4. Datenstrukturen in Python

Unterschied zwischen *Datenstrukturen* und *einfachen Datentypen*:

- Datenstrukturen sind zusammengesetzt aus Objekten *einfachen Datentyps* und damit wesentlich komplexer aufgebaut
- Sind die Objekte in einer Datenstruktur fortlaufend angeordnet, spricht man von *sequentiellen Datentypen*, z.B. String, Listen, Tupel, etc.
- Datenstrukturen besitzen neben ihren *Objekten* geeignete *Operatoren*, um so effizient wie möglich auf diese zugreifen zu können
- Es gibt sowohl veränderliche (*mutable*) als auch unveränderliche (*immutable*) Datenstrukturen, die einfachen Datentypen sind alle unveränderbar (*immutable*)

Beispiele von Datenstruktur-Operatoren anhand von Strings:

In [7]:
```
# Beispiel
wort = "Koeffizenten"
print(wort[0])              # Zugriff auf Elemente über []-Klammer
print("ente" in wort)       # Anwendung des Enthalten-Operators
print("Ambi" + wort[7:11])  # Konkatenation nur bei gleichartigen Datenstrukture
print(3*"bla")              # Vervielfältigung
print(len(wort))            # Jede Datenstruktur hat eine Länge, die ermittelt w

# Slicing "Syntax": s[start:ende:schrittweite]
print(wort[-1]) # letztes Element
print(wort[-5:]) # letzten 5 Elemente
print(wort[::2]) # jedes zweite Element
print(wort[1::2]) # Jedes zweite Element aber erst ab dem zweiten Element
print(wort[::-1]) # Umdrehen der Datenstruktur
```

```
K
True
Ambiente
blablabla
12
n
enten
Kefzne
ofietn
netneziffeoK
```

Anmerkung zur String-Formatierung

Der neuste/aktuellste Weg, einen String zu formatieren, ist via `f-String`. `f` steht für *formatted*. Vorteile gegenüber alten Formatierungsmethoden (z.B. `format()` oder `%`-Operator):

- prägnanter und lesbarer
- schneller

In [5]:
```
# Beispiele zum f-String
goldenerSchnitt = 1597/987      # Verhältnis einer Fibonaccizahl zu ihrem Vorgän
print(f"Mit dem Verhältnis zweier Fibonaccizahlen kann man den goldenen Schnitt
```

```
Dieser beträgt {goldenerSchnitt}")
print(f"Man kann aber auch schreiben, dass {goldenerSchnitt=}")
print(f"Schöner wäre: {goldenerSchnitt = }")
print(f"Wenn man nur die ersten vier Nachkommastellen braucht: {goldenerSchnitt
print(f"In Exponentialschreibweise wäre es: {goldenerSchnitt = :.4e}")
```

```
Mit dem Verhältnis zweier Fibonaccizahlen kann man den goldenen Schnitt annähern.
Dieser beträgt 1.618034447821682
Man kann aber auch schreiben, dass goldenerSchnitt=1.618034447821682
Schöner wäre: goldenerSchnitt = 1.618034447821682
Wenn man nur die ersten vier Nachkommastellen braucht: goldenerSchnitt = 1.6180
In Exponentialschreibweise wäre es: goldenerSchnitt = 1.6180e+00
```

Es gibt noch viele weitere Formatierungsmöglichkeiten mithilfe der Format Specification Mini-Language

Außerdem gibt es noch eigene String-Methoden, die für diese Vorlesung nicht weiter von Bedeutung sind.

4.1 Listen

Die Datenstruktur `list` ist eine geordnete Zusammenfassung **verschiedener** Objekte. Sie kann während der Laufzeit geändert werden und ist daher *mutable*. Der Python-Interpreter erkennt eine Listendefinition an den eckigen Klammern `[]`. Die Zählung beginnt immer beim Index 0.

In [2]:
```
# TODO Listendeklaration und Standardoperatoren
x = [3, 99, "Ein Text"]     # kann verschiedene Datentypen enthalten
y = [0b110, 8.5, x, "Wort"] # kann auch Listen in Listen geben
print(y)
print(y[2])
print(y[-1])
print(y[2][1])
print(y[0:2])
print(type(y))
print(id(y))
y[1] = 34
print(y)
print(id(y))
```

```
[6, 8.5, [3, 99, 'Ein Text'], 'Wort']
[3, 99, 'Ein Text']
Wort
99
[6, 8.5]
<class 'list'>
2487842805632
[6, 34, [3, 99, 'Ein Text'], 'Wort']
2487842805632
```

4.1.1 Operationen auf Listen

Methode	Beschreibung
`list.append(x)`	Die Liste list wird um das objekt x erweitert
`list.extend(L)`	Die Liste list wird um die Elemente der Datenstruktur L erweitert

Methode	Beschreibung
`list.insert(i, x)`	Objekt x wird and der Stelle i in die Liste list eingefügt
`list.remove(x)`	Löscht das *erste* Element mit dem Wert x aus der Liste
`list.pop()`	Das letzte Element wird aus der Liste gelöscht **und** zurückgegeben (Stapelverarbeitung)
`list.pop(i)`	Das i-te Element wird aus der Liste gelöscht **und** zurückgegeben
`list.index(x)`	Zurückgegeben wird der Index i des ersten Listenelements, bei dem `list[i] == x`
`list.count(x)`	Anzahl der Objekte mit dem Wert x
`list.sort()`	Die Elemente der Liste list werden aufsteigend sortiert
`list.reverse()`	Dreht die Reihenfolge der Objekte in der Liste um
`del list`	Löscht die **gesamte** Liste
`del liste[i]`	Löscht das Listenelement mit Index i, **ohne** es als Funktionswert zurückzugeben. Man kann auch Bereiche `[i:j]` löschen.

In [3]:
```
# TODO Beispiele zu Listenoperationen
liste = [] # leere Liste
print(liste)
liste.append(3)
liste.append(1)
liste.append(99)
print(liste)

andereListe = [5, 1, "?"]
liste.extend(andereListe)
#liste += andereListe
print(liste)

liste.insert(3, 22)
print(liste)

liste.remove("?")
print(liste)
liste.sort()
print(liste)

print(liste.index(5))

del liste[2:4]
print(liste)

del liste
#print(liste) #Fehler, da liste nicht mehr existiert

c = ["z", "f", "a", "b"]
c.sort()
c
```

```
[]
[3, 1, 99]
[3, 1, 99, 5, 1, '?']
[3, 1, 99, 22, 5, 1, '?']
[3, 1, 99, 22, 5, 1]
[1, 1, 3, 5, 22, 99]
3
[1, 1, 22, 99]
```

Out[3]: `['a', 'b', 'f', 'z']`

In [33]:
```python
# TODO Beispiel Liste als Stapel
l = []
l.append(3)
l.append(10)
l.append(2)

print(l)

e = l.pop()
print(e)
print(l)
```

```
[3, 10, 2]
2
[3, 10]
```

4.1.2 Listenabstraktion (*list comprehension*)

Wenn es nach Erfinder Guido van Rossum ginge, gäbe es statt `lambda` und Co. nur noch die *List Comprehension* in Python 3. Diese ist nämlich genauso eine elegante Methode, um Kollektionen in Python zu definieren und zu erzeugen. Demnach ist es auch eine einfache Methode, um neue Listen oder auch Unterlisten zu erzeugen, oder um innerhalb einer Liste, Anweisungen auszuführen.

List Comprehensions erzeugen aus einer Eingabeliste eine Ausgabeliste. Die einfachste und allgemeine Form folgt dem Syntax:

```
ausgabeliste = [ausdruck for element in eingabeliste]
```

In [34]:
```python
# TODO Erzeugen der Quadratzahlen von 1 bis 9 und speichern in einer Liste
# "Klassisch"
quadrate = []
for i in range(1, 10):
    quadrate.append(i**2)
quadrate
```

Out[34]: `[1, 4, 9, 16, 25, 36, 49, 64, 81]`

In [36]:
```python
# TODO Erzeugen der Quadratzahlen von 1 bis 9 und speichern in einer Liste
# Via list comprehension
quadrate_comp = [i**2 for i in range(1, 10)]
quadrate_comp
```

Out[36]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Der Syntax kann ebenso um eine `if`-Anweisung ergänzt werden:

```
ausgabeliste = [ausdruck for element in eingabeliste if bedingung]
```

In [37]:
```
# TODO alle Zahlen von 10 bis 100, die durch 4 teilbar sind
teilbarDurchVier = [x for x in range(10, 101) if x % 4 == 0]
print(teilbarDurchVier)
```

```
[12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72, 76, 80, 84, 88,
92, 96, 100]
```

4.1.3 `lambda`, `map`, `filter` und `reduce` als Alternative zur Listenabstraktion

Anwenden mit `map`

Mithilfe der *list comprehension* kann man also einen `ausdruck` (oder Funktion) auf eine Liste (*eingabeliste*) anwenden. Genau dasselbe ist mit der `map`-Funktion möglich:

```
ausgabe = map(func, seq)     # func = Funktion, seq = sequentieller
Datentyp
```

In [40]:
```
# zähle zu allen Zahlen einer Liste 2 dazu
neueListe = list(map(lambda x: x +2, range(10)))
print(neueListe)
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [4]:
```
# TODO Umrechnung von Grad Celsius in Fahrenheit: °F = (9/5)*°C +32
messungenCelsius = [35.4, 36.3, 37.8, 38.0, 39.6, 42.1 ]
messungenFahrenheit = list(map(lambda x: (9.0/5)*x + 32, messungenCelsius))
print(messungenFahrenheit)
print([round(x, 2) for x in messungenFahrenheit])
```

```
[95.72, 97.34, 100.03999999999999, 100.4, 103.28, 107.78]
[95.72, 97.34, 100.04, 100.4, 103.28, 107.78]
```

Filtern mit `filter`

`map` nimmt alle Ergebnisse aus der Funktion in die neue Liste auf. `filter` hingegen nur diejenigen Funktionsergebnisse, die `True` liefern. Sie liefert ein gefiltertes *Iterable* zurück.

Syntax:

```
ausgabe = filter(func, iter)     # func = Funktion, iter = seq. Datentyp
oder iterierbares Objekt
```

Einschub: Was ist ein *iterierbares Objekt (itarable)*?

> Bei einem iterierbaren Objekt (englisch: iterable) handelt es sich um ein Objekt, das seine

Elemente einzeln zurückgeben kann, d.h. eins nach dem anderen. Beispiele für iterierbare Objekte sind alle sequentiellen Datentypen wie list, str und tuple), aber auch die

Dictionary-Klasse dict.

→ Jeder sequentielle Datentyp ist iterierbar, aber nicht jedes iterierbare Objekt hat einen sequentiellen Datentyp. Bei sequentiellen Datentypen gibt es einen **Index**, um auf einzelne Elemente zuzugreifen

In [5]:
```
# TODO alle Zahlen von 10 bis 100, die durch 4 teilbar sind
teilbarDurchVierFilter = list(filter(lambda x:  x % 4 == 0, range(10,101))) # la
print(teilbarDurchVierFilter)
```

```
[12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 68, 72, 76, 80, 84, 88,
92, 96, 100]
```

Reduzieren mit `reduce`

"aka *Guido ärgern*", denn diese Funktion wurde aus den Standardmodulen von Python 3 verbannt. Genutzt werden kann es daher nur, wenn sie vorher aus dem Modul `functools` importiert wird. Syntax:

```
from functools import reduce
ausgabewert = reduce(func, seq)     # func = Funktion, seq =
sequentieller Datentyp
```

Dennoch kann die Funktion nützlich sein. Sie wird verwendet, um eine Funktion auf die Elemente einer Sequenz *kumulativ* anzuwenden und auf einen einzigen Wert zu reduzieren.

In [60]:
```
# TODO Summe einer Liste
from functools import reduce
liste = [3, 5, 1, 7, 6, 4]
summe = reduce(lambda x,y : x * y, liste)
summe
```

Out[60]: 2520

In [59]:
```
# TODO Maximum einer Liste
from functools import reduce
liste = [3, 5, 1, 7, 6, 4]
maximum = reduce(lambda x,y: x if x < y else y, liste)
maximum
```

Out[59]: 1

4.1.4 Kopieren von Listen

In [61]:
```
# Was nichts mit Kopieren zu tun hat
x = [ 1, 2, 3, 4, 5]
y = x
print("x: ", x)
print("y: ", y)
print(id(x), id(y), "\n")
# Veränderung der Liste
print("Veränderung")
x[1] = 33
print("x: ", x)
```

```
print("y: ", y)
print(id(x), id(y), "\n")
```

```
x:  [1, 2, 3, 4, 5]
y:  [1, 2, 3, 4, 5]
1896187632448 1896187632448

Veränderung
x:  [1, 33, 3, 4, 5]
y:  [1, 33, 3, 4, 5]
1896187632448 1896187632448
```

Was in obiger Zelle geschehen ist, nennt man *Aliasing*, weil ein Alias (anderer Name) für einen bereits existierenden Namen gebildet wird (`y = x`). Beide Namen zeigen intern auf das gleiche Objekt, daher kann nicht von einer Kopie gesprochen werden.

In [6]:
```
# TODO Flache Kopie
x = [ 1, 2, 3, 4, 5]
y = x[:] # Erzeugen einer flachen Kopie via Slicing
x[1] = 33
print("x: ", x)
print("y: ", y)
print(id(x), id(y), )
```

```
x:  [1, 33, 3, 4, 5]
y:  [1, 2, 3, 4, 5]
2487840883328 2487842715776
```

Das obige Beispiel scheint zu funktionieren. Statt des Slicing-Operators `[:]` kann man auch schreiben `y = x.copy()` .

Funktioniert das auch für verschachtelte Listen?

In [65]:
```
# TODO Flache Kopie einer verschachtelten Liste
x = [ 1, [2, 3], [4, 5]]
y = x[:] # Erzeugen einer flachen Kopie via Slicing
x[0] = 99
x[1][0] = 33
print("x: ", x)
print("y: ", y)
print(id(x), id(y) )
```

```
x:  [99, [33, 3], [4, 5]]
y:  [1, [33, 3], [4, 5]]
1896186534272 1896191103744
```

Die Antwort ist **nein**. Flache Kopien (*shallow copy*) funktionieren auch nur mit flachen Listen, d.h. Listen, die keine Unterlisten enthalten. Beim flachen Kopieren werden werden auch die Zeiger auf die Unterlisten mitkopiert, wodurch die Kopie auf dieselbe Unterliste zeigt, wie das Original.

Um auch die Unterobjekte mitzukopieren und sog. *tiefe Kopien* anzufertigen, benötigt man die Funktion `deepcopy()` . Eine tiefe Kopie ist ein völlig selbstständiges System von Objekten und hat keinerlei Zusammenhang mehr zum Original.

In [66]:
```
# TODO tiefe Kopie
import copy
x = [ 1, [2, 3], [4, 5]]
y = copy.deepcopy(x) # Erzeugen einer tiefen Kopie
x[0] = 99
x[1][0] = 33
print("x: ", x)
print("y: ", y)
print(id(x), id(y) )
```

```
x:  [99, [33, 3], [4, 5]]
y:  [1, [2, 3], [4, 5]]
1896189369024 1896189730944
```

4.2 Tupel

Ein Tupel ist eine Sequenz von Elementen, die iterierbar sind, aber nicht verändert werden können. Tupel sind *immutable*. Von der Idee her sind Listen eine Aufzählung vieler Einzelobjekte, wohingegen Tupel die Modellierung der Struktur *eines* komplexen Einzelobjektes betonen.

Faustregel: Tupel machen alles, was Listen auch machen. ABER sie sind unveränderlich.

In [7]:
```
# TODO Beispiele
t = ()
t = 1, 2, 3
print(t)
t = 1, 2, "wort", 8.5
print(t)
eins = (1,)
print(eins)
liste = list(t)
liste[1] = 4
print(liste)
t = tuple(liste)
print(t)

wortTupel = tuple("hallo")
print(wortTupel)
```

```
(1, 2, 3)
(1, 2, 'wort', 8.5)
(1,)
[1, 4, 'wort', 8.5]
(1, 4, 'wort', 8.5)
('h', 'a', 'l', 'l', 'o')
```

4.2.1 Anwendung von Tupel

In Python sind auch Mehrfachzuweisungen (mehrere Zuweisungen in einer Zeile) möglich, durch die man beispielsweise ideal Variablenwerte vertauschen kann:

In [76]:
```
# reine Mehrfachzuweisung
minimum, maximum, text = 3, 99, "Ein Text"
print(minimum, maximum, text)
```

```
maximum, minimum = minimum, maximum      # Tauschen zweier Variablenwerte
print(minimum, maximum, text)
```

```
3 99 Ein Text
99 3 Ein Text
```

Folgendes bezeichnet man als Tupel-Packing:

In [78]:
```
# TODO Tupel-Packing
t = "Peter", "Bernhard"
print(t)
print(type(t))
```

```
('Peter', 'Bernhard')
<class 'tuple'>
```

Von Tupel-Unpacking ("Tupel-Entpacken") spricht man, wenn man die einzelnen Werte eines Tupels Variablen zuordnet:

In [79]:
```
# TODO Tupel-Unpacking
vorname, nachname = t
print(vorname, nachname)
```

```
Peter Bernhard
```

In [81]:
```
# TODO Tupel-Unpacking im Funktionsaufruf
def f(x,y,z):
    print(x,y,z)

t = 34, 25, 59
f(*t)
```

```
34 25 59
```

4.2.2 Nutzung der `enumerate()` -Funktion

`enumerate()` ist eine *built-in function*, die zu jedem iterierbaren Objekt (*iterable*) einen Index/Zähler hinzufügt und das Ergebnis in Form eines Tupel zurückgibt. Der Syntax lautet `enumerate(iterable, start = 0)`, wobei:

- **iterable**: irgendein iterierbares Objekt
- **start**: Optional. Gibt den Startindex an. Default = 0.

In [85]:
```
# TODO Beispiel
liste = ["eat", "sleep", "repeat"]
list(enumerate(liste, 100))

for index, wort in enumerate(liste):
    print(index, wort)
```

```
0 eat
1 sleep
2 repeat
```

Die `next()` -Funktion:

In [88]:
```
fruits = ['apple', 'banana', 'cherry']
enum_fruits = enumerate(fruits)
```

```
next_element = next(enum_fruits)
print(f"Next Element: {next_element}")
next_element = next(enum_fruits)
print(f"Next Element: {next_element}")
next_element = next(enum_fruits)
print(f"Next Element: {next_element}")
#next_element = next(enum_fruits)      # geht nicht
#print(f"Next Element: {next_element}")
```

```
Next Element: (0, 'apple')
Next Element: (1, 'banana')
Next Element: (2, 'cherry')
```

"Verändern" eines Tupel:

In [95]:
```
# Erweiterung eines Tupels
t = ( 1, 2, 3 )
print("Original", t)
# Versuch1, Ziel: ( 1, 2, 3, 4 )
t1 =  (t, 4)
print("Versuch 1:", t1)
# Versuch 2, Ziel: ( 1, 2, 3, 4 )
t2 = (*t, 4)
print("Versuch 2:", t2)
# Versuch 3, Ziel: ( 1, 2, 3, 4 )
t3 = t + (4,)
print("Versuch 3:", t3)
# Versuch 4, Ziel: ( 1, 2, 3, 4 )
tmp = list(t)
tmp.append(4)
t4 = tuple(tmp)
print("Versuch 4:", t4)
```

```
Original (1, 2, 3)
Versuch 1: ((1, 2, 3), 4)
Versuch 2: (1, 2, 3, 4)
Versuch 3: (1, 2, 3, 4)
Versuch 4: (1, 2, 3, 4)
```

Achtung: Denkweise anpassen!

In [96]:
```
#  Iterationen mit der for-Schleife

t1 = ( (1,2,3), (4,5,6) ) # 2x3-Tuple
# Array-Denkweise mit Indizes (schlecht!)
for i in range(2):
    for j in range(3) :
        print(t1[i][j])
```

```
1
2
3
4
5
6
```

In [97]:
```
# Tupel-Denkweise mit Elementen (gut!)
for i in t1:
```

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

```
1
2
3
4
5
6
```

In [98]:
```
t2 = ( (1,), (2,3), (4,5,6)) # Dreiecks-Tupel
# TODO Ausgabe der Tupel-Elemente des Dreiecks-Tupels
for i in t2:
    for j in i :
        print(j)
```

```
1
2
3
4
5
6
```