Programmieren mit Python

Teil 5: Dictionaries und Funktionen

Dr. Aaron Kunert
aaron.kunert@salemkolleg.de
20. Mai 2021

Dictionaries

Problemstellung

In einem Notenrechner wie eben, sollen nicht nur die Noten gespeichert werden, sondern auch das Fach, in dem die Note erreicht wurde.

Wie macht man das?

Lösung

```
# ...
grades = { "Deutsch": 14, "Mathematik": 8, "Biologie": 11, "Sport": 13}

key = input("Welche Note möchtest Du wissen?")

print(f"Deine Note in { key } ist { grades[key] } Punkte")
```

Struktur eines *Dictionaries*

`my_dict = {key_1:value_1, key_2:value_2, ..., key_n:value_n}`

Das *Dictionary* `my_dict` enthält Schlüssel-Wert-Paare (*key-value-pairs*). Die Schlüssel müssen eindeutig und unveränderlich sein (z.B. vom Typ `string` oder `int`). Die Werte dürfen beliebige Datentypen sein.

Good to know

- Zur besseren Übersichtlichkeit werden Dictionaries oftmals wie folgt formatiert:

```
grades = {
  "Deutsch": 14,
  "Mathematik": 8,
  "Biologie": 11,
  "Sport": 13
}
```

- Dictionaries sind mutable, können also verändert werden.
- Dictionaries besitzen keine vernünftige Anordnung und können nicht geordnet werden.
- Ein Dictionary kann leer sein.

- Oftmals bietet es sich an, statt einem Dictionary eine Liste von Dictionaries zu verwenden:

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

Auf Dictionary-Elemente zugreifen

Sei `my_dict = {"a": 5, "b": 8}`.

Mit der Syntax `my_dict["a"]` kann man den Wert an der Stelle `"a"` auslesen.

Mit der Syntax `my_dict["a"] = 12` kann man einzelne Werte des Dictionaries verändern.

Auf diese Weise können auch ganz neue Paare hinzugefügt werden. Zum Beispiel:
`my_dict["c"] = -2`.

Übung

Dictionary manipulieren

Gegeben sei das folgende Dictionary:

```
grades = { "Mathe": 8, "Bio": 11, "Sport": 13}
```

Bestimme die Durchschnittsnote dieser drei Fächer. Verbessere danach Deine Mathenote um einen Punkt und füge noch eine weitere Note für Englisch hinzu (Abfrage über Konsole). Gib danach erneut den Durchschnitt an.

Dictionary manipulieren

```
grades = { "Mathe": 8, "Bio": 11, "Sport": 13}

grades_sum = grades["Mathe"] + grades["Bio"] + grades["Sport"]
average = grades_sum/len(grades)
print(f"Der Durchschnitt ist {average} Punkte")

grades["Mathe"] += 1

eng_grade = input("Welche Note hast Du in Englisch? ")
eng_grade = int(eng_grade)
grades["Englisch"] = eng_grade

grades_sum += grades["Englisch"]
average = grades_sum/len(grades)
print(f"Der Durchschnitt ist {average} Punkte")
```

Einen Eintrag aus einem Dictionary entfernen

Wie bei Listen, kann man mittels `del`-Statement einen Eintrag aus einem Dictionary entfernen:

```
del my_dict["a"]
```

Was wird hier passieren?

```
grades = { "Mathe": 8, "Bio": 11, "Sport": 13}
new_grades = grades

new_grades["Mathe"] = 15

print(grades)
print(new_grades)
```

Erklärung

Intern befindet sich in der Variable `grades` nur ein Verweis auf das Dictionary im Speicher. Die Variable `new_grades` enthält den gleichen Speicherverweis. Das heißt, wenn man das Dictionary `new_grades` verändert, verändert sich auch das Dictionary `grades`, weil es sich um ein-und dasselbe Dictionary handelt.

Eine Kopie von einem Dictionary erstellen

Mit der Funktion `dict()` kann man eine Kopie von einem Dictionary erstellen.

Beispiel: `dict(my_dict)` erstellt eine Kopie von `my_dict`.

Schleife über Dictionary I

Ähnlich wie bei Listen kann man Schleifen auch über ein Dictionary laufen lassen.

Beispiel

```
grades = { "Mathe": 8, "Bio": 11, "Sport": 13}

for item in grades:
  print(item)

# Mathe
# Bio
# Sport
```

Schleife über Dictionary II

Möchte man in der Schleife nicht nur die Schlüssel, sondern auch die Werte des Dictionaries zur Verfügung haben, so muss man die Methode `.items()` auf das Dictionary anwenden.

Beispiel

```
grades = { "Mathe": 8, "Bio": 11, "Sport": 13}

for key, value in grades.items():
  print(f"{key}:{value}")

# Mathe:12
# Bio:11
# Sport:13
```

Übungen

Zwei Dictionaries kombinieren

Gegeben seien zwei Dictionaries, z.B.

```
majors = {"Deutsch": 11, "Mathe": 7}
```

und

```
minors = {"Sport": 14, "Bio": 8 }
```

Füge die Einträge des zweiten Dictionaries zum ersten Dictionary hinzu.

Ein Dictionary „filtern“

Sei ein beliebiges Dictionary mit Noten gegeben. Entferne alle Einträge, deren Note schlechter als 5 Punkte ist.

Lösung

Zwei Dictionaries kombinieren

```
majors = {"Deutsch": 11, "Mathe": 7}
minors = {"Sport": 14, "Bio": 8}

for key,value in minors.items():
  majors[key] = value
print(majors)
```

Ein Dictionary „filtern“

```
grades = {"Deutsch": 11, "Mathe": 3, "Sport": 14, "Geschichte": 1}
#  Man darf die Länge eines Dictionaries in einer Schleife nicht verändern, deshalb machen wir eine
result = dict(grades)
for key, value in grades.items():
  if value < 5:
    del result[key]
print(result)
```

Ein Dictionary zerlegen

Mit der Methode `.keys()` erhält man eine Liste aller Schlüssel eines Dictionaries.

Mit der Methode `.values()` erhält man eine Liste aller Werte eines Dictionaries.

In beiden Fällen, muss das Ergebnis mittels der Funktion `list()` in eine Liste umgewandelt werden.

Beispiel

```
grades = { "Mathe": 8, "Bio": 11, "Sport": 13}

grade_subjects = grades.keys()
grade_subjects = list(grade_subjects)

grade_numbers = grades.values()
grade_numbers = list(grade_numbers)

print(grade_subjects)   # ["Mathe", "Bio", "Sport"]
print(grade_numbers)    # [8, 11, 13]
```

Funktionen

Wie man Code wiederverwerten kann

Problemstellung

Es sei eine Liste mit Noten gegeben:

```
grades = [7, 12, 8, 10, 2, 0, 3, 5, 6]
```

Es sollen zunächst folgende Durchschnittsnoten berechnet werden:

- Durchschnitt aller Noten
- Der Durchschnitt der ersten drei Noten
- Der Durchschnitt jeder zweiten Note

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

- `"Passt"` für Durchschnitte ≥ 5 Punkte
- `"Durchgefallen"` für Durschnitte < 5 Punkte

abgespeichert werden.

Wie macht man das *elegant*?

Lösung (Hauptsache es funktioniert)

```
average = sum(grades) / len(grades)
if average >= 5:
  average = "Passt"
else:
  average = "Durchgefallen"

average_2 = sum(grades[:3]) / len(grades[:3])
if average_2 >= 5:
  average_2 = "Passt"
else:
  average_3 = "Durchgefallen"

average_3 = sum(grades[::2]) / len(grades[::2])
if average_3 < 5:
  average_3 = "Durchgefallen"
else:
  average_3 = "Passt"
#
```

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 `"Passt"` soll das Ergebnis `"Bestanden"` heißen). In echten Projekten, kann das schnell ein paar Hundert Male sein.

Bessere Lösung

```
def compute_average(grade_list):
  result = sum(grade_list) / len(grade_list)
  if result >= 5:
    result = "Passt"
  else:
    result = "Durchgefallen"
  return result

average = compute_average(grades)
average_2 = compute_average(grades[:3])
average_3 = compute_average(grades[::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.

Funktion mit zwei Parametern

Schreibe eine Funktion, die die beiden übergebenen Zahlen multipliziert.

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 entsprechend der übergebenen Temperatur den Aggregatzustand von Wasser (`"fest"`,`"flüssig"`,`"gasförmig"`) als String zurückgibt.

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
```

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 Rechner für Deine Endnote. Hauptfachnoten werden immer doppelt gewichtet, daher verwendest Du die Funktion `weighted_average`, wie in der Übung. Plötzlich kommt raus, dass in der Abschlussprüfung, Hauptfächer vierfach gewichtet werden. Also erweiterst Du die Funktion, so dass der Gewichtungsfaktor anpassbar ist.

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

Die Definition startet dann mit `def weighted_average(grades, weight=2):`

Übung

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)
```

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.

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

```
outer_variable = 1

def my_function():
  inner_variable = 5

my_function()
print(outer_variable)
print(inner_variable)
```

Beobachtung

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

Experiment II

```
outer_variable = 1

def my_function():
  print(outer_variable)


my_function()
print(outer_variable)
```

Beobachtung

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

Experiment III

```
outer_variable = 1

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


my_function()
print(outer_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

```
outer_variable = 1

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


my_function()
print(outer_variable)
```

Beobachtung/Erklärung

Python entscheidet anhand des Kontexts ob `outer_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.

Warum ist das so 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.
- ...