Bisherige Themen

Letzte Woche haben wir folgende Themen behandelt:

- In VSCode mehrere Cursors auf einmal zu setzen
- Mit `for` Befehle in Schleifen zu verpacken, die pro Element einer Sequenz einmal ausgeführt werden
- Mit `if`, `elif`, `else` Befehle nur unter bestimmten Bedingungen auszuführen
- Verschiedene Operationen auf Strings anzuwenden
- Variablenwerte während der Laufzeit eines Programms zu überschreiben

Falls Sie Fragen zu diesen Themen oder zu den Übungsaufgaben haben, sprechen Sie uns bitte an!

Und so geht es weiter:

Editor-Trick des Tages: Zeilen verschieben, Zeilen verdoppeln

In VSCode können Sie mit der Tastenkombination `Alt + ArrowUp/ArrowDown` die Zeile, in der der Cursor sich befindet, nach oben bzw. unten verschieben! Probieren Sie es aus: Kopieren Sie die folgenden Zeilen in Ihren Editor und sortieren Sie sie aufsteigend, indem Sie nur `Alt` und die Pfeiltasten verwenden.

```
1
5
2
8
4
```

Wenn Sie stattdessen `Alt + Shift + ArrowUp/ArrowDown` drücken, wird die Zeile, in der der Cursor sich befindet, verdoppelt. Probieren Sie es aus: Kopieren Sie die folgenden Zeilen und verwenden Sie das Tastenkürzel, um den vollständigen Liedtext zu erzeugen.

```
We're up all night 'til the sun
We're up all night to get some
We're up all night for good fun
We're up all night to get lucky
```

Indexing und Slicing

Sowohl Strings als auch Listen sind Sequenzen, die aus einzelnen Elementen bestehen. Erinnern Sie sich an die `for`-Schleife, bei der wir jedes einzelne Element einer Liste verarbeitet haben. Auch durch Strings können wir mit so einer Schleife **iterieren**. Klicken Sie in das Kästchen unten und drücken Sie `Ctrl + Enter`, um das Beispiel auszuführen!

In [ ]:

```
# Wir verwenden hier c als Namen für das Element, das
# in jedem Durchlauf der Schleife einen neuen Wert
# hat. Wofür könnte "c" stehen?
for c in "superlanger String mit mehreren Wörtern":
    print(c)
```

Um ohne Schleife auf ein einzelnes Element der Sequenz zuzugreifen, können wir die Position in der Sequenz ansteuern. Dazu beginnen wir links beim ersten Element mit `0` und zählen nach rechts weiter. Den **Index**, den wir verwenden wollen, schreiben wir in eckige Klammern nach der Sequenz:

In [ ]:

```
# Welches Zeichen steht an Index 5?
print("kurzer String"[5])
```

Wenn man vom Ende der Sequenz zählen möchte, verwendet man negative Zahlen: `seq[-1]` ist der letzte Index einer Sequenz, `seq[-2]` der vorletzte usw. Jedes Element der Sequenz kann also mit einem positiven Index oder mit einem negativen Index ausgewählt werden.

In [ ]:

```
print("ABCDE"[0])
print("ABCDE"[-1])
print("ABCDE"[3])
print("ABCDE"[-2])
```

Noch nützlicher ist das **Slicing**, wobei wir Teilsequenzen aus der ursprünglichen Sequenz extrahieren können. Wir geben dazu den Startindex und den Endindex an, in der Form `seq[start:end]` .

In [ ]:

```
print("ABCDE"[0:1])
print("ABCDE"[0:3])
print("ABCDE"[0:0])    # leer :(
print("ABCDE"[-1:-3])  # leer :(
print("ABCDE"[-3:-1])
```

Der Startindex ist dabei Teil der extrahierten Sequenz, während der Endindex **außerhalb** der Teilsequenz liegt. Beachten Sie auch, dass Sie beim Verwenden negativer Indizes genau wie bei positiven Indizes die kleinere Zahl zuerst angeben müssen (siehe Beispiel oben).

Wenn die Teilsequenz am Anfang der ursprünglichen Sequenz beginnen oder am Ende der Sequenz enden soll, können Sie diesen Index einfach weglassen.

In [ ]:

```
print("ABCDE"[:3])
print("ABCDE"[2:])
print("ABCDE"[:])
```

Aufgabe

1. Vergewissern Sie sich durch Ausprobieren, dass das Slicing-Verhalten von Listen parallel zum Verhalten von Strings ist.

In [ ]:

```
meine_liste = [1,2,3,4,5]
mein_string = "12345"

# Ihre Tests:
```

Mutability

Zahlen und Strings sind in Python **immutable** (unveränderlich). Das bedeutet, dass bei den Methoden z.B. für Strings als Ergebnis immer eine Kopie des Strings erzeugt wird. Der ursprüngliche String bleibt immer so, wie er definiert wurde. Wir können aber den Wert einer Variable manuell überschreiben, damit der String, der in der Variable gespeichert ist, sich ändert:

In [ ]:

```
my_string = "Wer jetzt kein Haus hat, baut sich keines mehr"
print(my_string)
print(type(my_string))

print("+++")

# split() zerteilt den String, aber das Ergebnis
# wird nicht ausgegeben oder gespeichert
my_string.split()
print(my_string)
print(type(my_string))

print("+++")

# Ergebnis der Operation in die Variable speichern
# ACHTUNG: der Datentyp ändert sich!
my_string = my_string.split()
print(my_string)
print(type(my_string))
# !!! Diese Veränderung des Typs ist schlechter Stil !!!
```

Operationen auf Listen

Im Gegensatz dazu können Listen verändert werden, ohne dass man extra eine Kopie erstellen muss. **Achtung:** Diese Operationen haben daher keinen Rückgabewert. Wir dürfen also nicht den Wert der Variable überschreiben!

In [ ]:

```
my_list = [1,2,3]
print(my_list)

my_list.append(4)   # das Element 4 an die Liste anhängen
print(my_list)      # :)

my_list = my_list.append(5)
print(my_list)      # :(
```

Die folgende Tabelle enthält einige Operationen für Listen, die für uns wichtig sind.

Operation	Auswirkung
`l.append(element)`	Das `element` wird ans Ende der Liste angehängt.
`l1.extend(l2)`	Ergänzt die Liste `l1` um alle Elemente von `l2` in der originalen Reihenfolge.
`l.insert(i, element)`	An der Position `i` in der Liste wird das `element` eingefügt.
`l.remove(element)`	Entfernt das erste Vorkommen von `element` aus der Liste (schlägt fehl, falls das Element nicht in der Liste enthalten ist).
`l.reverse()`	Dreht die Liste um.
`l.sort()`	Sortiert die Liste. Schlägt fehl, wenn Sortierung nicht möglich ist (z.B. wenn unterschiedliche Datentypen in der Liste enthalten sind).
`l.pop()`	Entfernt das letzte Element der Liste. Außerdem **Rückgabe des letzten Elements**. Schlägt fehl, wenn die Liste leer ist.
`l.pop(i)`	Entfernt das Element an Position `i` aus der Liste und **gibt es zurück**; schlägt fehl, falls die Position `i` in der Liste nicht existiert.
`element in l`	**Rückgabe: Bool** - `True`, wenn das `element` in der Liste enthalten ist; sonst `False`.
`min(l), max(l)`	**Rückgabe** des kleinsten bzw. größten Elements in der Liste.
`len(l)`	**Rückgabe** der Anzahl von Elementen in der Liste.
`l.count(element)`	**Rückgabe: Integer** - Anzahl der Vorkommen von `element` in der Liste.
`l.index(element)`	**Rückgabe: Integer** - Position des ersten Vorkommens von `element` in der Liste. Schlägt fehl, wenn das Element nicht in der Liste vorkommt.

Zum Schluss noch eine Operation, die als Stringoperation definiert ist, die wir aber meist in Kombination mit Listen verwenden: `join()`. So wird `join` verwendet:

In [ ]:

```
einzelne_woerter = ["Eis", "Schokolade", "Chips", "Kuchen"]
verbindungs_string = " +++ "

print(verbindungs_string.join(einzelne_woerter))
```

Datentyp des Tages: Dictionarys

Dictionarys ermöglichen es uns, Informationen zueinander in Beziehung zu setzen. Zum Beispiel interessiert uns in der Computerlinguistik oft, **wie häufig** bestimmte Wörter/Phrasen/Zeichen/... in einem Text vorkommen.

Dafür erstellen wir ein Dictionary, dessen **Keys** die Strings sind, die uns interessieren, und dessen **Values** die Häufigkeit jedes Strings im Text angeben.

In [ ]:

```
frequencies = {"und": 12, "oder": 17, "nicht": 5}
print(frequencies)
```

Ab der Pythonversion 3.7 behalten Dictionarys immer die Reihenfolge, in der die Elemente eingefügt wurden. In früheren Pythonversionen können wir uns nicht darauf verlassen, dass das so ist. Das spielt dann eine Rolle, wenn wir z.B. mit einer `for`-Schleife durch das Dictionary iterieren wollen.

Hier sehen Sie, dass Python Dictionarys für identisch hält, wenn...

1. sie die gleichen Keys enthalten
2. und dabei den Keys jeweils die gleichen Werte zugeordnet werden.

In [ ]:

```
freq1 = {"dog": 3, "cat": 5}
freq2 = {"cat": 5, "dog": 3}
print(freq1 == freq2)
```

Die Informationen, die in einem Dictionary gespeichert sind, können wir abrufen, indem wir den Key angeben, der uns interessiert. Die Rückgabe ist dann der Wert, der zu diesem Key im Dictionary gespeichert ist. Die Syntax dafür ist `<dictionary>[key]`.

In [ ]:

```
frequencies = {"und": 12, "oder": 17, "nicht": 5}
print(frequencies["und"])
```

Ist ein Key nicht vorhanden, tritt ein Fehler auf:

In [ ]:

```
frequencies = {"und": 12, "oder": 17, "nicht": 5}
print(frequencies["weil"])
```

Dictionarys sind, genau wie Listen, ein veränderlicher Datentyp (**mutable**). Um einen Wert ins Dictionary einzufügen, schreiben wir:

In [ ]:

```
frequencies = {"und": 12, "oder": 17, "nicht": 5}

print(frequencies)          # bisheriger Inhalt des Dictionarys
frequencies["weil"] = 15    # dem Schlüssel "weil" soll jetzt der Wert 15 zugeordnet
                            # werden
print(frequencies)          # Dictionary mit zusätzlichem Schlüssel-Wert-Paar
```

Achtung: Jeder Schlüssel kann in einem Dictionary nur genau einmal vorkommen. Es gilt immer der zuletzt definierte Wert für den Schlüssel.

Und schließlich können wir auch durch Dictionarys iterieren. Die `for`-Schleife beginnt fast genauso wie bei Listen. Das aktuelle Element ist immer ein Schlüssel aus dem Dictionary. Wir können also beispielsweise alle Schlüssel-Wert-Paare nacheinander ausgeben:

In [ ]:

```
frequencies = {"und": 12, "oder": 17, "nicht": 5, "weil": 15}

for word in frequencies:
    print("Wort: " + word)
    print("Häufigkeit: " + str(frequencies[word]))
```

Als Schlüssel für Dictionarys sind nur **unveränderliche Datentypen** erlaubt. Meistens benutzen wir Strings oder Zahlen. Listen sind als Werte möglich, aber nicht als Keys:

In [ ]:

```
freq3 = {"yes": [1,2,3], "no": [4,5,6]}
print(freq3)
```

In [ ]:

```
freq4 = { [1,2]: "yes", [4,5]: "no"}      # :(
```

Für das Beispiel, in dem Worthäufigkeiten gezählt werden sollen, brauchen wir Strings für die Wörter und Integers für die Häufigkeiten. Als Schlüssel sollten wir die Wörter wählen.

Es gibt mehrere Möglichkeiten, die Inhalte eines Dictionarys explizit zu **sortieren**: Entweder nach der Default-Sortierreihenfolge der Keys, oder nach der Sortierreihenfolge der Values.

Aufgabe (optional)

1. Führen Sie die folgenden `for`-Schleifen aus und finden Sie heraus, welche Sortierung jedes Codebeispiel erzeugt.

In [ ]:

```
frequencies = {"und": 12, "oder": 17, "nicht": 5, "weil": 15}

print(frequencies)
for word in sorted(frequencies):                        # !!!
    print(word + "\t" + str(frequencies[word]))
```

In [ ]:

```
frequencies = {"und": 12, "oder": 17, "nicht": 5, "weil": 15}

print(frequencies)
for word in sorted(frequencies, key=frequencies.get):  # !!!
    print(word + "\t" + str(frequencies[word]))
```

In [ ]:

```
frequencies = {"und": 12, "oder": 17, "nicht": 5, "weil": 15}

print(frequencies)
for word in sorted(frequencies, key=frequencies.get, reverse=True): # !!!
    print(word + "\t" + str(frequencies[word]))
```

Wenn wir uns nur für die Keys oder nur für die Values in einem Dictionary interessieren, können wir darauf mit den folgenden Befehlen zugreifen.

In [ ]:

```
frequencies = {"und": 12, "oder": 17, "nicht": 5, "weil": 15}

print(frequencies.keys())
print(frequencies.values())
```

Die Ergebnisse dieser Befehle können wir in Listen umwandeln, damit wir Listenoperationen darauf anwenden können (z.B. Indexing und Slicing).

In [ ]:

```
frequencies = {"und": 12, "oder": 17, "nicht": 5, "weil": 15}

print(list(frequencies.keys())[:2])
```

Dabei müssen wir beachten, dass die Reihenfolge der Keys gegebenenfalls nicht vorhersehbar ist, weil Dictionarys ein unsortierter Datentyp sein können.

Zufallszahlen

Manchmal wollen wir zufällige Elemente aus einer Kollektion auswählen, z.B. weil nach dem häufigsten Wort in einem Text gesucht wird, es aber mehrere Wörter gibt, die mit der gleichen höchsten Häufigkeit vorkommen. Die zufällige Auswahl wird vom **Modul** `random` übernommen.

Module sind externe Pythonprogramme, die wir in unseren Code importieren können, um auf die Funktionen dort zugreifen zu können. Dazu schreiben wir **am Anfang unseres Codes**: `import <modulname>`

Danach können wir die Funktionen aus dem Modul verwenden. Damit der Interpreter weiß, wo die Funktionen definiert sind, schreiben wir bei jeder Verwendung auch den Namen des Moduls dazu.

Zum Beispiel geben wir eine zufällige Zahl zwischen 1 und 10 (beide inklusive) aus. Führen Sie die Zelle mehrmals aus (mit `Ctrl + Enter`), um zu sehen, dass jedes Mal eine andere Zahl gewählt wird:

In [ ]:

```
import random     # Modul random importieren
zahl = random.randint(1,10)  # Zufallszahl erzeugen
print(zahl)
```

Eine weitere Methode aus dem `random` -Modul, die wir benutzen, ist `random.choice()` . Damit wird ein zufälliges Element aus einer Kollektion gewählt. Führen Sie das Kästchen mehrfach aus und beobachten Sie, dass jedesmal ein anderes Element ausgegeben wird!

In [ ]:

```
import random

words = ["Since", "I", "left", "you", "mine", "eye", "is", "in", "my", "mind"]
zufallswort = random.choice(words)
print(zufallswort)
```

Zusammenfassung

Sie haben heute gelernt,

- Auf bestimmte Positionen oder Bereiche von sortierten Sequenzen (Strings, Listen) zuzugreifen
- Zwischen veränderlichen und unveränderlichen Datentypen (mutable/immutable) zu unterscheiden
- Verschiedene Operationen auf Listen anzuwenden
- Dictionarys zu erstellen
- Die Werte aus Dictionarys abzufragen oder neu hinzuzufügen
- Die Wertpaare in einem Dictionary nach den Keys oder Values sortiert zu verarbeiten
- Zufallszahlen zu erzeugen

Und morgen...

In der Übung am 23.10. werden Sie üben, mit Indexing und Slicing beliebige Teilsequenzen zu erzeugen. Sie werden sich außerdem intensiv mit Dictionarys beschäftigen.