Programmierung mit Python

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2026-05 Dozent: Karsten Flügge

Themen des Tages

- Kontrollstrukturen
- Bedingungen (if,elif,else)
- Schleifen (for,while)
- Blöcke

Kontrollstrukturen

Kontrollstrukturen sind grundlegende Anweisungen(Statements) des Programmierens.

Sie ermöglichen es dem Programm, bestimmte Entscheidungen zu treffen oder Aktionen wiederholt auszuführen.

Keywords: if (kein then) else, while

Wenn Dann Sonst : **if** condition**:** act() **elif** case2**: else:**
Solange : **while** condition**: act()**
Jedes... **for** item **in** list: act()
Versuche : **try:** act() **except:**
Mustererkennung **match x: case** ... switch
Spezialkontext **with x** Verwerfe x nach Verwendung

Alle mit Doppelpunkt : und Einrückung (alle ausser match mit else!)

Kontrollstrukturen

if condition : action # Bedingung sollte ≈ Truthy/False
Auswertung von einem Ausdruck "condition". Wahr? Dann 'Block' ausführen.

```
temperatur = 20
if temperatur > 15 :  # Wenn ... : dann
  print("Es ist warm.")
⚠️ ^^ 💡 Indent Einrückung und Doppelpunkt nicht vergessen!
```

2. If-Else Anweisung:

- **Zweck**: Entscheidet, ob ein bestimmter Codeblock ausgeführt wird, abhängig von einer Bedingung,
- ***oder* was geschieht, wenn die Bedingung nicht erfüllt ist**

```
if temperatur > 15:
  print("Es ist warm.")
else: # oder, otherwise, andernfalls
  print("Es ist kalt.")
```

Kontrollstrukturen (neu)

3. Elif (else if) deckt andere Fälle ab

```
temperatur = 10
if temperatur > 15:
  print("Es ist warm.")
elif temperatur > 5: # falls erste Bedingung nicht erfüllt ist
  print("Es ist kühl.")
elif temperatur > 0: # falls ersten Bedingungen nicht erfüllt sind
  print("Es ist kalt.")
else: # letzter "Default", keine obere Bedingung erfüllt
  print("Es ist eiskalt.")  # übrige Restfälle
```

elif ist kurz und identisch zu "else if" in anderen Sprachen
wir müssen für die **mittleren Fälle immer elif (oder wenn)** schreiben!!

Kontrollstrukturen mögliche Fehler

mögliche Fehler:

Fall vergessen

```
temperatur = float(input("Wie warm?")) # typ str muss in float umgewandelt werden
if 5 < temperatur < 15: # logische Fehler: 15,16 Grad ist nicht mit behandelt
  print("Es ist kühl.")
elif temperatur > 16:
  print("Es ist warm.")
else:  # Fängt alle Fälle auf … auch ungewollte?
  print("Es ist kalt.")
```

und Reihenfolge ist hier wild durcheinander

Kontrollstrukturen (Wiederholung)

for Schleife "**für** jedes Element einer Liste etc: mache etwas"

Kontrollstrukturen sind grundlegende Anweisungen(Statements) des Programmierens.
Sie ermöglichen es dem Programm, bestimmte Entscheidungen zu treffen oder Aktionen wiederholt auszuführen.

Bisher kennen wir in python:

1. **Einfache for Schleifen für Strings und Listen:**

Jedes Element "item" einer Liste, Iterable abarbeiten

```
for buchstabe in "abc": # Schleifen-Variable frei wählbar
  print(buchstabe)
  # block

for zahl in [1, 2, 3]:   # each aus anderen Sprachen wird in py weggelassen
  print(zahl) # aktuelle Zahl


Eventuell schon untergekommen:

for zahl in range(5):  # von 0 bis 1 weniger!  [0,5) "5 exklusive"
  print(zahl) # 0 bis 4
```

Kontrollstrukturen (Wiederholung)

Schleifen-Variable

in for Schleifen wird ad-hoc eine sogenannte **Schleifen-Variable** definiert
Diese ist frei wählbar (**benennbar**) und zeigt auf **aktuelles Element**

```
for buchstabe in "abc": # Schleifen-Variable frei wählbar
  print(buchstabe)


for zahl in [1, 2, 3]:   # each aus anderen Sprachen wird in py weggelassen
  print(zahl) # jeder Wert der Liste wird einmal angenommen
# anders als in vielen anderen Sprachen bleibt letzter Wert gesetzt
```

Eventuell schon untergekommen:

```
for zahl in range(5):  # von 0 bis 1 weniger!  [0,5) "5 exklusive"
  print(zahl) # 0 bis 4
```

Kontrollstrukturen (neu)

While-Schleife:

- **Zweck**: Führt einen Codeblock wiederholt aus, solange eine **Bedingung** wahr ist.

```
zaehler = 0
while zaehler < 5:
  print(zaehler)
  zaehler += 1
```

```
solange Bedingung:  # wird in jedem Durchlauf neu geprüft.
 aktions_block()
```

Damit die Schleife beendet werden kann, muss sich die Bedingung ändern.
Entweder **innerhalb** unseres Blocks oder **von außen:**

```
while test_environment():
   print("alles gut")
```

Vertiefung Kontrollstrukturen

Was haben if und while gemeinsam?

```
if Bedingung:
 Aktion

while Bedingung:
  Aktion
```

1. **wenn** eine Bedingung erfüllt ist, führe Aktion einmal aus
2. **solange** eine Bedingung erfüllt ist, führe Aktion ggf mehrfach aus

⚠️ die Bedingung wird bei while immer wieder neu geprüft werden

Bedingungen

Was macht (in python) im Kern eine Bedingung aus?

1. Ein Ausdruck, der zu einem **booleschen Wert** ausgewertet wird, also zu **`True`** oder **`False`**

```
if True: print("immer true")

# via Logische Operatoren:
a = True
b = False
bedingung = a or b
if bedingung: print("true")


# via Vergleichsoperatoren
a = 5
b = 10
bedingung = a < b  # Dies ergibt True, weil 5 kleiner als 10 ist.
if bedingung: print("true")

# Mitgliedschaftsoperatoren:
liste = [1, 2, 3]
bedingung = 2 in liste  # Dies ergibt True, weil 2 ein Element der Liste ist.
if bedingung: print("true")

# neu: Identitätsoperatoren
a = [1, 2]
b = [1, 2]
bedingung = a is b  # Dies ergibt False, weil a und b unterschiedliche Objekte sind, obwohl sie denselben Inhalt haben.
```

Bedingungen

VERGLEICHE == **äquivalent**

```
a = [1, 2]
b = [1, 2]
bedingung = a == b # True, weil Objekte selben Inhalt haben

nicht vergessen: doppeltes ==, weil = Zuweisung ist


# neu: Identitäts-Operatoren
a = [1, 2]
b = [1, 2]
bedingung = a is b  # Dies ergibt False, weil a und b unterschiedliche Objekte
sind, obwohl sie denselben Inhalt haben.
```

Bedingungen "truthyness"

Was macht (in python) im Kern eine Bedingung aus?

Ein Wert wird als "**truthy**" betrachtet, wenn er in einem booleschen Kontext als **`True`** interpretiert wird.
Ein Wert wird als "**falsy**" betrachtet, wenn er in einem booleschen Kontext als **`False`** interpretiert wird.

`In Python sind folgende Werte "falsy":`

- **`False`**
- **`None`** `(neu! 'Nichts' ähnlich wie null, nil, undefined in anderen Sprachen) Platzhalter`
- Jede Form von **numerischer Null**: **`0, 0.0, 0j`**
- **Leere** Sammlungen: **`[],{},(),set(),''`** (leerer String)
- Spezialfälle wie bestimmte **Klassen**instanzen, die eine `__bool__` oder `__len__` Methode definieren, die `False` oder `0` zurückgeben

if Bedingung: …
if **Existent**: …

⚠️ In python sind alle **Werte** die nicht falsy sind automatisch truthy!

```
if 1: print("truthy!")
if "a": print("truthy!")
if [1]: print("truthy!")
if {a:1} : print("truthy!")
```

```
if x: print("x exists!")
```

Bedingungen "falsyness"

Was macht (in python) im Kern eine Bedingung aus?

Ein Wert wird als "**falsy**" betrachtet, wenn er in einem booleschen Kontext als **`False`** interpretiert wird.

In Python sind folgende Werte "falsy":

- `False`
- **`None`** (neu! 'Nichts' ähnlich wie null in anderen Sprachen)
- Jede Form von **numerischer Null**: **`0, 0.0, 0j`**
- **Leere** Sammlungen: **`[], {}, (), set(), ''`** (leerer String)
- Spezialfälle wie bestimmte Klasseninstanzen, die eine `__bool__` oder `__len__` Methode definieren, die `False` oder `0` zurückgeben

- **falsy ≈ inhaltslos!**

⚠️ In python sind alle **Werte** die nicht falsy sind automatisch truthy!

```
if 0: print("wird nicht erreicht")  # 0.0
if None: print("wird nicht erreicht")
if []: print("wird nicht erreicht") # nichts da
if {} : print("wird nicht erreicht") # nichts da
if "" : print("wird nicht erreicht") # nichts da, kein Text
if x : print("wird nicht erreicht")  # Variablen werden auf den WERT geprüft
```

Bedingungen "truthyness"

Was macht (in python) im Kern eine Bedingung aus?

💡 Falsyness ist sehr nützlich um **Existenz** von Werten zu Prüfen (nicht versus kein)

```
freunde = None
freunde = []


if freunde:
   print("Es gibt Freunde!")
else:
  print("Es wurden keine Freunde gefunden")

Das ist zu lesen als "wenn es Freunde gibt" ….

Diese Bedingung ist dann nicht erfüllt, wenn freunde() None, 0, [], {}  liefert!
Diese Bedingung ist dann erfüllt, wenn freunde() True, 1, ["Peter"],… zurück liefert!

Damit spart man sich das hässliche if(value!=null) … aus anderen Sprachen

Kleines Problem
temperature = 0  # mehrdeutig: es gibt keine Temperatur Messung oder Temperatur ist 0 Grad Celsius/Fahrenheit ???
if temperature: wäre nicht gewünscht
Lösung: if temperature == 0: …  expliziter Vergleichsoperator == oder units
```

Bedingungen "truthyness"

```
>>> if print: print("wir koennen drucken!")
  wir koennen drucken!


>>> print==True  # truthy aber nicht True!
False

>>> if variable_exists: print("ok")
...
Traceback (most recent call last):
  File "<python-input-42>", line 1, in <module>
    if variable_exists: print("ok")
       ^^^^^^^^^^^^^^^
NameError: name 'variable_exists' is not defined


fast NIEMALS
if x == True:   => if x:
if x == False:  => if not x:
```

Wiederholung "Blöcke und Einrückung"

Eine Aktion ist immer per Einrückung (indent) zusammengefasst und kann mehrere Zeilen enthalten:

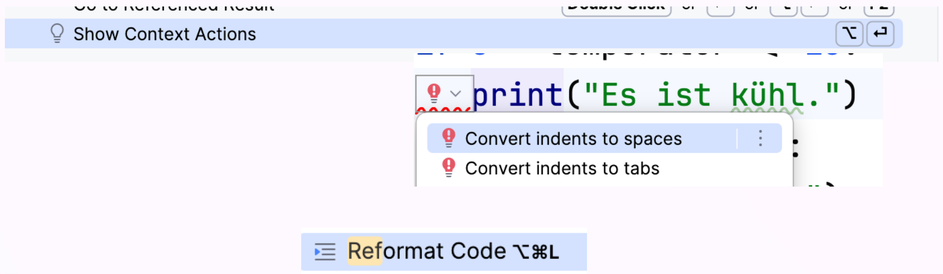
Diese Gruppierung (von Ausdrücken oder Statements) per indent nennt man "Block"

```
temperatur = 25
if temperatur > 25:
    print("Es ist schön warm:")  # Block Zeile 1
    print(f"{temperatur} Grad") # Block Zeile 2
```

Häufige Fehlerursache: falsche Einrückung

```
temperatur = 25
if temperatur > 25:
    print("Es ist schön warm:")
  print(f"{temperatur} Grad")
print("Weiter im Programm")
```

IndentationError "EinZahnung" EinRückung



Bei Kooperation mit anderen auf EinRückung einigen, PEP Standard: 4 spaces

Fehlerquelle

Fehlerquelle: Modifikation des for Schleifen-Objektes

Wenn in einer Schleife über eine Liste iteriert und **währenddessen die Liste verändert wird** (z.B. durch Hinzufügen oder Entfernen von Elementen), kann dies zu unerwarteten Ergebnissen führen.

```
liste = [1,2,3]
neue_liste = []
for element in liste:
  neue_liste.append(liste.pop()) # modifiziert liste

print(neue_liste) # [3, 2]  WAAAS?! Warum nicht [3,2,1]?
# 1. Liste wird verändert, während sie durchlaufen wird VERBOTEN!
```

Auswege:

- Konstruktion eines **neuen Objektes ohne altes Objekt zu modifizieren**
- **z.B. liste.copy() Kopie**
- in while Schleifen die **Bedingung vorher** fest machen:
  - bedingung = …; while bedingung: …

Fehlerquelle

Fehlerquelle: ungewollte Modifikation der Schleifenvariablen

```
i = 0
while i < 5:
  print(i)
  if i == 2:
  # Eine Modifikation, die die Bedingung beeinflusst
    i = 5  # Setzt i direkt auf 5, was die Schleife dann beendet
  else:
    i += 1
```

```
Ausgabe
0
1
2
```

Tipps zur Vermeidung solcher Fehler

- **Immutability nutzen**: Verwenden Sie unveränderliche Datenstrukturen (tuple) oder -methoden, die die Originaldaten nicht ändern.
- **Kopieren von Listen**: Erstellen Sie eine Kopie der Liste, bevor Sie sie in einer Schleife modifizieren.
- **Klare Schleifenlogik**: Stellen Sie sicher, dass die Logik innerhalb der Schleife **klar** und **vorhersehbar** ist.
- **Verwendung von Flags**: Setzen Sie Steuerungsflags, um den Zustand innerhalb der Schleife zu kontrollieren.

Fehlerquelle

Lösung Fehlerquelle: Modifikation der Schleifenbedingung

```
i = 0
ist_schleife_fertig = False # 'flag'
while not ist_schleife_fertig:
   print(i)
    if fertig:
       ist_schleife_fertig = True
    liste.pop()
```

Tipps zur Vermeidung solcher Fehler

- **Immutability nutzen**: Verwenden Sie unveränderliche Datenstrukturen oder -methoden, die die Originaldaten nicht ändern.
- **Kopieren von Listen**: Erstellen Sie eine **Kopie der Liste**, bevor Sie sie in einer Schleife modifizieren.
- **Klare Schleifenlogik**: Stellen Sie sicher, dass die Logik innerhalb der Schleife **klar** und **vorhersehbar** ist.
- **Verwendung von Flags**: Setzen Sie Steuerungsflags, um den Zustand innerhalb der Schleife zu kontrollieren.

Verlassen von Schleifen

mit dem break keyword lässt sich eine Schleife umgehend verlassen

```
for i in range(10): // [0..10)
  print(i)
   if i == 5:
      break  # Verlässt die Schleife, wenn i gleich 5 ist


i = 0
while True # Schleifen-Bedingung
  i += 1
  print(i)
   if i == 5: # Abbruch Bedingung
      break  # Verlässt die Schleife, wenn i gleich 5 ist


print("Fertig") # wird direkt durch break erreicht


# Ausgabe: 0 1 2 3 4

ähnlich return in Funktionen
der Code nach der Schleife wird ausgeführt (hier: print)
exit() # stoppe sofort das komplette Programm! unabhängig von Schleifen etc
```

Verlassen von Schleifen

mehrere Abbruch Bedingungen sind erlaubt,
aber es kann nur ein break erreicht werden

```
for i in range(10): // [0..10)  inklusive 0, exklusive 10
  print(i)
  if i == 5:
    break  # Verlässt die Schleife, wenn i gleich 5 ist
  if i == -5:
    break  # Verlässt die Schleife, wenn i gleich -5 is

sauerstoff = 100 # 100% Flasche ist voll
glas_heil = True
while True: # Tauche
  glas_heil, sauerstoff = tauche() # hole upgedateten Zustand
  glas_kaputt = not glas_heil
  if glas_kaputt:
    break
  if sauerstoff<20: # Abbruch Bedingung
    break # breche Tauchen sofort ab
  sleep(1) # Sekunden!! => 0.1
```

Beispiele

Wann nutze ich Schleifen?

a) Um alle Elemente einer Liste einmal 'anzugucken/abzuarbeiten':
   a) schreibe Email an *jeden Kontakt*
   b) addiere Kontostand von jedem Kunden

b) Um Texte zu durchforsten

c) Um wiederholt auf Zustand zu warten / prüfen
   a) Ist Mausknopf gedrückt (threads 'Hintergrundprogramme')
   b) Läuft Waschmaschine noch?

^ da gibt es auch andere Ansätze (async / call by event)

d) ...

Aufgaben Schleifen

a) Gib alle Zahlen von 1 bis 20 aus. mit **for** i in range(20): print(i**+1**) oder range(1,21)

b) Gib alle *geraden* Zahlen zwischen 1 und 50 aus.

c) Gib alle ungeraden Zahlen zwischen 1 und 30 aus. mit **while**

d) Berechne die Summe aller Zahlen von 1 bis 100.

e) Berechne das Produkt n! 1*2*3*4*5*...*n per Schleife

Extra:

1. Gib alle Elemente der Liste aus: farben = ["rot", "grün", "blau"]
2. Zähle, wie viele Zahlen größer als 10 sind: zahlen = [4, 11, 7, 25, 3, 18]
3. Suche die größte Zahl in einer Liste: zahlen = [3, 8, 2, 19, 5]
4. Prüfe, ob eine Liste eine gerade Zahl enthält: zahlen = [1, 5, 7, 8, 9]
5. Gib ein kleines Dreieck aus:

```
    1
  22
  333
  4444
  55555
```

Aufgaben Schleifen

12 etwas fortgeschrittene Aufgaben

1. Finde alle Zahlen von 1 bis 100, die durch 3 oder 5 teilbar sind.
2. Zähle, wie viele Primzahlen zwischen 2 und 100 liegen.
3. Gib die ersten 15 Fibonacci-Zahlen aus.
4. Finde das Maximum und Minimum in einer Liste ohne `max()` und `min()`.
5. Zähle Wörter nach ihrer Länge:

```
woerter = ["Haus", "Python", "und", "Kontrollstruktur", "if"]
```

6. Entferne Duplikate aus einer Liste, ohne `set()` zu verwenden.
7. Prüfe, ob eine Liste aufsteigend sortiert ist.
8. Finde alle gemeinsamen Elemente zweier Listen.
9. Zähle, wie oft jede Zahl in einer Liste vorkommt.
10. Erzeuge ein Schachbrettmuster aus `#` und `.` der Größe 8×8.
11. Simuliere 200 Würfe mit `random.randint(1, 6)` und zähle die Häufigkeiten.
12. Finde das zweitgrößte Element einer Liste ohne Sortieren.

Aufgaben Schleifen

12 einigermaßen schwierige Aufgaben

1. Prüfe, ob ein Wort ein Palindrom ist, ohne `[::-1]`.
2. Prüfe, ob zwei Wörter Anagramme sind, ohne `sorted()`.
3. Implementiere eine einfache Primfaktorzerlegung.
4. Finde alle Pythagoreischen Tripel `(a, b, c)` mit `1 <= a < b < c <= 100`.
5. Berechne den größten gemeinsamen Teiler mit dem euklidischen Algorithmus.
6. Berechne das kleinste gemeinsame Vielfache aus dem ggT.
7. Implementiere Binärsuche auf einer sortierten Liste.
8. Simuliere einen eindimensionalen Random Walk über 100 Schritte.
9. Finde die längste zusammenhängende steigende Teilfolge in einer Liste.
10. Löse FizzBuzz allgemein: Für beliebige Paare `(Teiler, Wort)` soll die Ausgabe kombiniert werden.
11. Implementiere eine einfache Klammerprüfung für `()[]{}`.
12. Schreibe eine Schleife, die Conway's Game of Life für ein kleines 5×5-Feld einen Schritt weiterrechnet.

geschachtelten Schleifen

```
Innerhalb einer Schleifen kann man natürlich andere Schleifen verwenden
for i in range(8): # [0..8)
  for j in range(8): # [0..8) # Innere Schleife
    print(i, j)
```

Aufgabe: gebe das Paar i,j aus (vorher: was erwartet man?)

```
0, 0
0, 1
0, 2
...
0, 7
1, 0
1, 1
...
7, 7
```

Verlassen von geschachtelten Schleifen

mit dem break keyword lässt sich die *innere* Schleife umgehend verlassen!

```
for i in range(10): # [0..10)
  for j in range(10): # [0..10)
    print(i, j)
    if j == 5:
      break  # verlässt die j-Schleife, i läuft aber weiter!
```

Aufgabe: gebe das Paar i,j aus (vorher: was erwartet man?)

*ähnlich **return** in Funktionen, welche auch Schleifen sofort beenden*
der Code nach der Schleife wird ausgeführt (hier: print)

Verlassen von geschachtelten Schleifen

mit dem break keyword lässt sich die *innere* Schleife umgehend verlassen!

um die äussere Schleife zu verlassen muss man diesen '**Trick**' verwenden:

```
exit_outer = False # Schleifen Flag
for i in range(10): # [0..10)
  if exit_outer:  # sogenanntes Schleifen Flag
    break # verlässt die i-Schleife
  for j in range(10): # [0..10)
    print(i, j)
    if j == 5:
      exit_outer = True
      break  # verlässt die j-Schleife und danach auch die i-Schleife
   break # verlässt die äusser-Schleife
```

Aufgabe: gebe das Paar i,j aus (vorher: was erwartet man?)

*ähnlich **return** in Funktionen, welche auch Schleifen sofort beenden*
der Code nach der Schleife wird ausgeführt (hier: print)

Verlassen von Schleifen

während break nur die Schleife verlässt,
wird mit **return** auch direkt die Funktion verlassen

```
def finde_erste_gerade(zahlen):
  for num in zahlen:
    if num % 2 == 0:
      return num  # Gibt die erste gerade Zahl zurück und verlässt die Funktion
# return None  # Keine gerade Zahl gefunden implizit immer
```

0 oder None?

```
def finde_erste_gerade(zahlen):
  for num in zahlen:
    if num % 2 == 0:
      return num  # Gibt die erste gerade Zahl zurück und verlässt die Funktion
# return None  # Keine gerade Zahl gefunden implizit immer

liste = [3,1,0,7,8] => gefunden = 0
liste = [3,1,7] => gefunden = None
gefunden = finde_erste_gerade(zahlen)

if gefunden or gefunden==0: print("gefunden", gefunden)  ⚠️ Achtung, 0 ist FALSY!!
else print("nichts gefunden")
```

Überspringen von Schleifenschritten

mit dem continue keyword lässt sich eine Schleifen-Schritt beenden.
Es wird dann mit dem nächsten Schritt am Anfang der Schleife fortgesetzt.

```
for i in range(10):
  if i % 2 == 0: # modulo "Rest" 0 heisst Zahl ist gerade
    continue  # Überspringt alle folgenden Befehle in der Schleife, wenn i gerade ist
    print(i)  This code is unreachable!
  print(i)
# Ausgabe: 1 3 5 7 9
```

```
for letter in "abcd":
  if letter in "aeiou":
    continue
  print(letter) # nur Konsonanten
# Ausgabe: b c d
```

continue ≈ fortsetzen ≈ skip ≈ next (in anderen Sprachen)

"Lebensdauer" von Schleifenvariablen

```
"Lebensdauer" von Schleifenvariablen

for i in range(10):
  if i % 2 == 0:
    continue
  print(i) # Achtung Einrückung!
print("Am Ende lebt i weiter: ", i)
# Ausgabe: 1 3 5 7 9    Am Ende lebt i weiter: 9
```

Abbrechen / Überspringen von Schleifenschritten

mit dem **break** keyword lässt sich eine **Schleife komplett beenden.**
Es wird dann mit dem nächsten Schritt am Anfang der Schleife fortgesetzt.

mit dem **continue** keyword lässt sich eine **Schleifen-Schritt beenden.**
Es wird dann mit dem nächsten Schritt am Anfang der Schleife fortgesetzt.

```
for i in range(10):
  if i % 2 == 0: # modulo "Rest" 0 heisst Zahl ist gerade
    break  # Breche ALLES ab, wenn i gerade ist
  print(i)
# Ausgabe:  nada;)
```

```
for i in range(10):
  if i % 2 == 0: # modulo "Rest" 0 heisst Zahl ist gerade
    continue  # Überspringt alle folgenden Befehle in der Schleife, wenn i gerade ist
  print(i)
# Ausgabe: 1 3 5 7 9
```

Else für Schleifen

mit dem **break** keyword lässt sich eine **Schleife komplett beenden.**
Es wird dann mit dem nächsten Schritt am Anfang der Schleife fortgesetzt.

Ein **else:** block für Schleifen wird erreicht, wenn in der Schleife KEIN break vorkommt!

```
for i in range(10):
  print(i)
else: # KEIN BREAK in dieser Schleife auf dieser Ebene, alles gut, keine Sonderunterbrechung
  print("es wurde kein break aufgerufen")
```

Das selbe für while

```
while True:
  if False: break
else: # KEIN BREAK aufgerufen, alles gut, keine Sonderunterbrechung
  print("es wurde kein break aufgerufen") # wird nicht gedruckt
```

Syntactic Diabetis: "Ein Stück Syntactic Sugar too much" / Semantic Border case

Hintergrund Schleifen

Heutzutage lässt man oft verschieden Aufgaben in einem Programm **parallel abwickeln.** Dazu gibt es das große Oberthema **async / threads.**

```
while True:
  connection = accept_client()
  connection.send("Hello")
```

selbst dieser Sonderfall sollte die Option offenhalten, zu beenden

Endlos-Schleifen

In seltenen Fällen möchte man ein Programm ewig weiter laufen lassen.
Vereinfachtes typisches Beispiel: **Server**

```
while True:
  connection = accept_client()
  connection.send("Hello")
```

selbst dieser Sonderfall sollte die Option offenhalten, zu beenden

Endlos-Schleifen

In seltenen Fällen möchte man ein Programm ewig weiter laufen lassen.
Vereinfachtes typisches Beispiel: **Server**

```
run_server = True
while run_server:
  connection = accept_client()
  connection.send("Hello")
  if genug:
    run_server = False

selbst dieser Sonderfall sollte die Option offenhalten, zu
beenden
```

goto statement

*es gab früher ein keyword '**goto**' mit dem man zu einer beliebigen Stelle im code springen konnte. Dieses ist glücklicherweise aus (fast) allen Sprachen entfernt worden, weil es zu sehr verwirrendem und fehlerhaften Spaghetti Code führte.*

***labeled** loops*

```
schleife1:
for i in range(10):
  continue schleife1
```

gibt es in anderen Sprachen, aber nicht in python

Match Case

Mustererkennung **neues keyword** **match ... case** syntactic sugar seit py 3.10 gibt es ein

Damit lassen sich Werte / Objekte 'strukturell' überprüfen:

```
punkt = (3, 4)

match punkt: # prüfe / analysiere 'switch'
    case (0, 0): # konkretest
        print("Ursprung")
    case (x, 0): # if y == 0:
        print("x-Achse", x)
    case (0, y):
        print("y-Achse", y)
    case (x, y):
        print("Punkt", x, y)
     case _: # NICHT default: else:  "wildcard"
       print("Punkt nicht erkannt")
```

Es wird in python nur ein Fall gematched, der ERSTE der matched "trifft"! kein break nötig

Das ist eine kompaktere Art, als Objekte mit **if ... elif** zu prüfen

Match Case

match ... case pattern matching ad-hoc variablen:

Man kann in den cases einen oder mehrere **Werte Vorgeben** und den/die anderen **Werte 'auffangen'**:
Ad hoc Variablen.

```
punkt = (3, 4)
match punkt:
    case (0, 0):  # x und y fest auf 0
        print("Ursprung")
    case (x, 0):  # nur y auf 0
        print("x-Achse", x)
    case (0, y):
        print("y-Achse", y)
    case (x, y):  # alle anderen Punkte
        print("Punkt", x, y)
     case _: # NICHT default: else:  "wildcard"
       print("kein Punkt!")
```

Das ist eine kompaktere Art, als Objekte mit **if ... elif** zu prüfen
Syntaktischer Zucker, nicht zu viel.

```
There should be one-- and preferably only one --obvious way to do it.
```

Match Case

match ... case pattern matching ad-hoc variablen:

Man kann in den cases **Werte 'verwerfen'**:

```
punkt = (3, 4)
match punkt:
    case (0, _):
        print("y-Achse, Wert egal")
    case (_, y):  # alle anderen Punkte
        print("Punkt mit Höhe", y) # x egal
    case _: # NICHT default: else:  "wildcard"
      print("kein Punkt!")
```

_ heißt in diesem Kontext meist:

- **wildcard** pattern
- **discard** pattern
- throwaway variable
- **dummy** variable
- ignored binding

Das ist eine kompaktere Art, als Objekte mit **if ... elif** zu prüfen

Tipp: Einzeiler

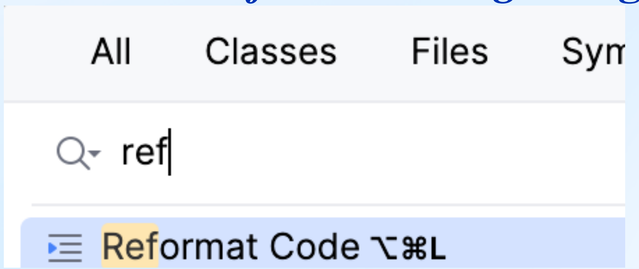
Bei Blöcken, welche nur eine Aktion enthalten, kann man sich die Einrückung sparen:

if True: print("OK") # Einzeiler gehen bei Reformat verloren
else: print("no!")

Bei Blöcken kann man sie mit Semikolon zusammenfassen
if True: print("OK");print("OK2")

nur für kleine Experimente, besonders in Interpreter >>>
Generell im Interpreter Blöcke/Indents vermeiden!

Wird bei *Reformatierung* in eigene Zeile gesteckt



Vorschau else in try

```
# try / except / else / finally

try:
    datei = open("test.txt")
    print("kein Fehler")
except FileNotFoundError:
    print("nicht gefunden")
else: # kein Fehler
    print(datei.read())
    datei.close()
finally:
    print("immer")
```

Aufgaben if / elif

a) Schulnoten Bewertung
gegeben Note von 1 bis 6, gib mit if…elif… das passende Wort aus (2==Gut)

b) Tropen
Schreibe einfaches if…elif… welches für verschiedene Temperatur-Bereiche folgende Ausgaben tätigt:
Heiße Zone
Gemäßigte Zone
Kalte Zone

c) Nutzerstatus
Es gibt zwei Arten von Nutzern mit passendem Status:
rolle = "Editor" / "Admin"
status = "aktiv" / "inaktiv"
Schreibe ein Programm welches für alle Kombinationen eine passende Meldung ausgibt, z.B. Admin+aktiv: print("Vollzugriff gewährt.")

Aufgaben if truthy / falsy

e) 1. truthies zählen

```
values = [0, 1, "", "Hallo", None, [], [1, 2], False, True, {}]
truthies = 0
for value in values:
  if …
     truthies += 1
```

Was ist der (erwartete) Wert von truthies? (erst im Kopf, dann überprüfen) # 4

f) 2. truthy Eingaben

```
x = input("Geben Sie etwas ein: ")
if x:
    print("Truthy")
else:
    print("Falsy")
```

Findet jemand eine Eingabe, welche Falsy ist? Warum ist "False" nicht falsy? Ist str, nicht Bool!!

g) 3. truthy/falsy in Logischen Operatoren

Etwas ungewohnt, aber man kann truthy/falsy in Logischen Operatoren verwenden!

```
lst = [0, 1, 2]
if len(lst) and lst[0]:
    print("Beide Bedingungen sind truthy.")
else:
    print("Mindestens eine Bedingung ist falsy.")
```

Aufgaben Schleifen

s1. **FizzBuzz**
Das Standard-**FizzBuzz**-Problem besteht darin, Zahlen von 1 bis 20 auszudrucken, aber für **Vielfache** von **drei** „**Fizz**“ anstelle der Zahl und für die **Vielfachen** von **fünf** „**Buzz**“ auszugeben. Für Zahlen, die Vielfache von sowohl drei als auch fünf sind, geben Sie „**FizzBuzz**“ aus.

s2a. **Passwortprüfer**
Schreibe ein Programm, das prüft ob ein Passwort eine Mischung aus **Buchstaben** (Groß- und Kleinbuchstaben), **Ziffern** und **Sonderzeichen** enthält. Tipp: if x in "abcde…"

s2b. **Passwortgenerator**
Schreiben Sie ein Programm, das ein sicheres Passwort generiert. Das Programm sollte nach der gewünschten Länge des Passworts fragen und sicherstellen, dass das generierte Passwort eine Mindestkomplexitätsanforderung (2a) erfüllt.
(TIPP: nutze zB Funktion randint(0,26) lower_chars[my_rand_index])

s3. **Temperaturumrechnungstabelle**
Schreiben Sie ein Python-Skript, das eine Tabelle von Temperaturen von **Grad Celsius in Fahrenheit** für 0 bis 100 Grad Celsius in 5-Grad-Schritten ausdruckt. Verwenden Sie eine Schleife, um jedes Paar im Format "XX°C ist YY°F" zu berechnen und auszudrucken.

s4. **Implementiere die Collatz-Vermutung:**
    Wenn n **gerade** ist, **teilen** Sie sie durch **2.**
    Wenn n **ungerade** ist, **multipliziere** sie mit **3** und addieren Sie **1.**
    Setze den Prozess fort, bis n 1 wird. Das Programm sollte jeden Zwischenwert von n ausdrucken.
Starte Programm mit verschiedenen Werten. Es sollte immer bei 1 landen.

Vorschau Sondersyntax

- **Einzeilensyntax**

Blöcke mit nur einer Aktion lassen in einer Zeile schreiben:

```
if True: print("OK")
else: print("NO")
```

Es gibt dann keine Einrückung!

- **Verkettete Vergleiche:**

Python erlaubt das Verketten von Vergleichsoperatoren: if **x < y <= z:** entspricht if x < y and y <= z

- **Bedingte Ausdrücke (Ternärer Operator)**

```
    Python unterstützt bedingte Zuweisungen in einer einzigen Zeile:
    x = a if Bedingung else b   (⚠️ ohne Doppelpunkte!)
    // x = Bedingung ? a : b    (in anderen Sprachen)
```

wenn Bedingung x = a , ansonsten setze x = b

- **Schleife mit else:**

Ein else-Block nach einer Schleife wird ausgeführt, nachdem die **Schleife normal endet** (d.h., ohne ein break):

```
for item in iterable:
    if Bedingung:
        break
else:
    # Wird ausgeführt, wenn kein `break` in der Schleife vorkommt
```

Iteratoren (Vorschau)

Was ermöglicht es Objekten wie <class 'range'>, in einer for Schleife verwendet werden zu können

```
for zahl in range(5):
  print(zahl) # 0 bis 4
```

__iter__

Die __iter__ Methode ist ein spezieller Teil eines Python-Klassenkonzepts, das es ermöglicht, dass ein Objekt in Schleifen, wie zum Beispiel in einer for-Schleife, verwendet werden kann. Sie macht ein Objekt zu einem "Iterator", was bedeutet, dass es eine Sequenz von Werten nacheinander zurückgeben kann. Kommen wir drauf zurück, sobald wir Klassen haben.

… später

Iteratoren (yield Generator)

Das Schlüsselwort **`yield`** in Python wird verwendet, um einen **Generator** zu definieren. Ein Generator ist eine spezielle Art von Iterator, der nacheinander Werte generiert, ohne alle Werte im Voraus zu speichern.

```
def zahlen_generator(max):
  i = 0
  while i < max:
    yield i   # return one at a time!
    i += 1

for zahl in zahlen_generator(5): # z.B. alle Primzahlen
  print(zahl)
```

Operatoren

Vorrang von Operatoren, Gegebenenfalls Operationen Klammern oder in mehrere Zeilen teilen.

'Präzedenz' / Priorisierung / Bindung / Reinfolge (Gruppierung)

```
(...)                # Klammern
x[i], x(...), x.y  # Zugriff, Aufruf  z.B. .real
**                   # Potenz
+x, -x, ~x         # Vorzeichen, bitweise Negation
*, /, //, %        # ÷ Punktrechnung   3 * True == 3
+, -                 # Strichrechnung
<<, >>             # Bitverschiebung
&                     # bitweises UND
^                     # bitweises XOR
|                    # bitweises ODER
== != < <= > >=          # Vergleiche
not                    # logisches NICHT  True, False
and                   # logisches UND
or                    # logisches ODER
=  :=  += ...       # Zuweisung; kein Ausdruck
```

Aufgaben Operator Priorisierung

IM KOPF: Was ist das Ergebnis der folgenden Berechnungen in Python und in der realen Welt?

17 % 5 + 2 * 3

25 // 4 + 3 ** 2

30 // 4 % 3

2 + 18 // 4 * 3

7 + 20 % 6 // 2

50 // 6 % 4 + 1

3 * 7 % 5 + 2

100 // 9 // 2

5 + 2 ** 3 % 3

8 % 3 * 4 // 2

9 + 24 // 5 % 2

15 % 4 + 6 // 2 ** 2

Aufgaben Operator Priorisierung

10 // 3 > 3 and 8 % 3 == 2

5 + 2 * 3 == 11 or 8 // 3 == 3

not 20 % 4 == 1

7 % 4 + 1 >= 4 and 9 // 2 < 5

3 ** 2 == 9 and 12 % 5 != 1

18 // 4 == 4 or 3 * 3 == 8

2 + 3 * 4 > 10 and not 5 % 2 == 0

8 // 2 ** 2 == 2 and 7 % 4 < 3

15 % 6 == 3 or 20 // 5 > 10

Lösungen Operator Priorisierung

```
10 // 3 > 3 and 8 % 3 == 2          # False
10 // (3 > 3) and 8 % 3 == 2       # ZeroDivisionError

5 + 2 * 3 == 11 or 8 // 3 == 3     # True
5 + 2 * (3 == 11) or 8 // 3 == 3   # 5

not 20 % 4 == 1                    # True
(not 20) % 4 == 1                  # False

7 % 4 + 1 >= 4 and 9 // 2 < 5      # True
7 % (4 + 1) >= 4 and 9 // 2 < 5    # False

3 ** 2 == 9 and 12 % 5 != 1        # True
3 ** (2 == 9) and 12 % 5 != 1      # False

18 // 4 == 4 or 3 * 3 == 8         # True
18 // (4 == 4) or 3 * 3 == 8       # 18

2 + 3 * 4 > 10 and not 5 % 2 == 0  # True
(2 + 3) * 4 > 10 and not 5 % 2 == 0 # True

8 // 2 ** 2 == 2 and 7 % 4 < 3     # False
(8 // 2) ** 2 == 2 and 7 % 4 < 3   # False

15 % 6 == 3 or 20 // 5 > 10        # True
15 % (6 == 3 or 20) // 5 > 10      # False
```

with operator

Spezialkontext **with x Verwerfe x nach Verwendung**

Deteien Lesen / Schreiben

```
with open("test.txt") as f:
    text = f.read() # Eine Zeile zur Zeit
# f wird am Ende vom Block geschlossen und verworfen
```

f.close() kann man sich so sparen, aber braucht man ohnehin nicht,
weil nach Ende des Programmes/Skriptes alles aufgeräumt wird

Als 'unsauberer' Einzeiler:
lines = open(file).**readlines**()

Am Ende vom Programm wird Datei sowieso geschlossen.

Vokabeln

Kontroll Strukturen
Bedingung (condition)
Einrückung (indent)
Block (Aktionen bei condition)
Ausrückung (dedent)

elif : ansonsten andernfalls / oder wenn ...
default : "durchfall" bis zum letzten Fall / übrige Restfälle