

# Programmierung mit Python für Programmierer

## Kapitel 2 - Kontrollstrukturen

Autor: Dr. Christian Heckler

# Vorbemerkungen

- Verwendete Literatur: siehe Referenzen
- Verwendete Symbole:
  - **!**: Beispielprogramm
  - **i**: Weitere Erläuterungen im Kurs
  - **?**: Übung

# Ein- und Ausgabe

- Eingabe:

- Input-Funktion: `eingabe = input("Ihre Eingabe")`
- Die Input-Funktion liefert immer eine Zeichenkette zurück.
- Möchte man also eine Zahl eingeben, muss die Zeichenkette konvertiert werden:

```
zahl = float(input("Eingabe Zahl: " ))
```

- Ausgabe:

- Print-Funktion
- Ausgabe mehrere Objekte durch Komma getrennt
- Automatische Umwandlung in Zeichenkette
- Bsp.: `print("Das Ergebnis ist:", 42)`

# Anweisungsblöcke

- Anweisungen werden in „Anweisungsblöcke“ strukturiert.
- Dies geschieht in Python durch Einrückung.
- Anweisung mit dergleichen Einrücktiefe gehören dabei zusammen und bilden einen Block
- Die Einrückungen müssen einheitlich mit Leerzeichen oder Tabulatoren erfolgen.
- Empfehlung:
  - Leerzeichen (Tabulatortaste mit Leerzeichen „belegen“)
  - Pro Stufe 4 Leerzeichen

# Bedingte Anweisungen

Drei Formen:

- Einfache `if`-Anweisung:
  - Wenn eine Bedingung erfüllt ist, werden die folgenden Anweisungen (Anweisungsblock) ausgeführt.
  - Andernfalls nicht, und die Programmausführung geht hinter der `if`-Anweisung weiter
- `if ... else`:
  - Je nach Bedingung wird der eine oder der andere Anweisungsblock ausgeführt
- `elif`:
  - Vereinfachte Schreibweise bei verschachtelten Bedingungen

# Einfache if-Anweisungen

- Allgemeine Form

```
if bedingung:  
    anweisungen  
anweisungen
```

- Beispiel [\[Kle\]](#):

```
alter = int(input("Dein Alter"))  
if alter < 12:  
    print("Zu jung")  
    print("Waehle einen anderen Film")  
print("Programmende")
```

# If-Anweisung mit Else-Teil

- Allgemeine Form

```
if bedingung:  
    anweisungen  
else:  
    anweisungen  
anweisungen
```

- Beispiel [\[Kle\]](#):

```
alter = int(input("Dein Alter"))  
if alter < 12:  
    print("Zu jung")  
    print("Waehle einen anderen Film")  
else:  
    print("Viel Spass")  
print("Programmende")
```

# „Verschachtelte“ Blöcke

- In einem Anwendungsblock, der in einer If-Anweisung vorkommt, kann auch eine weitere If-Anweisung vorkommen.
- Das nennt man dann verschachtelte If-Anweisung.
- Bsp [\[Kle\]](#):

```
alter = int(input("Dein Alter"))
if alter < 4:
    print("Der Film ist zu kompliziert")
else:
    if alter < 12:
        print("Viel Spass")
    else:
        if alter < 16:
            print("Bist Du Dir sicher?")
        else:
            print("Wollen Sie sich das antun?")
print("Programmende")
```



# Elif

- Das ist im Zweifelsfall schwer zu lesen. Daher gibt es mit „elif“ eine Abkürzung [\[Kle\]](#)

```
alter = int(input("Dein Alter"))
if alter < 4:
    print("Der Film ist zu kompliziert")
elif alter < 12:
    print("Viel Spass")
elif alter < 16:
    print("Bist Du Dir sicher?")
else:
    print("Wollen Sie sich das antun?")
print("Programmende")
```

# Kurzschreibweisen

- Den „ternären Operator“ ( `x = bedingung ? wert1 : wert2` ) gibt es nicht. Stattdessen:

```
x = wert1 if bedingung else wert2
```

- Einzeilige Bedingungen:

```
if bedingung: anweisung
```

- Kein Switch. Statt dessen:
  - `elif` (insbesondere einzeilig, s. oben)
  - evtl. Verwendung von Dictionaries

# Bedingungen

- Was genau steht hinter dem „if“?
- Ein „Ausdruck“, dessen Wert „wahr“ oder „falsch“ ist.
- Vergleichsoperatoren ⓘ:

Operator	Bedeutung	Beispiel
==	gleich	42 == 42
!=	ungleich	42 != 43
<	kleiner	42 < 43
<=	kleiner gleich	42 <= 42
>	größer	43 > 42
>=	größer gleich	42 >= 42

# Bedingungen II

- Der Ausdruck hinter `if` wird automatisch nach bool konvertiert, z.B.

```
liste = []  
if liste: ❶
```

❶ Statt: `if len(liste) == 0:`

- Zu `False` ausgewertet wird:
  - numerische Null-Werte
  - leere Zeichenketten
  - leere Listen, leere Tupel, leere Dictionaries
  - Der `None`-Wert

# Schleifen

- In Python gibt es zwei Typen von Schleifen:
  - `while`-Schleife
  - `for`-Schleife

# 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*.)

# Break und Continue

- `break` :
  - Die Ausführung des Schleifenrumpfes wird abgebrochen.
  - Es wird mit der ersten Anweisung hinter der Schleife fortgefahren.
  - Beispiel: `while.py` !
- `continue` :
  - Die Ausführung des Schleifenrumpfes wird abgebrochen.
  - Es wird mit dem Schleifenkopf (Bedingung) fortgefahren.
  - Beispiel: `continue.py` !
- Beide Anweisungen wirken auf die aktuelle Schleife. Nicht auf evtl. vorhandene äußere Schleifen.

# Else-Teil einer While-Schleife

- Python-Spezialität.
- Der `else`-Teil wird ausgeführt, wenn und sobald die Schleifenbedingung nicht mehr zu trifft.
- Beim Verlassen der Schleife via `break` wird der `else`-Teil nicht ausgeführt.
- Beispiel: Zahlenraten **!**
- Übung: Wie würde man das implementieren, wenn es kein Else der While-Schleife gäbe? **?**



# Die For-Schleife

- Allgemeine Form:

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


- `variable`: Eine beliebige (auch neue) Variable.
- `kollektion`: Ein Ausdruck, der ein iterierbares Objekt bezeichnet (s. später)
- 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.

# For-Schleife: Sonstiges

- Iteration über beliebige „iterierbare Objekte“
  - Listen, Tupel
  - Strings
  - Dictionaries
  - auch eigene Klassen, wenn sie die entspr. Methoden impl.
- Auch bei for-Schleifen:
  - `break`
  - `continue`
  - `else`-Teil
- Weiteres Beispiel: `for .py` 

# „Klassische“ For-Schleifen

- In anderen Programmiersprachen gibt es häufig For-Schleifen, die einen Bereich von ganzen Zahlen durchlaufen, z.B. Java, C(++):

```
for (int i=1; i<13; i++) {}
```

- Diese werden häufig benötigt, um die Indizes einer Zeichenkette oder einer Liste zu durchlaufen.
- Da man in Python mit der For-Schleife direkt über solche Objekte schleifen kann, wird die klassische Form selten benötigt.


# Range-Funktion

- Wenn doch, gibt es in Python die `range`-Funktion:
  - liefert einen Bereich ganzer Zahlen, über den man iterieren kann.
  - Bsp.:

```
for i in range(1,13):  
    print("Monat: ", i)
```

- Es wird nicht a priori eine Liste der Zahlen erstellt.

# Schleife über Listen

- 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` 

# Dateien

- Die im Programm verarbeiteten Daten sind nach der Programmausführung weg.
- Möglichkeit der dauerhaften („persistenten“) Speicherung: Dateien
- Datei: Menge von logisch zusammenhängenden und meist sequentiell geordneten Daten, die auf einem Speichermedium dauerhaft gespeichert werden und mittels eines Bezeichners bzw. Namens wieder identifizierbar und damit ansprechbar sind.
- Eindimensionale Aneinanderreihung von Bits.

# Lesen aus einer Datei


- Lesen aus einer Datei:

```
fobj = open("dateiname", "r")
for line in fobj:
    print(line)
fobj.close()
```

- Beispiel: 🚫 `datei1.py`
- Datei wird zeilenweise gelesen als Zeichenkette (String).
- Jede Zeile enthält am Ende den Zeilenumbruch
- Den muss man evtl. entfernen ( `line.strip()` )
- Beispiel: 🚫 `datei2.py`
- Die Datei muss am Ende „geschlossen“ werden.




## Lesen aus einer Datei 2

- Möglicherweise muss man zu Weiterverarbeitung eine Typkonvertierung durchführen:
- Beispiel:  `datei3.py`

# Schreiben in eine Datei

- Analog werden Strings zeilenweise in eine Datei geschrieben:

```
fobj = open("ausgabedatei", "w")  
fobj.write("Zeile1\n")  
fobj.write("Zeile2\n")  
fobj.close
```

- Dabei ist `"\n"` ein Zeichen (!), nämlich das Zeichen für den Zeilenumbruch (für alle Betriebssysteme!).
- Das Zeilenumbruchszeichen muss explizit für jede Zeile geschrieben werden.
- Gegebenenfalls müssen also die zu schreibenden Daten in Strings umgewandelt werden (plus Zeilenumbruch).
- Beispiel:  `datei4.py`

# Funktionen `read` und `readlines`

- Lesen aller Zeilen in eine Liste:

```
list_of_lines = open("meinedatei", "r").readlines()
```

- Lesen aller Zeilen in einen einzigen String:

```
string_with_lines = open("meinedatei", "r").read()
```

# With -Anweisung

- Geht während der Verarbeitung der Schleife etwas schief (Programm-Abbruch, Ausnahme (s. später)), wird die Datei nicht geschlossen. Abhilfe:

```
with open("dateiname","w") as fobj:  
    anweisungsblock
```

- Ein explizites close ist nicht mehr nötig.
- Die Datei wird auch geschlossen, wenn im Anweisungsblock eine Ausnahme geworfen wird.

# Einschub Kontextmanager

- Die With-Anweisung läßt sich auch zur Verwaltung selbst programmierter Ressourcen nutzen.
- Stichwort dazu: „Context Manager“

# Dateiöffnungsmodi

- "r" : lesen
- "w" : (über-) schreiben
- "a:" anhängen
- "x" : schreiben, wenn die Datei noch nicht existiert. Andernfalls wird Ausnahme geworfen.
- angehängtes + : Datei wird zum Lesen und Schreiben geöffnet.

# Binärdateien

- Dateien können auch im binären Modus geöffnet werden.
- `"rb"`, `"wb"`, `"ab"`, `"xb"`
- Als Objekte müssen dann bytes statt Strings verwendet werden.

# Formatierte Ausgaben (Bildschirm, Datei)

- Print-Funktion
  - mehrere Ausgaben möglich
  - Trennzeichen kann über `sep=` gesetzt werden (Standard `" "`).
  - Endezeichen kann puer `end=` gesetzt werden (Standard `"\n"`).
- Format-Methode auf Strings:
  - analog `printf` bei C
  - Beispiel:

```
betrag = 123.45678  
print("Ergebnis={0:10.3f} DM".format(betrag))
```

- Ausgabe: `Ergebnis= 123.457 DM`
- s. auch: [https://www.python-kurs.eu/python3/formatierte\\_ausgabe.php](https://www.python-kurs.eu/python3/formatierte_ausgabe.php)



# ❓ Übungen Schleifen

- Fröhliche Zahlen
- Römische Zahlen
- (Andreas-) Kreuz
- Bestimmung der maximalen Wortlänge in einer Zeichenkette, die einen Satz darstellt.

# Referenzen

- [Ste] Ralph Steyer: Programmierung Grundlagen, Herdt-Verlag
- [Kle] Bernd Klein: Einführung in Python 3, Hanser-Verlag
- [Kof] Kofler: Python – Der Grundkurs, Rheinwerk Computing
- [EK] Johannes Ernesti, Peter Kaiser: Python 3 – Das umfassende Handbuch, Rheinwerk Computing
- [Mat] Eric Matthes: Python Crashkurs – Eine praktische, projektbasierte Programmierereinführung, dpunkt.verlag
- [Swe] Sweigart: Eigene Spiele programmieren: Python lernen