Editor Trick des Tages: `Expand Selection`

Oft finden Sie sich beim Programmieren in einer recht komplexen (Klammer)-Struktur wieder und wollen Teile der Struktur an anderer Stelle bearbeiten.

In [4]:

```
my_complex_structure = ["I", ["saw", ["the", "man"],["with", ["the", "telescope"]]]]
```

In VS Code können Sie mit Hilfe von `Expand Selection` von der aktuellen Position Ihres Cursors ausgehend ihre Auswahl auf die jeweils nächste größere Struktureinheit erweitern. Damit lassen sich schnell auch große Sinneinheiten auswählen.
Sie finden den Befehl in der Command Palette, aber viel nützlicher wird der Befehl, wenn man ihn mit dem Tastatur Short-Cut anwendet:

`Shift, Alt + ->/<-`

So können Sie Ihre Auswahl schnell erweitern und auch wieder verringern.

Datentype des Tages: Tupel

Das Tupel ist ein weiterer Sequenz-Datentyp in Python, die Listen sehr ähnlich sind. Sie sind eine Sammlung von Objekten, die verschiedenen Typs sein können. Einzelne Tupelelemente können per Index aufgerufen werden.

In [1]:

```
my_tuple_1 = ("schlafen", "Verb", "intransitiv")
print(my_tuple_1[0])
```

Slicing ist auch bei Tupeln möglich.

In [1]:

```
my_tuple_1 = ("schlafen", "Verb", "intransitiv")
my_tuple_2 = my_tuple_1[0:2]
print(my_tuple_2)
```

ABER anders als Listen sind Tupel nicht veränderbar (non-mutable).

In [1]:

```
my_tuple_1 = ("schlafen", "Verb", "intransitiv")
my_tuple_1[1] = "Nomen"
```

Damit eignen sie sich gut für Datensammlungen, die nicht verändert werden sollen. Sie können als Keys von Dictionary genutzt werden. Wie alle anderen nicht-veränderbaren Datentypen lassen sich Tupel schneller verarbeiten als veränderbare Datentypen.

Werkzeug des Tages: Comprehensions

Oft benutzen wir Schleifen um ein(e) Liste, Dictionary oder Set mit Werten zu füllen. Im Beispiel unten erstellen wir erst eine Liste mit den Zahlen von 0 - 9 und anschließend eine Liste mit den geraden Zahlen im gleichen Intervall. Beachten Sie: Der Operator "%" führt eine Ganzzahldivision (Modulo) durch und gibt den Restwert zurück: 5 % 2 = 1

In [1]:

```
my_list_1 = []
for n in range(10):
  my_list_1.append(n)

print(my_list_1)
print("------------------")

my_list_2 = []
for n in range(10):
  if n % 2 == 0:
    my_list_2.append(n)

print(my_list_2)
```

List Comprehensions

Python bietet eine weitere Möglichkeit, diese Art von Aufgabe effizient zu erledigen: Comprehensions.

Die kurze Schreibweise unten erfüllt den selben Zweck, wie die Schleifen oben.

In [1]:

```
my_list_3 = [n for n in rang(10)]
print(my_list_3)

print("------------------")

my_list_4 = [n for n in range(10) if n % 2 == 0]
print(my_list_4)
```

Comprehension-Syntax

Die Syntax eine Comprehension funktioniert wie folgt:

- Die Klammern bestimmen den Typ des Outputs: [] für Listen, {} für Dictionarys, () für Sets:
  [i for i in range(1,20) if i % 2 == 0]
- Als nächstes bestimmen wir die Laufvariable der Comprehension:
  [i for i in range(1,20) if i % 2 == 0]
  - Sie bestimmt den Wert, der der Sequenz hinzufügt wird.
  - Auf der Laufvariable können auch zusätzliche Operationen ausgeführt werden (optional).
- der nächste Teil sieht aus wie eine gewöhnlicher Schleifenkopf:
  [i for i in range(1,20) if i % 2 == 0]
  - Der Laufvariablenname wird hier wieder aufgenommen und
  - die Quelle für den Wert der Laufvariable definiert.
- Der letzte Teil der Comprehension ist eine Bedingung, wie wir sie auch innnerhalb unserer Schleife oben benutzt haben, die jeweils für den Ausgangswert der Laufvariable überprüft wird (optional):
  [i for i in range(1,20) if i % 2 == 0]

Ein weiteres Beispiel mit Operation auf dem Laufvariablenwert:

In [1]:

```
my_list_5 = [10*n for n in range(10) if n % 2 == 0]
print(my_list_5)
```

Set Comprehensions

Set comprehensions ähneln von der Form her (bis auf die Klammern) den List comprehensions. Beachten Sie, dass Sets ungeordnet sind und jedes Element nur einmal enthalten ist:

In [1]:

```
set_of_characters = {c for c in "Dieser Text enthält viele Buchstaben mehrfach"}
print(set_of_characters)
```

Dict Comprehensions

Dictionary comprehensions sind nur minimal komplexer: Hier müssen wir für jedes Element der Sequenz einen passenden Key angeben. Zum Beispiel so:

In [1]:

```
woerter_und_wortlaengen = {wort: len(wort) for wort in "Dieser Satz besteht aus sechs W
örtern".split()}
print(woerter_und_wortlaengen)
```

Für alle Comprehensions gilt: Sie können eigene Funktionen schreiben und diese innerhalb der Comprehension auf den Laufvariablenwert anwenden:

In [1]:

```
def wort_sortieren(wort):
    sortiertes_wort = "".join(sorted(wort))
    return sortiertes_wort

sortierte_woerter = [wort_sortieren(w) for w in "Dieser Satz kein Verb".split()]
print(sortierte_woerter)
```

Aufgabe:

1. Können Sie durch Nachdenken herausfinden, welche Ergebnisse die folgenden Comprehensions liefern? Führen Sie die Beispiele aus, um zu prüfen, ob Ihre Vorhersage korrekt ist.

In [1]:

```
print([i % 2 for i in range(20)])
```

In [1]:

```
print({i % 2 for i in range(20)})
```

In [1]:

```
print({i: i % 2 for i in range(20)})
```

In [1]:

```
print([c.upper() for c in "kleinbuchstaben"])
```

Konzept des Tages: Rekursion

Rekursion ist ein mächtiges Werkzeug in der Programmierung, dass in in sich jedoch auch Gefahren bergen kann, falls ein Algorithmus unsauber programmiert ist. Wir bezeichnen eine Funktion als rekursiv, falls sie sich selbst aufruft.

In [1]:

```
def my_weird_recursive_function():
  return my_weird_recursive_function()

my_weird_recursive_function()
```

Die Funktion oben tut nichts anderes als einen Funktionsaufruf zurückzugeben. Dabei wird keine Alternative zu diesem Selbstaufruf bzw. eine andere Abbruchbedingung gegeben, sodass die Funktion nach einmaligem Aufruf theoretisch für immer so weiter laufen könnte. Glücklicherweise erkennt Python jedoch solche Probleme und gibt nach gewisser Zeit und einer bestimmten Anzahl an rekursiven Aufrufen eine Fehlermeldung aus.

Anwendungsbeispiele

Aus der Mathematik:

Die Berechnung der Fakultät einer natürlichen Zahl, *n!*, zum Beispiel *4!= 4*3*2*1*, lässt sich gut mit Hilfe eines rekursiven Algorithmus berechen. Dabei unterscheidet man zunächst Fälle:

- Entweder Entweder ist *n=0*. Dann gilt: *n!=1*.
- Oder n ist eine positive ganze Zahl. Dann gilt: *n!=n·(n-1)!*

Die Fakultät von n ist also das Produkt von n und der Fakultät von n-1.Entweder ist n=0. Dann gilt: n!=1. Das ist der **Rekursionsanfang**. Oder n ist eine positive ganze Zahl. Dann gilt: n!=n·(n-1)!, die Fakultät von n ist also das Produkt von n und der Fakultät von n-1. Dieser zweite Fall ist der **Rekursionsschritt**. Über die Unterscheidung zwischen diesen beiden Fällen Rekursionsschritt und Rekursionsanfang lässt sich so jede beliebige Fakultät berechnen.

In [3]:

```
def fac(n):
    if n == 0:
        return 1
    else:
        return n * fac(n-1)

print(fac(4))
```

24

Die Funktion oben setzt die beschriebene Methode zur Berechnung der Fakultät um. Durch den wiederholten Aufruf werden nach und nach alle Faktoren bis zum Rekursionsanfang bereitgestellt. Dann kann das Produkt ganz leicht berechnet werden.

Rekursive Datenstrukturen:

Einer der wichtigen Einsatzzwecke für Rekursion ist das Verarbeiten von rekursiven Datenstrukturen. Eine rekursive Datenstruktur ist eine Datenstruktur, die wiederum Datenstrukturen desselben Typs enthalten kann. Python-Listen sind zum Beispiel rekursive Datenstrukturen, da sie wiederum Python-Listen enthalten können.

In [1]:

```
my_recursive_list_1 = ["I", ["saw", ["the", "man"],["with", ["the", "telescope"]]]]
print(my_recursive_list_1)
```

Nehmen wir an wir möchten aus einer Struktur wie oben, auschließlich Strings ausgeben lassen.

In [1]:

```
def print_recursive_structures(some_list):
  for element in some_list:
    if type(element) is str:
      print(element)
    elif type(element) is list:
      print_recursive_structures(element)
    else:
      print("Sorry, I can't do that Dave")

print(print_recursive_structures)
```

Die Funktion oben löst die Aufgabe indem sie zunächst prüft, ob das aktuelle Listenelement den richtigen Typ hat und gibt entweder einen String aus oder startet einen Rekursionsschritt.

Zusammenfassung

Sie haben heute gelernt,

- dass der Datentyp Tuple die nicht-veränderbare Version von List ist.
- wie man Daten kompakt und effizient mit Comprehensions bearbeiten und kann.
- wo man in Python Rekursion findet und wie man das Konzept anwenden kann.