



Tag 2

Verzweigungen & Schleifen

Vorkurs-Schnupsis

30. September 2025

TU Darmstadt



Logische Operatoren

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

Logische Operatoren

- Wie kombiniert man die zwei logischen Ausdrücke $3 < x$ und $x < 7$?
⇒ Mit logischen Operatoren



Logische Operatoren

- Wie kombiniert man die zwei logischen Ausdrücke $3 < x$ und $x < 7$?
⇒ Mit logischen Operatoren
- Logische Operatoren arbeiten auf dem Datentyp `bool` und können diesen transformieren oder zum Abbilden von Logik verwendet werden



Logische Operatoren

- Wie kombiniert man die zwei logischen Ausdrücke $3 < x$ und $x < 7$?
⇒ 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
<code>and</code>	Beide sind wahr	<code>is_car and is_boat</code>
<code>or</code>	Mindestens einer ist wahr	<code>is_car or is_boat</code>
<code>not</code>	Negiert den Ausdruck	<code>not is_car</code>

Logische Operatoren

Verwendung

- Ein regnerischer Tag ist ein Tag mit Wolken und Regen

Logische Operatoren

Verwendung

- Ein regnerischer Tag ist ein Tag mit Wolken und Regen

```
1 raining = False
2 cloudy = True
3 rainyday = raining and cloudy
```

Logische Operatoren

Verwendung

- Ein regnerischer Tag ist ein Tag mit Wolken und Regen

```
1 raining = False
2 cloudy = True
3 rainyday = raining and cloudy
```

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

Live-Coding



Verzweigungen

Verzweigungen

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

Verzweigungen

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

Zur Umsetzung dessen werden verschiedene Zweige im Kontrollfluss benötigt.

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?



- Unterscheidung nach ja oder nein bzw. wahr oder falsch

if-Anweisung

- Unterscheidung nach ja oder nein bzw. wahr oder falsch
- Fragt einen bool-Wert ab

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

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

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

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

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

if-Anweisung

- Mit `if` können wir Bedingungen prüfen
- Wenn die Bedingung `True` ist, wird der eingerückte Block ausgeführt

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!

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")
```

Live-Coding

if-Anweisung

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

- 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

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")
```

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`

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

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

if-Anweisung

Codebeispiel

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

Auch Verneinungen sind möglich:

```
1 num = int(input('Gib eine Zahl an: '))
2 if num % 3 != 0:
3     print('Die eingegebene Zahl ist nicht durch 3 teilbar')
4 else:
5     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:

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

match-Anweisung

Die (teilweise) noch elegantere Lösung:

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

match-Anweisung

(Teilweise noch eleganter)

- Um zwischen einer Reihe an ähnlichen Fällen zu unterscheiden:

match-Anweisung

(Teilweise noch eleganter)

- Um zwischen einer Reihe an ähnlichen Fällen zu unterscheiden:
⇒ z.B: `x == 1`, `x == 2`, `x == 3`

match-Anweisung

(Teilweise noch eleganter)

- Um zwischen einer Reihe an ähnlichen Fällen zu unterscheiden:
⇒ z.B: `x == 1`, `x == 2`, `x == 3`
- `case` ist äquivalent zu einem `if/elif`-Fall

match-Anweisung

(Teilweise noch eleganter)

- Um zwischen einer Reihe an ähnlichen Fällen zu unterscheiden:
⇒ z.B: `x == 1`, `x == 2`, `x == 3`
- `case` ist äquivalent zu einem `if/elif`-Fall
- `default` ist äquivalent zu `else`

match-Anweisung

(Teilweise noch eleganter)

- Um zwischen einer Reihe an ähnlichen Fällen zu unterscheiden:
⇒ 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.)

match-Anweisung

(Teilweise noch eleganter)

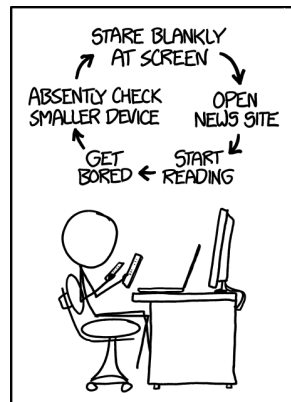
- Um zwischen einer Reihe an ähnlichen Fällen zu unterscheiden:
⇒ z.B: `x == 1`, `x == 2`, `x == 3`
- `case` ist äquivalent zu einem `if/elif`-Fall
- `default` ist äquivalent zu `else`
- Kann nicht alle Bedingungen eines `if`-Statements umsetzen (`and`, etc.)
- Ähnlich zu `switch` in Java, kann allerdings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

Live-Coding

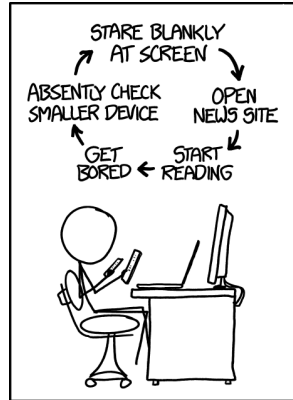
Schleifen I



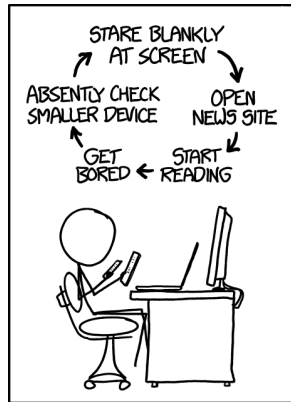
- Oft ist es nötig, einen Code-Block mehrfach auszuführen



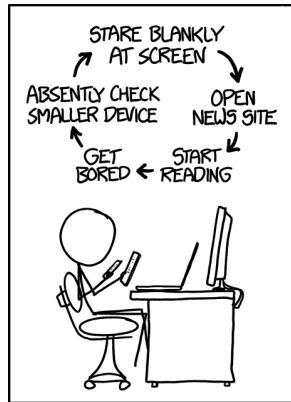
- Oft ist es nötig, einen Code-Block mehrfach auszuführen
- Dies kann manuell durch Kopieren und Einfügen erreicht werden, ist aber sehr fehleranfällig und unübersichtlich



- 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



- 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

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

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

```
1 while <bool>:  
2     <body>
```

while

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

```
1 while <bool>:  
2     <body>
```

- Der Körper der Schleife wird so lange wiederholt, bis die Bedingung `False` ergibt

while

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

```
1 while <bool>:  
2     <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

while

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

```
1 while <bool>:  
2     <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

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

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

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."

- Eine solche Schleife gibt es offiziell nicht

if-Schleife

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

```
1  int i = 0;
2  label:
3      printf("Iteration: %d\n", i);
4      if (i < 10) {
5          goto label;
6      }
```

if-Schleife

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

```
1     int i = 0;
2 label:
3     printf("Iteration: %d\n", i);
4     if (i < 10) {
5         goto label;
6     }
```

- Bitte nutzt so etwas nicht...

Live-Coding

while-Schleife

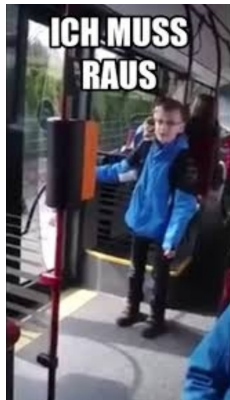
Iterationen überspringen

- In einigen Fällen möchte man nicht für alle Iterationen den Schleifenkörper ausführen

while-Schleife

Iterationen überspringen

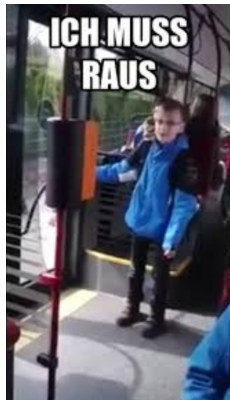
- In einigen Fällen möchte man nicht für alle Iterationen den Schleifenkörper ausführen
- Oder sogar früher abbrechen



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:

while-Schleife

Iterationen überspringen

- Dafür gibt es aber einen einfacheren und vor allem effizienteren Weg:
 - `continue`: Mit dieser Anweisung wird die aktuelle Iteration beendet und zum Schleifenkopf / der Bedingung zurückgekehrt

while-Schleife

Iterationen überspringen

- Dafür gibt es aber einen einfacheren und vor allem effizienteren Weg:
 - **continue**: Mit dieser Anweisung wird die aktuelle Iteration beendet und zum Schleifenkopf / der Bedingung zurückgekehrt
 - **break**: Diese Anweisung beendet die Schleife komplett. Die Ausführung fährt hinter der Schleife fort

while-Schleife

Iterationen überspringen

- Dafür gibt es aber einen einfacheren und vor allem effizienteren Weg:
 - **continue**: Mit dieser Anweisung wird die aktuelle Iteration 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

```
1 i = 0
2 while i < 10:
3     if i == 5:
4         i += 1
5         continue
6     print(i)
7     i += 1
```

while-Schleife

Iterationen überspringen

```
1 i = 0
2 while i < 10:
3     if i == 5:
4         i += 1
5         continue
6     print(i)
7     i += 1
```

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

Live-Coding

while-Schleife

Endlosschleife

- Natürlich ist es möglich, dass es bei Schleifen zu keinem Abbruch kommt



Noo you can't just
print money infinitely



```
while True:  
    print("money")
```

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)



Noo you can't just
print money infinitely



```
while True:  
    print("money")
```

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



Noo you can't just
print money infinitely



```
while True:  
    print("money")
```

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



Noo you can't just
print money infinitely



```
while True:  
    print("money")
```



Listen

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

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

- Wie geht dies effizienter bzw. kürzer?

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

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

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

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

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

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

⇒ **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]
```

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* $0, \dots, \text{len}(lst)-1$.

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

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

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

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

Live-Coding

Übliche Funktionen

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

Übliche Funktionen

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

Übliche Funktionen

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

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

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

- Slices sind Teile von Listen (oder Strings)
- Die allgemeine Syntax ist `lst[start:end:step]`
 - `start`: Erster Index des Slices $\Rightarrow$ inklusives Verhalten
 - `end`: Index nach Ende des Slices $\Rightarrow$ 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]
```


Ähnlichkeit zu Listen

- Zeichenketten verhalten sich in vielen Fällen wie Listen, da sie effektiv nur eine Liste von einzelnen Zeichen sind

Ähnlichkeit zu Listen

- Zeichenketten verhalten sich in vielen Fällen wie Listen, da sie effektiv nur eine Liste von einzelnen Zeichen sind
- Beispiele:

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]`

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`

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]`

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]`
 - Konkatenation: `s1 + s2` und `lst1 + lst2`
 - Slices: `s[1:3]` und `lst[1:3]`
- Funktionen wie `.pop()`, `.sort()`, etc. sind für Zeichenketten nicht implementiert

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

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

Arten von Fehlern

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

```
1 x = 3 * 5)
2 File "<python-input-25>", line 1
3     x = 3 * 5)
4             ^
5 SyntaxError: unmatched ')'
```

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

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

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

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

- `IndentationError: unexpected indent`

- 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

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

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


- 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'?`

- 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')
```

- 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 ')`

- 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 ':'`

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

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

- 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

- 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?

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?

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

In der morgigen Vorlesung erwarten uns:

- Noch mehr Schleifen

In der morgigen Vorlesung erwarten uns:

- Noch mehr Schleifen
- Funktionen

In der morgigen Vorlesung erwarten uns:

- Noch mehr Schleifen
- Funktionen
- Dictionaries

In der morgigen Vorlesung erwarten uns:

- Noch mehr Schleifen
- Funktionen
- Dictionaries
- Dateien