Iterables

Iterables sind Datentypen, die mehrere Elemente haben. Über diese Datentypen kann iteriert werden, das heißt die Elemente können nacheinander durchlaufen werden, zum Beispiel in einer `for` -Schleife. Wir kennen bereits Strings als einen solchen Datentyp, es gibt allerdings noch weitere: Listen, Tuples und Dictionaries. Diese drei Datentypen können als Elemente wiederum jeden beliebigen Datentyp enthalten. Jedes Element in einem solchen Datentyp erhält einen Index, über den es später wieder aufgerufen werden kann. Dabei gibt es zwei Sorten von Indizes: positionell und keyword.

Listen

Listen enthalten mehrere Werte, die in einer Variable gespeichert werden. Eine Liste wird mit eckigen Klammern geschrieben und die einzelnen Elemente werden mit Kommata voneinander getrennt.

In [1]:

```
1 liste_1 = [1, 2, 3, 4, 5]
2
3 print(liste_1)
```

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

Listen speichern ihre Elemente positionell ab, wenn man also auf ein bestimmtes Element der Liste zugreifen will, kann man dies über seine Position in der Liste als Index tun. Diese Positionen fangen bei `0` an für das erste Element. Der Index wird nach der Liste in eckigen Klammern angegeben. Ein zu großer Index wirft einen Fehler.

In [6]:

```
1 print(liste_1[0])  # Index == 0, also 1. Element
```

```
1
```

In [5]:

```
1 print([1, 2, 3][2])  # Liste als Literal, Index == 2, also 3. Element
```

```
3
```

In [21]:

```
1 fruits = ['apple', 'banana', 'pear', 'tomato']  # 4 Elemente, größter Index ist also 3
2
3 print(fruits[3])  # tomato
4 print(fruits[4])
```

```
tomato

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-21-a66f4323ed1c> in <module>
      2
      3 print(fruits[3])  # tomato
----> 4 print(fruits[4])

IndexError: list index out of range
```

Wichtig: Die Fehlermeldung verrät uns direkt, was für ein Fehler vorliegt, und in welcher Zeile der Fehler aufgetreten ist!

In Python lassen Listen auch negative Indizes zu. Dabei wird am Ende der Liste angefangen: `-1` ist also das letzte Element in der Liste, `-2` das vorletzte und so weiter. Auch hier kann es zu einem **`IndexError`** kommen, wenn der Index zu klein gewählt wird.

In [9]:

```
1 print(fruits[-1])  # tomato
2 print(fruits[-4])  # apple
3 print(fruits[-5])
```

```
tomato
apple

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-94dc3d7252f2> in <module>
      1 print(fruits[-1])  # tomato
      2 print(fruits[-4])  # apple
----> 3 print(fruits[-5])

IndexError: list index out of range
```

Über den Index können Elemente der Liste durch Neuzuweisung auch verändert werden.

In [22]:

```
print(fruits)

fruits[0] = 'orange'

print(fruits)
```

```
['apple', 'banana', 'pear', 'tomato']
['orange', 'banana', 'pear', 'tomato']
```

Um ein Element aus einer Liste zu entfernen, gibt es zwei Möglichkeiten: `del` und `remove`. `del` ist ein keyword, welches eine Variable oder, wie in diesem Fall, ein Teil einer Variablen aus dem Speicher löscht. `remove` ist eine Funktion von Listen, Tuples und Dicts, die den spezifizierten Wert aus der Liste löscht. Wenn dieser Wert mehrfach vorhanden ist, wird nur der erste Wert, also derjenige mit dem kleinsten Index, gelöscht.

In [23]:

```
del fruits[3]
print(fruits)
```

```
['orange', 'banana', 'pear']
```

In [32]:

```
fruits.remove('banana')
print(fruits)
```

```
['orange', 'pear']
```

In [24]:

```
liste_2 = [1, 2, 3, 1, 4, 5, 1, 6]

print(liste_2)

liste_2.remove(1)

print(liste_2)
```

```
[1, 2, 3, 1, 4, 5, 1, 6]
[2, 3, 1, 4, 5, 1, 6]
```

> ***Einschub: Punkt-Notation bei Funktionen***
>
> Die `remove`-Funktion wird auf einer Liste aufgerufen, indem die Funktion mit einem Punkt `.` an die Liste bzw. die Variable angehängt wird. Dieser Punkt bedeutet, dass die Funktion ein Teil der Variablen ist. In diesem Fall ist die `remove`-Funktion ein Teil des Datentyps Liste und kann deshalb auf jeder Liste aufgerufen werden. Selbiges kennen wir zum Beispiel auch bereits von Strings mit den Funktionen `.lower()` und `.upper()`. Diese Funktionen sind Bestandteil des Datentyps String.

Um eine leere Liste zu erzeugen kann man einfach zwei eckige Klammern ohne Elemente schreiben. Eine andere Möglichkeit ist der Aufruf `list()`. Ohne Argument erzeugt dies einfach eine leere Liste. Alternativ kann auch ein Argument übergeben werden: bei dem Argument muss es sich dabei um einen iterierbaren Wert handeln, der dann in eine Liste umgewandelt wird.

In [30]:

```
a = []
b = list()
c = list([1, 2, 3])
d = list('Hallo')

print(a)
print(b)
print(c)
print(d)
```

```
[]
[]
[1, 2, 3]
['H', 'a', 'l', 'l', 'o']
```

Mit der `.append()` Funktion kann ein neues Element an eine Liste angehängt werden. Mit der `.insert(i, elem)` Funktion kann ein Element `elem` beim Index `i` in eine Liste eingefügt werden.

In [38]:

```
veggies = ["cucumber", "onion"]

print(veggies)

veggies.append("zucchini")

print(veggies)

veggies.insert(2, "pumpkin")  # Index == 2, also 3. Element

print(veggies)
```

```
['cucumber', 'onion']
['cucumber', 'onion', 'zucchini']
['cucumber', 'onion', 'pumpkin', 'zucchini']
```

Wie bereits erwähnt können for-Schleifen über Listen laufen. Dabei werden die Elemente der Liste in ihrer Reihenfolge nacheinander durchlaufen. Die einzelnen Elemnte werden dabei einer temporären Variable zugewiesen, die aber dann keinen Bezug mehr auf die Liste hat. Wenn diese temporäre Variable verändert wird, ändert das **nicht** die Liste. Eine andere Möglichkeit ist die `len()` Funktion. Diese nimmt als Argument ein iterierbares Objekt und gibt die Anzahl der Elemente zurück. Dabei denken wir wieder daran: der Index fängt bei `0` an, die Länge einer Liste ist also immer um eins größer als der größte Index. Daher eignet sich die Länge gut für eine `range()`.

In [39]:

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

for elem in l:
    print(elem)
```

```
1
2
3
4
5
```

In [44]:

```
l2 = [1, 2, 3, 4, 5]
print(l2)

for elem in l2:
    elem += 5  # Liste wird nicht verändert
    print(elem)

print(l2)
```

```
[1, 2, 3, 4, 5]
6
7
8
9
10
[1, 2, 3, 4, 5]
```

In [41]:

```
l3 = [1, 2, 3, 4, 5]
print(l3)

for i in range(len(l3)):  # i als Index
    l3[i] += 5  # Liste wird verändert

print(l3)
```

```
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
```

Tuples

Tuples funktionieren im Prinzip wie Listen, nur dass sie *immutable* sind. Das heißt, dass sie nicht veränderbar sind. Tuples werden mit runden Klammern geschrieben. Wie bei Listen kann über den Index auf einzelne Elemente eines Tuples zugegriffen werden, allerdings können diese nicht überschrieben werden. Ebenso kann an einen Tuple kein neues Element angehängt werden. Nach der initialen Zuweisung kann ein Tuple **nicht** mehr verändert werden.

In [45]:

```
t = (1, 2, 3)

print(t)

print(t[0])

for elem in t:
    print(elem)
```

```
(1, 2, 3)
1
1
2
3
```

In [47]:

```
t[1] = 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-c53026d12066> in <module>
----> 1 t[1] = 5

TypeError: 'tuple' object does not support item assignment
```

In [48]:

```
t.append(4)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-48-ada7ed8a579e> in <module>
----> 1 t.append(4)

AttributeError: 'tuple' object has no attribute 'append'
```

In [49]:

```
t.remove(3)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-49-e105b2f059c5> in <module>
----> 1 t.remove(3)

AttributeError: 'tuple' object has no attribute 'remove'
```

In [50]:

```
1 del t[2]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-2d0f41a77003> in <module>
----> 1 del t[2]

TypeError: 'tuple' object doesn't support item deletion
```

> Selbiges tirfft übrigens auch auf Strings zu. Strings sind, wie Tuples, *immutable*.

In [52]:

```
1 s = "Hello World"
2
3 print(s[3])
4
5 s[3] = 'x'
```

```
l

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-52-906504e1269b> in <module>
      3 print(s[3])
      4
----> 5 s[3] = 'x'

TypeError: 'str' object does not support item assignment
```

In [53]:

```
1 s.remove('H')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-53-ca78b73fb3fb> in <module>
----> 1 s.remove('H')

AttributeError: 'str' object has no attribute 'remove'
```

Um ein Tuple mit nur einem Element zu erzeugen, muss trotzdem ein Komma nach diesem Element gesetzt werden, bevor die schließende Klammer kommt. Da Klammern auch zur Priorisierung von Aktionen verwendet werden, wird ein Wert ohne Komma nur als einfacher Wert interpretiert. Leere Tuples können allerdings mit leeren Klammern erzeugt werden, da diese im Nachhinein jedoch nicht mehr gefüllt werden können, wird diese Funktion in der Regel nicht benötigt.

In [57]:

```
t1 = (2,)  # tuple mit einem Element
print(t1, type(t1))

t2 = (2)  # ohne Klammer ist es nur ein int
print(t2, type(t2))

t3 = ()
print(t3, type(t3))
```

```
(2,) Type: <class 'tuple'>
2 Type: <class 'int'>
() Type: <class 'tuple'>
```

Listen und Tuples können über die entsprechenden Aufrufe `list()` und `tuple()` in den jeweils anderen Typ umgewandelt werden.

In [58]:

```
t1 = (1, 2, 3)
l1 = list(t1)

l2 = [4, 5, 6]
t2 = tuple(l2)

print(t1, type(t1))
print(l1, type(l1))
print(t2, type(t2))
print(l2, type(l2))
```

```
(1, 2, 3) <class 'tuple'>
[1, 2, 3] <class 'list'>
(4, 5, 6) <class 'tuple'>
[4, 5, 6] <class 'list'>
```

Dictionaries

Auch Dictionaries enthalten mehrere Elemente, diese haben allerdings keinen positionellen Index, sondern einen keyword-Index. Dictionaries bestehen also aus mehreren (oder einem) key-value Paaren. Dictionaries werden mit geschweiften Klammern geschrieben, je Paar kommt zunächst der key, dann ein Doppelpunkt, und dann der value. Die einzelnen Paare werden durch Kommata voneinander getrennt. Einzelne Elemente des Dicts können über das entsprechende keyword als Index aufgerufen und verändert werden. Ebenso können mit einem neuen Index auch neue Paare hinzugefügt werden.

In [1]:

```
auto = {"brand": "Honda", "model": "Civic", "year": 2005, "color": "midnight blue", "do

print(auto)

print(auto['color'])

auto['color'] = "red"

print(auto['color'])

auto['convertible'] = False

print(auto)
```

```
{'brand': 'Honda', 'model': 'Civic', 'year': 2005, 'color': 'midnight blue',
'doors': 4}
midnight blue
red
{'brand': 'Honda', 'model': 'Civic', 'year': 2005, 'color': 'red', 'doors':
4, 'convertible': False}
```

Die keywords müssen keine Worte bzw. Strings sein. Deshalb wird oft auch einfach nur der Begriff key verwendet. Das ist auch der Grund, weshalb nur die Keys als Index verwendet werden können, und **nicht** die Positionen.

In [65]:

```
nums = {5: "five", 2: "two", 3: "three", 1: "one"}

print(nums[1])
```

```
one
```

Bei Dicts liefert die `len()` Funktion die Anzahl der key-value Paare. Möchte man nur die keys oder nur die values erhalten, kann man die Funktionen `.keys()` und `.values()` verwenden. In einer for-Schleife über ein Dict wird standardmäßig über die keys gelaufen. Mit der Funktion `.items()` werden die key-value Paare als Tuple zurück gegeben.

In [2]:

```
print(len(auto))
print(auto.keys())
print(auto.values())
print(auto.items())
```

```
6
dict_keys(['brand', 'model', 'year', 'color', 'doors', 'convertible'])
dict_values(['Honda', 'Civic', 2005, 'red', 4, False])
dict_items([('brand', 'Honda'), ('model', 'Civic'), ('year', 2005), ('colo
r', 'red'), ('doors', 4), ('convertible', False)])
```

In [103]:

```
for x in auto:
    print(x)
```

```
brand
model
year
color
doors
convertible
```

In [104]:

```
for x in auto.values():
    print(x)
```

```
Honda
Civic
2005
red
4
False
```

In [105]:

```
for x in auto:
    print(x, '-', auto[x])
```

```
brand - Honda
model - Civic
year - 2005
color - red
doors - 4
convertible - False
```

In [106]:

```
for x, y in auto.items():
    print(x, '-', y)
```

```
brand - Honda
model - Civic
year - 2005
color - red
doors - 4
convertible - False
```

> ***Einschub: Mehrere Rückgabewerte***
>
> In Python kann eine Funktion mehrere Werte gleichzeitig zurück geben, wie zum Beispiel die `.items()` Funktion bei Dicts. Diese Werte können direkt in mehrere Variablen gespeichert werden, die einfach durch Kommata voneinander getrennt werden. In einer for-Schleife wie im obigen Beispiel kann so auch direkt über mehrere Variablen gleichzeitig gelaufen werden.

Sets

Sets enthalten auch mehrere Elemente, sind aber ungeordnet und ohne keys. Das heißt, die Elemente können nicht über einen Index aufgerufen werden. Ebenfalls können Sets keine doppelten Elemente enthalten. Sets werden auch mit geschweiften Klammern geschrieben.

In [91]:

```
s = {'apple', 'banana', 'orange', 'apple'}  # das zweite 'apple' wird ignoriert
print(s)
```

```
{'apple', 'banana', 'orange'}
```

In [93]:

```
print(s[0])  # Indizes werfen einen Fehler
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-93-b165bd46c8fc> in <module>
----> 1 print(s[0])  # Indizes werfen einen Fehler

TypeError: 'set' object is not subscriptable
```

In [94]:

```
print('apple' in s)  # das 'in' keyword funktioniert aber noch
```

```
True
```

In [96]:

```
for x in s:  # eine for-Schleife kann auch über ein Set laufen
    print(x)
```

```
apple
banana
orange
```

Um einem Set ein neues Element hinzuzufügen, gibt es die `.add()` Funktion.

In [97]:

```
s.add('cherry')
print(s)
```

```
{'cherry', 'apple', 'banana', 'orange'}
```

Referenzvariablen

Wenn eine Liste o.ä. einer Variablen zugewiesen wird, handelt es sich dabei nur um eine Referenz auf die Liste. Um das besser zu verstehen, betrachten wir zunächst nochmal primitive Datentypen:

In [110]:

```
1 a = 5
2 b = a  # also b = 5
3 a += 2
4
5 print(a, b)
```

```
7 5
```

Wir haben in Zeile zwei gesagt, dass `b = a` ist. Danach haben wir `a` verändert, `b` hat aber den Wert 5 behalten. `a` und `b` sind also trotz der Gleichsetzung in Zeile 2 voneinander unabhängig. Das liegt daran, dass bei primitiven Datentypen der entsprechende Wert direkt in der Variable gespeichert wird.

Bei Listen o.ä. ist das etwas anders:

In [113]:

```
1 a = [1, 2, 3]
2 b = a
3 print(a, b)  # a und b sind identisch
4
5 a.append(4)  # a wird verändert, b nicht...
6 print(a, b)  # ...aber trotzdem hat auch b eine 4 am Ende
7
8 a = [5, 6, 7]  # dies ist eine komplette Neuzuweisung
9 print(a, b)  # deshalb sind a und b jetzt unterschiedlich
```

```
[1, 2, 3] [1, 2, 3]
[1, 2, 3, 4] [1, 2, 3, 4]
[5, 6, 7] [1, 2, 3, 4]
```

Bei Listen enthalten die Variablen nur einen Zeiger auf die tatsächliche Liste. Wenn also in Zeile 2 die Listen gleich gesetzt werden, enthalten beide Variablen denselben Zeiger auf dieselbe Liste. Deshalb ist jede Änderung an `a` auch eine Änderung an `b`. In Zeile 8 wird `a` eine neue Liste zugewiesen. Dadurch bekommt `a` einen neuen Zeiger auf diese neue Liste, `b` behält aber den alten Zeiger.

Mit der Funktion `id()` können wir uns anschauen, auf welche Speicherposition eine Variable zeigt:

In [116]:

```
a = 5
b = a
a += 2

c = [1, 2, 3]
d = c
c.append(4)

print(id(a))
print(id(b))
print(id(c))
print(id(d))
```

```
140715699156960
140715699156896
1871200754944
1871200754944
```

Diese Referenzeigenschaft von komplexen Datentypen ist sehr wichtig und muss immer im Hinterkopf behalten werden!

In [3]:

```
a = [1, 2, 3]
b = a.copy()

a.append(4)

print(a, b)
```

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

Einschub: `random.choice()`

Wir kennen bereits das Modul `random`. Bisher haben wir davon die Funktion `randint()` verwendet, um uns eine zufällige ganze Zahl zu erzeugen. Wenn wir ein zufälliges Element aus einer Liste brauchen, können wir natürlich einfach `randint()` verwenden, um uns damit einen zufälligen Index zu generieren. Alternativ bietet das `random` Modul aber auch die Funktion `choice()`. Diese gibt direkt ein zufälliges Element aus einer Liste zurück.

In [10]:

```python
import random

l = ['a', 'Hallo', 42, 3.14, True, [4, 5, 6], 'Welt']

for i in range(10):
    print(random.choice(l))
```

```
a
[4, 5, 6]
[4, 5, 6]
a
Hallo
[4, 5, 6]
3.14
True
Welt
42
```