

Programmieren mit Python

Teil 8: Comprehensions und Funktionen

Dr. Aaron Kunert

aaron.kunert@salemkolleg.de

2. Dezember 2021

Comprehensions

Typische Manipulationen

Sehr häufig möchte man eine Datenstruktur (d.h. eine Liste oder ein Dictionary) basierend auf den Werten manipulieren. Dabei werden vor allem zwei Aspekte immer wieder gebraucht: Maps und Filter.

Map

Ersetzt man jedes Element einer Liste durch ein aus dem ursprünglich berechnetem Element, so spricht man von einer *Map* (bzw. einem Mapping).

Beispiel

Gegeben ist die Liste `my_list = [2, 5, 3, 12, 7]`. Die Liste soll so manipuliert werden, dass alle Einträge durch ihren doppelten Wert ersetzt werden.

Traditionelle Lösung

```
my_list = [2, 5, 3, 12, 7]
result = []
for k in my_list:
    result.append(2 * k)
print(result)
```

The Pythonian Way

```
my_list = [2, 5, 3, 12, 7]
result = [2 * k for k in my_list]
print(result)
```

Filter

Streich man Elemente entsprechend ihres Wertes aus einer Liste, so spricht man von einem *Filter*.

Beispiel

Gegeben ist die Liste `my_list = [2, 5, 3, 12, 7]`. Aus der Liste sollen alle ungeraden Einträge gestrichen werden.

Traditionelle Lösung

```
my_list = [2, 5, 3, 12, 7]
result = []
for k in my_list:
    if k % 2 == 0:
        result.append(k)
print(result)
```

The Pythonian Way

```
my_list = [2, 5, 3, 12, 7]
result = [k for k in my_list if k % 2 == 0]
print(result)
```

Kombination aus Map und Filter

Selbstverständlich können Maps und Filter auch kombiniert werden.

Beispiel

Lösche alle ungeraden Zahlen und verdopple dann alle Zahlen:

```
my_list = [2, 5, 3, 12, 7]
result = [2 * k for k in my_list if k % 2 == 0]
print(result)
```

Struktur List Comprehension

```
result = [ Wunschetrug(k) for k in my_list if Bedingung(k) ]
```

Dictionary Comprehension

Man kann das gleiche Verfahren auch auf Dictionaries anwenden. Dabei können jeweils key und value für die Maps und Filter verwendet werden.

Beispiel

```
my_dict = {"a": 2, "b": 3}
result = {key: value for (key, value) in my_dict.items()}
print(result)
```

Struktur Dictionary Comprehension

```
result = { Wunsch-schlüssel(k,v) : Wunsch-Wert(k,v)  
          for (k,v) in my_dict.items() if Bedingung(k,v) }
```

List Comprehension

Gegeben sei eine beliebige Liste von ganzen Zahlen. Streiche alle Zahlen, die ungerade oder negativ sind. Ersetze die übrigen Zahlen durch ihre Hälfte.

Lösung

```
my_list = [2, -3, 40, 15, 7, 8]
result = [k//2 for k in my_list if k % 2 == 0 and k >= 0]
```

Dictionary Comprehension

Gegeben sei das Dictionary `{"Mathe": 9, "Sport": 13, "Physik": 4, "Bio": 12}`.
Lösche nun daraus alle Noten unter 5 Punkte sowie die Sportnote. Zusätzlich soll das Dictionary danach wie folgt aussehen: `{"In Mathe": "9 Punkte", "In Bio": "12 Punkte"}`.

Lösung

```
grades = {"Mathe": 9, "Sport": 13, "Physik": 4, "Bio": 12}
result = {
    "In " + key: f"{value} Punkte"
    for (key, value) in grades.items()
    if value >= 5 and key != "Sport"
}
```

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 ≥ 15 Grad
- "Zu kalt" für Durchschnitte < 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 Funktionsdefinition *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 entscheidende 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.

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?

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
```

Aggregatzustand von Wasser

Schreibe eine Funktion, die die Temperatur als Parameter 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`?

Aggregatzustand von Wasser

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

Komplexere Übung

Gewichtete Durchschnittsnote

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.

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
```

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.

Zinsrechner

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

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.

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):`

Flexibler Zinsrechner

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

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
```

Flexibler Durchschnittsrechner

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

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)
```

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?

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
```
