If-Else & Loops

by Woche 3

Ein grundlegendes Konzept in der Programmierung sind Kontrollstrukturen und Loops (Schleifen). In diesem Kapitel werden wir uns daher mit `if-else`-Anweisungen und den `for` und `while` Loops beschäftigen.

If-Else Anweisungen

Die `if-else`-Anweisung ermöglicht es uns, Entscheidungen zu treffen und Code nur dann auszuführen, wenn eine bestimmte Bedingung erfüllt ist.

if

Tatsächlich reicht auch ein `if`-Block aus, um eine Entscheidung zu treffen. Wichtig zu beachten ist, dass der Code, der innerhalb des `if`-Blocks steht, eingerückt sein muss und zwar um vier Leerzeichen oder ein Tab. Um dies besonders deutlich zu machen, sind im folgenden Code zwei `print()`-Statements: Nur das eine ist eingerückt und gehört somit zum `if`-Block.

```
x = 10

if x < 5:
    print("x ist kleiner als 5")
print("Ich werde immer geprintet")
```

```
Ich werde immer geprintet
```

```
x = 3

if x < 5:
    print("x ist kleiner als 5")
print("Ich werde immer geprintet")
```

```
x ist kleiner als 5
Ich werde immer geprintet
```

else

Optional kann auch ein `else`-Block angehängt werden, der ausgeführt wird, wenn die Bedingung im `if`-Block nicht erfüllt ist. Der `else`-Block wird ebenfalls eingerückt und zwar auf der gleichen Ebene wie der `if`-Block.

```
x = 10

if x < 5:
    print("x ist kleiner als 5")
else:
    print("x ist größer oder gleich 5")
```

```
x ist größer oder gleich 5
```

```
x = 3

if x < 5:
    print("x ist kleiner als 5")
else:
    print("x ist größer oder gleich 5")
```

```
x ist kleiner als 5
```

elif

Schließlich kann auch ein `elif`-Block genutzt werden. Dies ist dann sinnvoll, wenn nacheinander mehrere Bedingungen geprüft werden sollen. Der `elif`-Block wird ebenfalls auf die gleiche Ebene wie der `if`-Block eingerückt.

```
x = 10

if x < 5:
    print("x ist kleiner als 5")
elif x == 5:
    print("x ist gleich 5")
else:
    print("x ist größer als 5")
```

```
x ist größer als 5
```

```
x = 5
```

```
if x < 5:
    print("x ist kleiner als 5")
elif x == 5:
    print("x ist gleich 5")
else:
    print("x ist größer als 5")
```

```
x ist gleich 5
```

```
x = 3

if x < 5:
    print("x ist kleiner als 5")
elif x == 5:
    print("x ist gleich 5")
else:
    print("x ist größer als 5")
```

```
x ist kleiner als 5
```

Loops

Loops (Schleifen) sind ein weiteres grundlegendes Konzept in der Programmierung. Sie ermöglichen es, Code mehrmals auszuführen. In Python gibt es zwei Arten von Loops: `for` und `while`.

for

Der `for`-Loop wird genutzt, um sequenziell durch Elemente einer iterierbaren Sammlung, wie beispielsweise einer Liste oder einem String, zu gehen. Eine solche Sammlung wird auch als Sequenz bezeichnet. Eine Sequenz kann als eine geordnete Reihe von Elementen definiert werden, wobei jedes Element einen bestimmten Platz in dieser Reihenfolge hat. Beispielsweise besteht ein String aus einer Sequenz von Buchstaben in einer bestimmten Reihenfolge. Die for-Schleife ermöglicht es, für jedes Element in dieser Sequenz eine bestimmte Aktion oder einen Block von Code auszuführen. Diesen Prozess nennt man Iteration. Der Begriff ‘Iterieren’ bedeutet, jedes Element in der Sequenz einzeln zu durchlaufen, wobei bei jedem Durchgang der Schleife der Codeblock, der innerhalb der Schleife definiert ist, ausgeführt wird. Der Codeblock wird für jedes Element der Sequenz einmal ausgeführt und ist durch Einrückung kenntlich gemacht. Obwohl es üblich ist, `i` als Namen für die Variable zu verwenden, die den aktuellen Wert während der Iteration hält, kann dafür tatsächlich jeder beliebige Name verwendet werden.

```
liste = [1, 2, 3, 4, 5]

for i in liste:
    print(i)
```

```
1
2
3
4
5
```

```
alle_jahre = [2022, 2023, 2024]

for jahr in alle_jahre:
    print(f"Frohes neues Jahr {jahr}!")
```

```
Frohes neues Jahr 2022!
Frohes neues Jahr 2023!
Frohes neues Jahr 2024!
```

Hinweis: Was es mit dem `f` vor dem String und den geschweiften Klammern auf sich hat, wird in einem späteren Kapitel erklärt.

while

Der `while`-Loop wird benutzt, um Code so lange auszuführen, wie eine Bedingung erfüllt ist. Der Code, der bei jeder Iteration ausgeführt werden soll ist auch wieder eingerückt. Es ist wichtig, dass die Bedingung irgendwann nicht mehr erfüllt ist, da der Loop sonst unendlich lange ausgeführt wird. Meines Erachtens wird der `while`-Loop im Bereich Data Analytics nur sehr selten genutzt, soll hier aber wenigstens einmalig vorgestellt werden. Hier wird also zu Beginn Variable `x` auf 0 gesetzt. Im Loop wird der Wert der Variable geprintet und dann um 1 erhöht. Der Loop wird solange ausgeführt, bis `x` größer oder gleich 5 ist.

```
x = 0

while x < 5:
    print(x)
    x += 1
```

```
0
1
2
```

```
3
4
```

List Comprehension

List Comprehension ist eine Technik in Python, die zumindest für spezifische Anwendungsfälle die Nutzung von Loops vereinfachen kann - nämlich wenn Listen erstellt werden sollen. Die Syntax ist dabei sehr kompakt und besteht aus einer Kombination von eckigen Klammern und einem Loop. Als Beispiel wollen wir hier folgendes einmal mit einem Standard Loop und einmal mit List Comprehension tun: Für die Werte 1-5 sollen die Quadratzahlen berechnet werden.

```
# Liste mit Loop
x = range(1, 6)
y = []

for i in x:
    y.append(i**2)

print(x)
print(y)
```

```
range(1, 6)
[1, 4, 9, 16, 25]
```

```
# Liste mit List Comprehension
x = range(1, 6)
y = [i**2 for i in x]

print(x)
print(y)
```

```
range(1, 6)
[1, 4, 9, 16, 25]
```

Es wird direkt offensichtlich, dass der Code mit List Comprehension deutlich kürzer ist. Beim Betrachten des Codes wird auch klar, dass die List Comprehension recht intuitiv ist. Die eckigen Klammern geben an, dass eine Liste erstellt wird. Dies allein spart uns schon den Schritt vorher eine leere Liste `y` vorzubereiten, die dann im Loop mittels `.append()` befüllt werden kann. Innerhalb der eckigen Klammern steht dann der Loop, der die Werte für die Liste generiert. In diesem Fall ist es `for i in x`, also derselbe Ausdruck wie beim Standardloop. Anstatt aber noch eine Zeile zu benötigen,

um die Liste zu erstellen, wird hier direkt vor dem Loop die Berechnung für die Liste angegeben.

List Comprehension kann aber sogar noch mehr. So können auch Bedingungen in die List Comprehension eingebaut werden. Hier wird z.B. eine Liste erstellt, die nur die positive Zahlen aus einer anderen Liste enthält. Der Ansatz ist wie eben, nur das `i` nicht quadriert wird, aber stattdessen nach dem Loop noch ein `if` Ausdruck ergänzt wurde.

```
x = [1, -2, 3, -4, 5]
y = [i for i in x if i > 0]

print(x)
print(y)
```

```
[1, -2, 3, -4, 5]
[1, 3, 5]
```

Die generelle Schreibweise einer List Comprehension ist also:

```
neue_liste = [AUSDRUCK for ELEMENT in SAMMLUNG if BEDINGUNG == True]
```

Dabei steht

- AUSDRUCK für die Berechnung, die für jedes Element in der Liste durchgeführt werden soll.
- ELEMENT für das aktuelle Element in der Liste.
- SAMMLUNG für die Liste (oder Tuple, Set, Dictionairy, …)[1], über die iteriert werden soll.
- BEDINGUNG für eine Bedingung, die das aktuelle Element erfüllen muss, um in die neue Liste aufgenommen zu werden.

Dabei ist der AUSDRUCK flexibler als man vielleicht denken mag. Die beiden vorangegangenen Beispiele haben bisher nur das ELEMENT selbst (`i`) zurückgegeben, oder es aber quadriert (`i**2`). Tatsächlich kann aber auch schlichtweg ein Wert zurückgegeben werden, der nichts mit dem ELEMENT zu tun hat. So könnte z.B. auch `42` zurückgegeben werden, wenn das ELEMENT größer als 0 ist. Oder aber man schreibt in den AUSDRUCK selbst noch eine `if-else` Bedingung hinein:

[1] Streng genommen muss es sich bei der SAMMLUNG nicht zwangsläufig um eine Sammlung im herkömmlichen Sinn handeln. Entscheidend ist, dass das Objekt die Methoden `__iter__` und `__next__` implementiert, wodurch es das Iterator-Protokoll erfüllt. Dies ermöglicht es der for-Schleife, über das Objekt zu iterieren. Viele der eingebauten Datentypen in Python, wie Listen, Tupel, Sets und Dictionaries, erfüllen diese Anforderung. Ein praktisches Beispiel hierfür ist die Verwendung von range(5), das zwar keine Sammlung von Zahlen im Speicher anlegt, aber dennoch iterierbar ist, da es die erforderlichen Methoden implementiert.

```
x = [1, -2, 3, -4, 5]
y = [42 for i in x]

print(x)
print(y)
```

```
[1, -2, 3, -4, 5]
[42, 42, 42, 42, 42]
```

```
x = [1, -2, 3, -4, 5]
y = [i if i < 0 else i*100 for i in x]

print(x)
print(y)
```

```
[1, -2, 3, -4, 5]
[100, -2, 300, -4, 500]
```

Weitere Ressourcen

- Python Tutorial deutsch 10/24 - Die if Anweisung
- Python Tutorial deutsch 11/24 - if-Anweisung mit elif- und else-Zweigen erweitern
- Python Tutorial deutsch 16/24 - Die for Schleife
- Python Tutorial deutsch 17/24 - for Schleife als Zählerschleife verwenden
- Python Tutorial #16 | List Comprehension | Deutsch

Übungen

Was wird geprintet?

```
if 2+5==7:
    print("Data")
print("Analytics")
```

- (A) Data
- (B) Data Analytics
- (C) Analytics
- (D) nichts

Was wird geprintet?

```
if True or False:
    print("Data")
```

- (A) Data
- (B) nichts

Welchen Wert hat `x` am Ende?

```
x = 25

if x > 15:
    x = 10
elif x < 20:
    x = 5
else:
    x = 0

while x < 20:
    x += 1

x = x**2
```

x ist ___

Schreibe für alle List Comprehensions, die oben gezeigt wurden auch Code, der das selbe Ergebnis mit einem Standard Loop erzielt.

- (A) Geschafft

Nutze List Comprehension um basierend auf dieser Liste `zahlen = list(range(1, 21))` eine neue Liste zu erstellen, die nur gerade Zahlen enthält. Hinweis: Um zu prüfen ob eine Zahl `i` gerade ist, kann `i % 2 == 0`[2] als BEDINGUNG genutzt werden.

- (A) Geschafft

Nun soll die List Comprehension aus der vorigen Übung noch erweitert werden. Anstatt einfach die geraden Zahlen zu behalten, sollen die Wurzeln aus den geraden Zahlen in die neue Liste eingehen.

- (A) Geschafft

Schließlich soll die List Comprehension aus der vorigen Übung nochmals erweitert werden. Es sollen weiterhin die Wurzeln gezogen werden, allerdings soll dann noch 3

[2]Die Bedingung `i % 2 == 0` wird verwendet, um zu überprüfen, ob eine Zahl i gerade ist. In der Mathematik bedeutet der Operator `%` Modulo und gibt den Rest einer Division zurück. Wenn eine Zahl durch 2 geteilt wird und der Rest 0 ist, bedeutet dies, dass die Zahl ohne Rest durch 2 teilbar ist, was sie zu einer geraden Zahl macht. Anders gesagt, gerade Zahlen sind immer vollständig in Zweier-Schritte teilbar, was durch `i % 2 == 0` überprüft wird.

addiert werden. Außerdem soll das nicht für alle geraden Zahlen gemacht werden, sondern nur für die, die größer als 10 sind. Die BEDINGUNG enthält also zwei Bedingungen, die beide erfüllt sein müssen.

- (A) Geschafft