Tag 2

Verzweigungen & Schleifen

Vorkurs-Schnupsis

30. September 2025

TU Darmstadt

Logische Operatoren

Logische Operatoren

- Wie kombiniert man die zwei logischen Ausdrücke `3 < x` und `x < 7`?

  $\Rightarrow$ Mit logischen Operatoren
- Logische Operatoren arbeiten auf dem Datentyp `bool` und können diesen transformieren oder zum Abbilden von Logik verwendet werden
- Das unterscheidet sie von den anderen Operatoren

Logische Operatoren

Auflistung

- Es gibt hierbei 3 verschiedene operatoren die man verwenden kann

Operator	Beschreibung	Beispiel
`and`	Beide sind wahr	`is_car and is_boat`
`or`	Mindestens einer ist wahr	`is_car or is_boat`
`not`	Negiert den Ausdruck	`not is_car`

Logische Operatoren

Verwendung

- Ein regnerischer Tag ist ein Tag mit Wolken und Regen

```
raining = False
cloudy = True
rainyday = raining and cloudy
```

```
x = 10
a = (x == 60)          # -> False
b = 5 > 3              # -> True
not (a or b)           # -> False
b or (x > 50) and a    # -> True
```

Logische Operatoren

Verwendung

Live-Coding

Verzweigungen

Verzweigungen

Es gibt Fälle, in denen Anweisungen nur bei Erfüllung bestimmter Bedingungen ausgeführt werden sollen.

Zur Umsetzung dessen werden verschiedene Zweige im Kontrollfluss benötigt.

Was also ist ein Kontrollfluss?

if-Anweisung

- Unterscheidung nach ja oder nein bzw. wahr oder falsch
- Fragt einen bool-Wert ab
- Anweisung wird ausgeführt, wenn die Abfrage im if-Statement `True` zurückgibt
- Mit `else` kann eine Anweisung definiert werden, die andernfalls geschehen soll
- Die Auswertung erfolgt von oben nach unten

Merksatz: ”Wenn <Bedingung>, dann führe <X> aus, sonst <Y>.”

if-Anweisung

- Mit `if` können wir Bedingungen prüfen
- Wenn die Bedingung `True` ist, wird der eingerückte Block ausgeführt
- Wichtig: **Einrückung** (meist 4 Leerzeichen) gehört zur Syntax!

```
1 x = 5
2 if x > 3:
3     print("x ist größer als 3")
```

if-Anweisung

Live-Coding

if-Anweisung

- Mit `else` können wir einen Block definieren, der ausgeführt wird, wenn die Bedingung `False` ist

```
1 x = 5
2 if x > 3: # if benötigt einen bool als Bedingung
3     print("x ist größer als 3")
4 else:
5     print("x ist nicht größer als 3")
```

if-Anweisung

- Der Code-Block muss immer eingerückt sein! Ansonsten kann Python nicht erkennen, welche Anweisungen zum Block gehören

```
1 if x > 3:
2 print("x ist größer als 3")
3 else:
4 print("x ist nicht größer als 3")
```

⇒ `IndentationError: expected an indented block after 'if'statement on line 1`

- Später mehr zu Einrückungen und Fehlern

if-Anweisung

Codebeispiel

```
num = int(input('Gib eine Zahl an: '))
if num % 3 == 0:
    print('Die eingegebene Zahl ist durch 3 teilbar')
else:
    print('Die eingegebene Zahl ist nicht durch 3 teilbar')
```

Auch Verneinungen sind möglich:

```
num = int(input('Gib eine Zahl an: '))
if num % 3 != 0:
    print('Die eingegebene Zahl ist nicht durch 3 teilbar')
else:
    print('Die eingegebene Zahl ist durch 3 teilbar')
```

if-Anweisung

Mehrere Fälle

Die naheliegende Lösung:

```
1  if my_obj == "cube":
2      print('the square hole')
3  else:
4      if my_obj == "circular":
5          print('into the circle?')
6      else:
7          if my_obj == "rectangular":
8              print('please the rectangle')
9          else:
10             print('INTO THE SQUARE HOLE')
```

if-Anweisung

Mehrere Fälle

Die elegantere Lösung:

```
if my_obj == "cube":
    print('the square hole')
elif my_obj == "circular":
    print('into the circle?')
elif my_obj == "rectangular":
    print('please the rectangle')
else:
    print('INTO THE SQUARE HOLE')
```

match-Anweisung

Die (teilweise) noch elegantere Lösung:

```
match my_obj:
    case "cube":
        print('the square hole')
    case "circular":
        print('into the circle?')
    case "rectangular":
        print('please the rectangle')
    default:
        print('INTO THE SQUARE HOLE')
```

match-Anweisung

(Teilweise noch eleganter)

- Um zwischen einer Reihe an ähnlichen Fällen zu unterscheiden:
  $\Rightarrow$ z.B: x == 1, x == 2, x == 3
- `case` ist äquivalent zu einem `if`/`elif`-Fall
- `default` ist äquivalent zu `else`
- Kann nicht alle Bedingungen eines if-Statements umsetzen (`and`, etc.)
- Ähnlich zu `switch` in Java, kann allerdings noch mehr (z.B. match nach Länge einer Liste), sprengt jedoch den Rahmen des Vorkurses

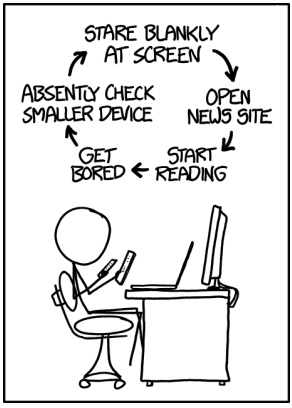
if-Anweisung

Live-Coding

Schleifen I

Grundidee

- Oft ist es nötig, einen Code-Block mehrfach auszuführen
- Dies kann manuell durch Kopieren und Einfügen erreicht werden, ist aber sehr fehleranfällig und unübersichtlich
- Schleifen ermöglichen es, einen Code-Block mehrfach auszuführen, ohne ihn mehrfach schreiben zu müssen
- Ebenso kann die Ausführung anhand einer Bedingung gesteuert werden



while

- Solange die Bedingung im `while`-Statement `True` ist, wird der eingerückte Block immer wieder ausgeführt
- Wichtig: **Einrückung** (meist 4 Leerzeichen) gehört zur Syntax!

```
while <bool>:
    <body>
```

- Der Körper der Schleife wird so lange wiederholt, bis die Bedingung `False` ergibt
- Das heißt die Schleife hört nicht von selbst auf, sondern läuft potentiell unendlich weiter, wenn die Bedingung immer `True` bleibt
- Dies wird meist durch eine Veränderung der Werte im Schleifenkörper erreicht

while

- Beispiel einer Schleife, die von 0 bis 4 zählt und die Buchstaben eines Strings ausgibt:

```
1 i = 0
2 s = 'Lorem'
3 while i < len(s):
4     print(f'Buchstabe an Stelle {i}: {s[i]}')
5     i += 1
```

Merksatz: ”Solange <Bedingung>, dann führe <X> aus.”

if-Schleife

- Eine solche Schleife gibt es offiziell nicht
- In der Programmiersprache C könnte man es aber so umsetzen:

```
    int i = 0;
label:
    printf("Iteration: %d\n", i);
    if (i < 10) {
        goto label;
    }
```

- Bitte nutzt so etwas nicht...

Live-Coding

while-Schleife

Iterationen überspringen

- In einigen Fällen möchte man nicht für alle Iterationen den Schleifenkörper ausführen
- Oder sogar früher abbrechen
- Natürlich kann man den Code in Verzweigungen stecken und einfach ungewollte Iterationen herausfiltern

while-Schleife

Iterationen überspringen

- Dafür gibt es aber einen einfacheren und vor allem effizienteren Weg:
  - `continue`: Mit dieser Anweisung wird die aktuelle Iterationen beendet und zum Schleifenkopf / der Bedingung zurückgekehrt
  - `break`: Diese Anweisung beendet die Schleife komplett. Die Ausführung fährt hinter der Schleife fort
- Wichtig: Beide Anweisungen gelten nur für die innerste Schleife!

while-Schleife

Iterationen überspringen

```
i = 0
while i < 10:
    if i == 5:
    i += 1
        continue
    print(i)
    i += 1
```

```
i = 0
while i < 10:
    if i == 5:
        break
    print(i)
    i += 1
```

Iterationen überspringen

Live-Coding

while-Schleife

Endlosschleife

- Natürlich ist es möglich, dass es bei Schleifen zu keinem Abbruch kommt
- In manchen Fällen ist dies gewollt (z.B. bei periodischen Aufgaben)
- In einigen Fällen ist dies unerwünscht
- Mit Strg+C kann man die Ausführung eines Programmes im Terminal abbrechen

Listen

Grundidee

Wenn man bisher eine Reihe an Werten speichern wollte, hätte man dies eventuell so umgesetzt:

```
a1 = 3
a2 = 4
a3 = 1
a4 = 2
```

- Wie geht dies effizienter bzw. kürzer?
- Wie kann man mit einer Schleife über diese Werte iterieren?

$\Rightarrow$ Listen!

Definition

Eine Liste wird mit eckigen Klammern und (optional) einer Aufzählung an Werten definiert:

```
1 lst = list()       # leere Liste
2 lst = []           # leere Liste
3 lst = [3, 4, 1, 2]
```

Mit `len(lst)` bekommt man die Anzahl der Elemente, oder *Länge* der Liste. Eine Liste speichert Werte an den sog. *Indizes* $0, \ldots,$`len(lst)-1`.

Zugriffe

Um auf einzelne Werte (Lesen oder Schreiben) zuzugreifen, werden ebenfalls eckige Klammern verwendet.

```
# Lesen
lst[0] => 3
lst[2] => 1
```

```
# Besonderheit in Python:
lst[-1] => 2 # entspricht lst[len(lst)-1]
```

```
# Schreiben
lst[0] = 7
lst[1] = 1
lst[2] = 0
```

Live-Coding

Übliche Funktionen

```
lst = [3, 4, 1, 2]
4 in lst            # => True
lst.append(5)       # => [3, 4, 1, 2, 5]
lst.pop()           # => [3, 4, 1, 2]
lst.reverse()       # => [2, 1, 4, 3]
lst.sort()          # => [1, 2, 3, 4]
del lst[1]          # => [1, 3, 4]
```

Übliche Funktionen

```
lst1 = [3, 1, 2]
lst2 = [5, 6, 1]
lst1 + lst2      # => [3, 1, 2, 5, 6, 1]
```

Wichtig: Auch nach einer Zuweisung wird die ”alte” Liste modifziert, eine neue Variable ist nur eine andere Referenz auf die gleiche Liste!

```
lst2 = lst1
lst1.sort()
lst2 => [1, 2, 3]
```

Slices

- Slices sind Teile von Listen (oder Strings)
- Die allgemeine Syntax ist `lst[start:end:step]`
  - `start`: Erster Index des Slices ⇒ inklusives Verhalten
  - `end`: Index nach Ende des Slices ⇒ exklusives Verhalten
  - `step`: Schrittweite und Richtung, erlaubt das Überspringen von Elementen und eine umgekehrte Reihenfolge

Beispiele:

```
1 lst = [3, 4, 1, 2]
2 lst[1:3]    # => [4, 1]
3 lst[1:3:2]  # => [4]
```

Strings

Ähnlichkeit zu Listen

- Zeichenketten verhalten sich in vielen Fällen wie Listen, da sie effektiv nur eine Liste von einzelnen Zeichen sind
- Beispiele:
  - Zugriff: `s[1]` und `lst[1]`
  - Konkatentation: `s1 + s2` und `lst1 + lst2`
  - Slices: `s[1:3]` und `lst[1:3]`
- Funktionen wie `.pop()`, `.sort()`, etc. sind für Zeichenketten nicht implementiert
- Dafür gibt es `.split('trenner')`, mit dem ein String in eine Liste von Strings anhand eines Trennstrings aufgeteilt werden kann
- Außerdem kann man mit `'lorem'.replace('ore', 'i')` einzelne Teile eines Strings ersetzen oder löschen

Live-Coding

Fehlermeldungen verstehen und beheben

Arten von Fehlern

- Python nennt euch fast immer die Stelle, an der der Fehler liegt
-

```
x = 3 * 5)
 File "<python-input-25>", line 1
   x = 3 * 5)
           ^
SyntaxError: unmatched ')'
```

Einrückung

- Python reagiert stark auf fehlende oder unnötige (kurz falsche) Einrückung

```
1 x = 5
2     y = 3
3 z = 2
```

- `IndentationError: unexpected indent`
- Es ist nicht erlaubt, in einer Datei Tabs und Spaces zu mischen

Lexikalische Fehler

- Python beachtet die Groß- bzw. Kleinschreibung von Variablen und Funktionen

```
1 eineZahl = 3
2 print(einezahl)
```

- `NameError: name 'einezahl' is not defined. Did you mean: 'eineZahl'?`

Syntaktische Fehler

- Nichtbeachten der ”Grammatik” von Python führt zu sogenannten *Syntaxfehlern*

```
1 x = 2 + 5) * 3 # ( fehlt am Anfang des Ausdrucks
2 if x < 10      # : fehlt am Ende von if
3     print('Die Zahl ist klein')
```

- `SyntaxError: unmatched ')'`
- `SyntaxError: expected ':'`

Semantische Fehler

- Ein Fehler in der Programmlogik wird als *Semantikfehler* bezeichnet

```
1 x = []
2 y = x[1]
```

- Klassische Beispiele: Teilen durch 0 oder Zugriff auf eine leere Liste
- *IndexError: list index out of range*

Quiz

Quiz

- Wie überprüft man die Teilbarkeit einer Zahl?
- Seien `a = 0` und `s = []`. Welche Fehler treten bei `if (0 < a < 3):` und `a /= len(s)` auf?
- Wie tief darf man in Python schachteln?

Ausblick

Ausblick

In der morgigen Vorlesung erwarten uns:

- Noch mehr Schleifen
- Funktionen
- Dictionaries
- Dateien