temp > 100:
    return "gasförmig"
  return "flüssig"
```

Komplexere Übung

Gewichtete Durschnittsnote

Schreibe eine Funktion, die eine Liste der folgenden Struktur erwartet:

```
grades = [
  {
    "subject": "Deutsch",
    "grade": 14,
    "is_major": True
  },
  # ...
  {
    "subject": "Sport",
    "grade": 11,
    "is_major": False
  }
]
```

Berechne die Durchschnittsnote, wobei Hauptfächer doppelt gewichtet werden sollen.

Lösung

Gewichtete Durchschnittsnote

```
def weighted_average(grades):
  weighted_sum = 0
  weighted_length = 0
  for grade in grades:
    if grade["is_major"]:
      weighted_sum += 2 * grade["grade"]
      weighted_length += 2
    else:
      weighted_sum += grade["grade"]
      weighted_length += 1
  result = weighted_sum/weighted_length
  return result
```

Übung

Zinsrechner

Angenommen, Du hast 1000€ so angelegt, dass es darauf jeden Monat 0,5% Zinsen gibt (Haha – als ob!). Schreibe eine Funktion, die einen Geldbetrag erwartet und zurückgibt, nach wie vielen Monaten, dieser Geldbetrag erreicht wurde.

Lösung

Zinsrechner

```
def months_until_rich(target):
    month = 0
    balance = 1000
    while balance < target:
        balance *= 1.005
        month += 1
    return month
```

Übung

Zinsrechner mit Sparrate

Angenommen, Du hast wieder 1000€ so angelegt, dass es darauf jeden Monat 0,5% Zinsen gibt (Haha – als ob!). Zusätzlich gibt es nun jedoch noch eine monatliche Sparrate von 25€, die ebenfalls auf das Konto eingezahlt wird. Schreibe eine Funktion, die einen Geldbetrag erwartet und zurückgibt, nach wie vielen Monaten, dieser Geldbetrag erreicht wurde.

Lösung

Zinsrechner mit Sparrate

```
def months_until_rich(target):
    month = 0
    balance = 1000
    while balance < target:
        balance += 25
        balance *= 1.005
        month += 1
    return month
```

Optionale Parameter

Manchmal wirst Du bei Funktionen bemerken, dass einige der Parameter fast immer den gleichen Wert haben. In diesem Fall, möchtest Du diese Parameter nicht bei jedem Aufruf immer hinschreiben, sondern nur dort, wo er vom Standardfall abweicht. Dies ist möglich, wenn man den Standardwert (*default value*) bei der Definition mit angibt.

Wichtig: Bei der Definition müssen die optionalen Parameter immer hinter den Pflichtparametern stehen.

Beispiel

```
def double(number, factor=2):
  return number * factor
```

Diese Funktion ist sehr vielseitig: Im einfachen Fall verdoppelt sie die eingegebene Zahl. Optional lässt sich der Faktor aber beliebig verändern.

Typischer Einsatzbereich

Oftmals merkt man im Verlauf eines Projektes, dass eine gegebene Funktion nicht flexibel genug ist, dann kann man sie um einen optionalen Parameter erweitern, ohne den bisherigen Code verändern zu müssen.

Fiktives Beispiel

Stell Dir vor, Du baust einen Zinsrechner wie oben, den auch schon einige Deiner Freund*innen, die den gleichen Sparplan haben, ebenfalls verwenden. Eine Deiner Freundinnen hat jedoch eine bessere Bank gefunden, die ihr 0,6% Zinsen gibt. Also erweiterst Du die Funktion, so dass auch die Höhe der Zinsen anpassbar ist.

Jedoch möchtest Du den bisherigen Code nicht verändern. Daher definierst Du den Zinssatz als optionalen Parameter, so dass die Funktion „abwärtskompatibel“ zu ihrer bisherigen Verwendung ist.

Die Definition startet dann mit `def months_until_rich(target, interest=0.5):`

Übung

Flexibler Zinsrechner

Erweitere den Zinsrechner, so dass optional der Zinssatz angegeben werden kann.

Lösung

Flexibler Zinsrechner

```
def months_until_rich(target, interest=0.5):
    month = 0
    balance = 1000
    while balance < target:
        balance += 25
        balance *= (1 + interest/100)
        month += 1
    return month
```

Schwierige Übung

Flexibler Durchschnittsrechner

Erweitere die Funktion zur Berechnung von gewichteten Durchschnittsnoten so, dass optional der Gewichtungsfaktor angegeben werden kann.

Lösung

Flexibler Durchschnittsrechner

```
def weighted_average(grades, weight=2):
  weighted_sum = 0
  weighted_length = 0
  for grade in grades:
    if grade["is_major"]:
      weighted_sum += weight * grade["grade"]
      weighted_length += weight
    else:
      weighted_sum += grade["grade"]
      weighted_length += 1
  result = weighted_sum/weighted_length
  return result
```

Named Parameters

Hat eine Funktion viele Parameter, von denen etliche optional sind, so kann man einen Parameter statt über die Reihenfolge auch über den Namen übergeben.

Beispiel

```
def my_function(parameter1, parameter2=0, parameter3="x", parameter4=-17):
  # ...
```

Möchte man jetzt die Funktion mit einem eigenen Wert `parameter1` und `parameter4` aufrufen aber alles andere auf Standard lassen, so geht das wie folgt:

```
my_function(15, parameter4=-20)
```

Übung

Ganz flexibler Zinsrechner

Erweitere den Zinsrechner, so dass zusätzlich optional der Startbetrag und die monatliche Sparrate angepasst werden können.

Welchen der Parameter muss man verdoppeln, um am schnellsten 10.000€ zu erreichen?

Lösung

Ganz flexibler Zinsrechner

```
def months_until_rich(target, interest=0.5, initial_amount=1000, savings_rate=25):
    month = 0
    balance = initial_amount
    while balance < target:
        balance += savings_rate
        balance *= (1 + interest/100)
        month += 1
    return month
```

Scope

Wo Variablen gültig sind

Problemstellung

Sei `my_variable` eine Variable mit Wert 1. Schreibe eine Funktion, die bei Aufruf die Variable `my_variable` um 1 erhöht. Schreibe eine Funktion, die bei Aufruf die Variable `my_variable` um 1 erhöht.

Wie macht man das?

Das Problem

```
my_variable = 1

def increment():
  my_variable = my_variable + 1

increment()
print(my_variable)
```

Die offensichtliche Lösung funktioniert nicht. Warum nicht?

Experiment I

```
global_variable = 1

def my_function():
  local_variable = 5

my_function()
print(global_variable)
print(local_variable)
```

Beobachtung

Eine Variable, die innerhalb einer Funktion definiert wurde, ist auch nur innerhalb der Funktion sichtbar.

Experiment II

```
global_variable = 1

def my_function():
  print(global_variable)


my_function()
print(global_variable)
```

Beobachtung

Eine *globale* Variable ist auch innerhalb einer Funktion definiert.

Experiment III

```
global_variable = 1

def my_function():
  global_variable = 5
  print(global_variable)


my_function()
print(global_variable)
```

Beobachtung

Eine Variable innerhalb einer Funktion kann den gleichen Namen wie eine Variable außerhalb haben, allerdings ist die innere Variable nur innerhalb der Funktion sichtbar.

Experiment IV

```
global_variable = 1

def my_function():
  print(global_variable)
  global_variable = 5


my_function()
print(global_variable)
```

Beobachtung/Erklärung

Python entscheidet anhand des Kontexts ob `global_variable` eine globale Variable ist, oder eine lokale Variable, die zufällig den gleichen Namen wie eine globale Variable trägt.

Falls Python denkt, dass es sich um eine globale Variable handelt, so kann diese nur gelesen, nicht aber geschrieben (d.h. neu definiert) werden.

Das Eingangsbeispiel

```
my_variable = 1

def increment():
  my_variable = my_variable + 1

increment()
print(my_variable)
```

Erklärung

Da `my_variable` rechts vom Gleichheitszeichen steht, denkt Python, dass es sich um die globale Variable `my_variable` handelt. Da `my_variable` aber auch links vom Gleichheitszeichen steht, wird auch schreibend auf die Variable zugegriffen. Das ist nicht erlaubt.

Mögliche Lösung

```
my_variable = 1

def increment(var):
  return var + 1

my_variable = increment(my_variable)
print(my_variable)
```

Definition

Der Gültigkeitsbereich einer Variable wird *Scope* genannt.

Scope in Python

In Python unterscheidet man zwischen *global Scope* und *local Scope*. Im local Scope hat man nur Lesezugriff auf den global Scope.

Achtung Ausnahme

```
my_list = [1, 2, 3]

def append(item):
  my_list.append(item)

append(4)
print(my_list)
```

Erklärung

Da die Variable `my_list` nicht überschrieben wird, sondern nur das referenzierte Objekt verändert wird, erkennt Python dies nicht als Schreibzugriff und erlaubt dieses Vorgehen.

Warum ist der Zugriff auf den Global Scope eingeschränkt?

- Funktionen sollen möglichst wenige Nebeneffekte haben. Wenn eine Funktion den global Scope verändern kann, ist dies ein großer Nebeneffekt.
- Wenn man eine Funktion schreibt, muss man sich keine Gedanken machen, ob ein Variablenname schon vergeben ist.
- Wenn man sich innerhalb einer Funktion den Kontakt zum global Scope reduziert, so ist die Funktion besser zu verstehen, zu warten und zu testen.
- ...