

Programmierung mit Python für Einsteiger

Kapitel 3 - Schleifen

Autor: Dr. Christian Heckler

Fallbeispiel: Durchschnittsverbrauch

- Es soll ein Programm geschrieben werden, bei dem der Anwender beliebig viele Verbrauchsdaten eingeben kann.
- Es soll der Durchschnitt ermittelt werden.
- Das Programm soll beendet werden, wenn der Anwender `ende` eingibt.
- Algorithmus in Umgangssprache („Pseudocode“):

```
summe = 0
anzahl = 0
Gebe eine Zahl oder `ende` ein
so lange eine Zahl eingegeben wird:
    summe = summe + eingegebene zahl
    anzahl = anzahl + 1
    gebe eine Zahl oder `ende` ein
durchschnitt = summe / anzahl
```

Schleife

- Der Codeblock

```
summe = summe + eingegebene zahl  
anzahl = anzahl + 1  
gebe eine Zahl oder `ende` ein
```

wird also mehrfach durchlaufen.

- Das nennt man **Schleife**.
- Das Programm in Python:

```
summe = 0  
anzahl = 0  
verbrauch = input("Geben Sie einen Verbrauch ein (oder ende): ")  
while verbrauch != "ende":  
    summe = summe + float(verbrauch)  
    anzahl = anzahl + 1  
    verbrauch = input("Geben Sie einen Verbrauch ein (oder ende): ")  
durchschnitt = summe / anzahl  
print(durchschnitt)
```


Die While-Schleife

- Allgemeine Form:


```
while Bedingung:  
    code_block  
weitere Anweisungen
```

- Solange die Bedingung (*Schleifenkopf, Header*) erfüllt ist, wird der Code-Block (*Schleifenrumpf*) ausgeführt.
- Danach wird mit den „weiteren Anweisungen“ fortgefahren
- Ist die Bedingung „von Anfang an“ nicht erfüllt, wird der Schleifenrumpf überhaupt nicht durchlaufen.
- Natürlich kann der Code-Block wieder Schleifen enthalten (man spricht dann von *verschachtelten Schleifen*.)
- oder auch if-Anweisungen.

Die Break-Anweisungen

- Kleine Unschönheit der aktuellen Lösung: Die Input-Anweisung muss zweimal programmiert werden (Die Eingabe wird ja in der Schleifenbedingung benötigt).
- Es gibt eine weitere Lösung:  `break.py`
- `break` :
 - Die Ausführung des Schleifenrumpfes wird abgebrochen.
 - Die Schleife wird verlassen, dass es wird mit der ersten Anweisung hinter der Schleife fortgefahren.

Die Continue-Anweisung

- Die Lösung `break.py` erlaubt negative Zahlen als Eingabe. Das ist bei einem Verbrauchswert aber nicht sinnvoll.
- Bei der Eingabe eines negativen Wertes soll die Schleife aber nicht verlassen werden.
- Lösung:  `continue.py`
- `continue`:
 - Die Ausführung des Schleifenrumpfes wird abgebrochen.
 - Die Schleife wird nicht verlassen. Es wird mit dem Schleifenkopf (der Schleifenbedingung) fortgefahren.
- Die Anweisungen `break` und `continue` wirken auf die aktuelle Schleife. Nicht auf evtl. vorhandene äußere Schleifen.

Fallbeispiel: Quersumme

- Die Quersumme einer Zahl ist die Summe der Ziffern.
- Bsp: Die Quersumme von 4328 ist $4 + 3 + 2 + 8 = 17$
- Beispiel: Eine Zahl ist durch 3 teilbar, wenn die Quersumme durch 3 teilbar ist.
- Um die Quersumme zu berechnen, muss man alle Ziffern einer Zahl betrachten.
- Algorithmus in Umgangssprache:

```
Eingabe einer Zahl  
quersumme = 0  
für jede Ziffer der Zahl tue:  
    quersumme = quersumme + Wert der Ziffer
```

- Man muss also über die Ziffern der Zahl „schleifen“.
- Das Ganze in Python: 📄 quersumme.py

Die For-Schleife

- Allgemeine Form:

```
for variable in kollektion:  
    code_block  
    weitere Anweisungen
```

- `variable`: Eine beliebige (auch neue) Variable.
- `kollektion`: Ein Ausdruck, der eine Kollektion bezeichnet, z.B. eine Zeichenkette.
- Der `code_block` wird so oft durchlaufen, wie es Elemente in der Kollektion gibt.
- In jedem Schleifendurchlauf nimmt die `variable` einen Wert der Kollektion an.
- Wenn alle Element durchlaufen wurden, wird das Programm mit den `weitere Anweisungen` fortgesetzt.

For-Schleife: Beispiel

```
for buchstabe in "Python":  
    print("Aktueller Buchstabe: ", buchstabe)
```

- Als Schleifenvariable (hier `buchstabe`) kann eine beliebige Variable verwendet werden (neu oder vorher schon verwendet).
- Im Beispiel wird der Schleifenrumpf 6 mal durchlaufen.
- Im ersten Schleifendurchlauf enthält die Variable `buchstabe` den Wert `"P"`.
- Im zweiten Schleifendurchlauf enthält die Variable `buchstabe` den Wert `"y"`.
- usw.

Zusammenfassung Schleifen

In Python gibt es zwei Arten von Schleifen:

While-Schleife

Ein Codeblock (*Schleifenrumpf*) wird so lange durchlaufen, wie eine Bedingung (*Kopf*) wahr ist.

For-Schleife

Im Schleifenrumpf werden alle Elemente einer Kollektion (z.B. String, Liste) durchlaufen.



Die `break`- und `continue`-Anweisung gibt es bei beiden Schleifenformen.

❓ Übung 05 - Schleife

- Schreiben Sie ein Programm, das als Eingabe erhält:
 - eine Zeichenkette (z.B. `"Anna"`)
 - ein Zeichen (z.B. `"n"`)
- und als Ausgabe die Anzahl des Vorkommens des Zeichens in der Zeichenkette ausgibt (im Beispiel 2).
- siehe Ordner `uebungen/05_uebung_schleife_buchstaben`

❓ Übung 06 - Schleife II

- Schreiben Sie ein Programm, das als Eingabe eine Zeichenkette erhält, die einen Satz enthält,
- und als Ausgabe die Länge des längsten Wortes in dem Satz ausgibt.
- Beispiel:
 - Eingabe: "Dies ist ein langer Satz ohne Nebensatz"
 - Ausgabe: 9
- siehe Ordner `uebungen/06_uebung_max_wortlaenge`

❓ Übung 06A - Zusatzübung Schleifen

- Schreiben Sie ein Programm, das ein Kreuz auf dem Bildschirm ausgibt. Die Anzahl der Zeile sind eine Programmeingabe. Bei der Eingabe 5 soll das Kreuz also folgendermaßen aussehen:

```
x    x
  x  x
   x
  x  x
x    x
```

- siehe Ordner `uebungen/06A_uebung_kreuz`

Fallbeispiel

- Aufgabenstellung: Der Anwender gibt eine a priori nicht bekannte Anzahl von Zeichenketten ein. (Die Zeichenketten können beispielsweise auch aus einer Datei kommen).
- Es soll die Anzahl unterschiedlicher Zeichenketten bestimmt werden.
- Problem: Da die Anzahl der Zeichenketten nicht bekannt ist, kann nicht für jede Zeichenkette eine Variable eingeführt werden.
- Da alle Zeichenketten verglichen werden müssen, funktioniert auch das Vorgehen bei der Durchschnittsberechnung nicht (addiere die eingegebenen Zahlen so lange nicht `ende` eingegeben wird).
- Wir brauchen also eine Möglichkeit, mehrere Werte in einer Datenstruktur zu speichern.
- In Python gibt es dazu den Datentyp Liste (`list`).

Der Datentyp Liste

- Eine Liste ist eine Sequenz von (beliebigen) Werten (beliebigen Typs).
- Einführung einer Liste ohne Elemente („leere Liste“):

```
meine_liste = []
```

- Hinzufügen eines Elementes zu einer Liste:

```
meine_liste.append("Max Mustermann")
```

- Testen, ob ein Element zu einer Liste gehört:

```
"Max Mustermann" in meine_liste ①  
"Thomas Müller" in meine_liste ②
```

① ergibt True

② ergibt False

Beispiel Liste

siehe: `beispiel_liste.py` 

```
mitgliederliste = []
anzahl = int(input("Wieviele Mitglieder wollen Sie eingeben: "))
nummer = 0
while nummer < anzahl:
    nummer = nummer + 1
    neues_mitglied = input("Geben Sie den Namen des Mitgliedes ein: ")
    if neues_mitglied in mitgliederliste:
        print("Dieses Mitglied gibt es schon!")
    else: mitgliederliste.append(neues_mitglied)

print("Es gibt", len(mitgliederliste), "Mitglieder!")
```


Methoden

- Wdh.: `meine_liste.append("Max Mustermann")`
- Eine *Methode* ist eine Funktion, die für einen Wert aufgerufen wird.
- Dies geschieht mit der „Punkt-Notation“
- In dem obigen Beispiel wird also die *Methode* `append` für die Liste `meine_liste` aufgerufen.
- Und hat den Effekt, dass die Zeichenkette `"Max Mustermann"` zu der Liste hinzugefügt wird.

Erzeugung von Listen

- Eine Liste ist eine Sequenz von (beliebigen) Werten (beliebigen Typs).
- Eine Liste wird mit `[]` erzeugt.
- Beispiel:

```
s = "Ein String"  
x = 3.14  
i = 123  
l = [s, x, i, True]
```

- Die Liste `l` hat 4 Elemente.

Länge, Indizierung und Änderung von Elementen

- Länge und Indizierung wie bei Strings: `len(l): 4`
- `l[0]` : Ist die Zeichenkette "Ein String"
- Im Gegensatz zu Zeichenketten, kann ein Element einer Liste verändert werden:

```
l[0] = "Ein anderer String"
```

- Nach dieser Anweisung ist das 0-te Element der Liste `l` die Zeichenkette

```
"Ein anderer String"
```

- Daher nennt man den Datentyp Liste einen *veränderlichen Datentyp*.

Verschachtelte Listen

- Da Listen beliebige Werte enthalten können, können auch wiederum Listen in Listen enthalten sein:

```
l = ["Hallo", "Welt", [1, "1"]]
```

- `l[2]` referenziert die Liste `[1, "1"]`.
- `l[2][0]` referenziert den `0`-ten Eintr. dieser Liste, also den Wert `1`

Löschen von Elementen

Mit der `remove`-Methode kann ein Element aus einer Liste gelöscht werden:

```
l = [1,2,"Hallo"]  
l.remove(2)
```

Danach sieht die Liste so aus:

```
[1,"Hallo"]
```

Weitere Listenmethoden^[1]

```
l.pop(i)
```

Gibt i-tes Element zurück und entfernt es aus Liste

```
l.extend(t)
```

Erweitert Liste um alle Elemente der Liste t.

```
l.remove(x)
```

Entfernt das erste Vorkommen von x aus l

```
l.count(x)
```

Gibt die Anzahl der Vorkommen von x in l zurück

```
l.index(x)
```

Gibt die Position von x in der Liste l zurück

```
l.insert(i, x)
```

Fügt den Wert x an die Stelle i in der Liste ein

+ -Operation auf Listen

- Zwei Listen können „addiert“ werden.
- Das Ergebnis ist eine neue Liste, die die Elemente der beiden addierten Liste enthält:
- Beispiel:

```
l = [1,2]  
s = ["h","a"]  
t = l + s
```

Dann ist `t` eine neue Liste:

```
[1,2,"h","a"]
```

Schleife über Listen

Mit der for-Schleife kann auch über die Elemente einer Liste geschleift werden (❗
schleife_liste.py)

```
l = ["Das", "ist", "eine", "Liste", 1, 2, 3]

for element in l:
    print(element)
```


❓ Übung 07 - Listen

- Die Verbrauchsdaten eines PKW seien in einer Liste von Gleitkommazahlen gegeben.
- Schreiben Sie ein Programm, das den Durchschnittsverbrauch des PKW berechnet.
- s. Verzeichnis `Uebungen/07_uebung_liste_verbrauch`

Referenzen

- Wdh.: Einer Variablen kann man einen Wert zuweisen. Die Variable stellt also einen „Behälter“ für einen Wert dar.
- Genauer: Die Variable verweist auf einen Wert.
- Bei einer Zuweisung wird die Referenz auf den Wert kopiert.
- Bsp: **i**

```
l1 = [1,2,3]  
l2 = l1
```


- Die Variable `l2` verweist nun auf denselben Wert, also dieselbe Liste, wie `l1`.
- Dies sieht man über `id(l1)` und `id(l2)`
- Konsequenz: Ändert man die Liste `l2`, so ist auch `l1` geändert.

Erläuterung

Kopieren von Listen

- Soll `l2` nicht auf dieselbe Liste wie `l1` verweisen, so muss man eine Kopie der Liste erstellen:
 - `l2 = l1.copy()`
 - `l2 = l1[:]`
- Analog dazu, dass in einer Variablen eine Referenz auf einen Wert gespeichert wird, werden in einer Liste Referenzen auf die enthaltenen Element gespeichert.
- Enthält also eine Liste eine Liste („verschachtelte Listen“) und soll diese Liste auch kopiert werden, so ist eine sog. „Tiefenkopie“ notwendig.
- Mehr dazu: https://www.python-kurs.eu/python3_deep_copy.php

Schleifen und Listenkopien

- Ändert man eine Liste während man über die Liste schleift, sollte man über eine Kopie der Liste schleifen.
- Andernfalls könnte es zu Verwerfungen kommen.
- Bsp.: `schleife_liste_kopie.py` 

Fallbeispiel

- Ein Anwender gibt eine Menge von Namen ein (oder sie werden aus einer Datei gelesen).
- Es soll bestimmt werden, wie oft ein Name vorkommt.
- Wir müssen uns also pro Name seine Anzahl merken (und gegebenenfalls hochzählen).
- Wünschenswert wäre also ein Datentyp, mit dem man Zuordnungen der Art `Name → Anzahl` speichern kann.
- So etwas gibt es und nennt sich `Dictionary` (Wörterbuch) - in anderen Programmiersprachen `Map` (Abbildung).
- Ein Dictionary ist eine Menge von Schlüssel, Wert - Paaren (Key, Value).
- In unserem Beispiel sind die Namen die Schlüssel und die Werte die jeweilige Anzahl.


Fallbeispiel Lösung (! beispiel-dict.py)

```
name_zu_anzahl_dict = {} ❶
while True:
    name = input("Geben Sie einen Namen ein: ")
    if name == "Ende":
        break
    if name in name_zu_anzahl_dict: ❷
        name_zu_anzahl_dict[name] += 1 ❸
    else:
        name_zu_anzahl_dict[name] = 1 ❹

for name in name_zu_anzahl_dict: ❺
    print("Der Name", name, "kommt", name_zu_anzahl_dict[name] , "mal vor.")
```

- ❶ Erzeugung eines „leeren“ Dictionaries (ohne Einträge)
- ❷ Test, ob es schon einen Eintrag für den eingegebenen Namen gibt.
- ❸ Ja: erhöhe den Eintrag um 1.
- ❹ Nein: erzeuge für den Namen einen neuen Eintrag mit dem Wert 1
- ❺ Schleife über die Schlüssel des Dictionaries

Fallbeispiel Anmerkung

- Wie würden Sie das Problem lösen, wenn es in Python keine Dictionaries gäbe?
- Idee (beispielsweise): Verwaltung von zwei Listen:
 - Eine mit den eingegebenen Namen
 - und „parallel“ dazu eine mit den entsprechenden Zahlen
- Wenn ein Name eingegeben wurde, muss man zunächst in der ersten Liste schauen, ob es den Namen schon gibt:
 - ja: Sei `i` der entsprechende Index des Namens in der ersten Liste. Erhöhe in der zweiten Liste den Eintrag an der Stelle `i` um eins.
 - nein: Hänge an die erste Liste den Namen und an die zweite Liste eine `1`
- Übung : ausprogrammieren
- Frage: Ist das effizient für eine sehr große Anzahl von Namen?

Erzeugung von Dictionaries und Zugriff

- Leeres Dictionary: `d = {}`
- Mit Einträgen:

```
en_de = {"red": "rot", "green": "grün"}  
verbr = {"Januar": 32.1, "Februar": 21.3, "März": 48.7}
```

- Zugriff auf einen Eintrag: `verbr["Januar"]` wird zu `32.1` ausgewertet
- Änderung eines Eintrages: `verbr["Januar"] = 34.7` ändert den Eintrag für `"Januar"`
- Zufügen eines Eintrages: `verbr["April"] = 21.3` fügt einen Eintrag für `"April"` hinzu

Erlaubte Datentypen

- Als Einträge in einem Dictionary („rechte Seite“, *Wert*) sind beliebige Datentypen erlaubt, also auch
 - Listen oder
 - wieder Dictionaries (man spricht dann von „verschachtelten“ Dictionaries)
- Als *Schlüssel* („linke Seite“) sind nur *unveränderliche* Datentypen erlaubt, also z.B. Strings, aber keine Listen)

```
dict = {}  
dict["Hallo"] = "Welt" ①  
dict[[1,2]] = "Schöne Liste" ②
```

- ① Erlaubt
- ② Nicht erlaubt

Schleifen über Dictionaries

Schleifen_dict.py !

```
d = {"eins":1, "zwei":2, "drei": 3}

# Schleife ueber die Schluessel
for key in d:
    print(key)

# Schleife ueber die Schluessel-Wert-Paare
for key, value in d.items():
    print("Schlüssel: ", key, "Wert: ", value)

# Schleife ueber die Werte
for value in d.values():
    print(value)
```

Weitere Operationen auf Dictionaries^[2]

```
d.clear()
```

Löscht alle Einträge

```
d.copy()
```

Erzeugt eine „flache“ Kopie

```
d.pop(k)
```

Gibt den Wert zum Schlüssel `k` und löscht diesen

```
d.popitem()
```

Liefert beliebiges Schlüssel-Wert-Paar und löscht dieses

```
d.update(d2)
```

Fügt Dictionary `d2` zu `d` hinzu

Schleife - Programmierprinzip „Flagge“

Anwendungsfall

Innerhalb einer Schleife soll getestet werden, ob ein gewisser Fall eintritt.

Prinzip

```
flagge = False
while ... :
    if .... :
        flagge = True
        break
if flagge:
    ....
```

Beispiel

flagge.py 

Einschub - Der Datentyp NoneType

- Es gibt genau einen Wert vom Typ NoneType: None
- Eine Variable wird erst dann angelegt, wenn man ihr einen Wert zuweist.
- Möchte man eine Variable anlegen, aber verdeutlichen, dass sie (noch) keinen „vernünftigen“ Wert hat, so kann man ihr den Wert

```
None
```

zuweisen.

- Beispiel: Ein Wert einer Liste ist (noch) unbekannt:

```
verbr = [21.3, None, 31.4]
```

- In einem booleschen Ausdruck (Ausdruck, der einen Wahrheitswert ergibt), wird None als False interpretiert.
- Beispiel: siehe [Programmierprinzip Vorgänger](#)

Einschub - Auswertung boolescher Ausdrücke

- Ein boolescher Ausdruck wird von links nach rechts ausgewertet.
- Sobald das Ergebnis feststeht, wird die Auswertung abgebrochen
- Beispiel:

```
if a > 20 and a < 100:
```

- Wenn `a` den Wert `10` hat, steht nach der Prüfung von `a > 20` fest, dass der Ausdruck den Wert `False` ergibt.
- Es wird also nicht mehr geprüft, ob `a < 100` ist.

Schleife - Programmierprinzip „Vorgänger“

Anwendungsfall

In einem Schleifendurchlauf werden Werte aus dem vorherigen Durchlauf benötigt.

Prinzip

```
vorgaenger_wert = None
while ... :
    aktueller_wert = ...
    if vorgaenger_wert:
        # Berechnung mit dem aktuellen Wert und
        # dem vorangegangenen Wert
    vorgaenger_wert = aktueller_wert
```

Beispiel

vorgaenger.py 

Fallbeispiel - Römische Zahlen

- Eingabe: Zeichenkette, die eine römische Zahl darstellt, z.B. XIV
- Ausgabe: Wert als Dezimalzahl, z.B. 14
- Systematik:
 - Symbole: "I" \triangleq 1, "V" \triangleq 5, "X" \triangleq 10, "L" \triangleq 50, "C" \triangleq 100, "D" \triangleq 500, "M" \triangleq 1000
 - „Normalerweise“ steht ein „größeres“ Symbol vor einem „kleineren“ Symbol. Dann werden die Werte addiert (XV \triangleq 15).
 - Stehen gleiche Symbole nebeneinander, werden sie addiert (IIII \triangleq 3).
 - Steht ein Symbol mit einem kleineren Wert vor einem Symbol mit einem größeren Wert, so wird der Kleinere subtrahiert (IV \triangleq - 1 + 5)

Algorithmus

```
ergebnis = 0
```

```
Schleife über die Eingabe:
```

```
    Aktueller Wert = Dezimalwert des aktuellen Zeichens
```

```
    Wenn es einen Vorgänger gibt:
```

```
        Wenn Wert des Vorgängerzeichens < Aktueller Wert:
```

```
            ergebnis = ergebnis - Wert des Vorgängerzeichens
```

```
        andernfalls
```

```
            ergebnis = ergebnis + Wert des Vorgängerzeichens
```

```
    Wert Vorgängerzeichen = aktueller Wert
```

```
    ergebnis = ergebnis + aktueller Wert
```

Implementierung

- Die Abbildung von einer römischen Ziffer auf ihren Dezimalwert speichern wir in einem Dictionary.
- Wir wenden die Idee aus [Programmierprinzip Vorgänger](#) an.
 - Wir können es uns hier einfach machen und zum Start den Vorgängerwert auf 0 setzen. Die Addition / Subtraktion von 0 ist unschädlich.
- Siehe: `roem3.py` !

Schleife über mehrere Zahlen

- Wir wollen eine Schleife einbauen, so dass der Anwender mehrere römische Zahlen nacheinander eingeben kann.
- Siehe: `roem4.py` !
- Die Lösung ist unschön und unübersichtlich.

Funktion zur Umrechnung

- Motivation für das nächste Kapitel *Funktionen*
- Wir haben bisher schon Funktionen verwendet (z.B. `input`, `Wurzel`, `abs` ...).
- Wünschenswert: Funktion:
 - Eingabe: Zeichenkette, die eine römische Zahl darstellt
 - Ausgabe: Dezimalwert
- Diese kann dann von beliebigen Stellen aufgerufen werden, z.B. innerhalb einer Schleife.
- In der Schleife ist die Implementierung der Funktion nicht wichtig.
- Siehe: `roem5.py` !

Modul für römische Zahlen

- Motivation für *Module*
- Wir haben schon die import-Anweisung gesehen.
- Wünschenswert: Auslagerung der Funktion zur Umrechnung römischer Zahlen in ein „Modul“ und dessen Verwendung.
- siehe `roem6_mod.py` und `roem6_anw.py` !

Referenzen

- [Kle] Bernd Klein, Einführung in Python 3