11. Sequenzen

Sequenztypen:

bekannt:
-Strings: `str` Aneinanderreihung von Character `char`

neu:

- Tupel: `tuple`
- Listen: `list`

11.1 Tupel

- Sequenz von Objekten
- es können beliegibe Objekte sein, auch andere Tupel und Listen
- unveränderlich: Ein Tupel ändert sich nie, es enthält immer dieselben Objekte in der selben Reihenfolge.
- aber: die Enthaltendne Objekte können sich ändern.

Syntax:

`Variablenname = (Obkjekt 0, Objekt 1, ... , Objekt n-1)`

oder wenn keine Mehrdutigkeiten entstehen auch ohne Klammern:

`Variablennema = Objekt 0, Objekt 1, ..., Objekt n-1)`

Beispiel 1:

In [47]:
```
myfirsttuple = (2,3,5,7,11)
print (myfirsttuple)
```

```
(2, 3, 5, 7, 11)
```

Beispiel 2:

In [48]:
```
mysecondtuple = 2,4,6,8,10
print(mysecondtuple)
```

```
(2, 4, 6, 8, 10)
```

Beispiel 3:

In [49]:
```
thirdtuple = ((1,6),(2,4),(3,4))
print(thirdtuple)
```

```
((1, 6), (2, 4), (3, 4))
```

Beipsiel 4: Einelementiges Tupel

In [50]:
```
onetuple = (14,)
print(onetuple)
```

```
(14,)
```

11.2 Listen

- eine Sequenz von Objekten
- es können beliegibe Objekte sein, auch andere Tupel und Listen
- Elemente können angehängt, eingefügt oder entfernt werden.

Syntax:

```
Variablenname = [Objekt 0, Objekt 1, ... , Objekt n-1]
```

Beispiel 5:

In [51]:
```
myfirstlist = [3, 5, 7, 9]
print
```

Out[51]:
```
<function print(*args, sep=' ', end='\n', file=None, flush=False)>
```

11.3 Tuple Unpacking

- entspricht komponentenweise Zuweisung von Tupeln
- kleine Form von Pattern Matching
- funktioniert mit Listen und Strings etc.

Beispiel 6:

In [52]:
```
a, b = 2, 3
print(a)
print(b)
```

```
2
3
```

Gleichwertig zu

In [53]:
```
a=2
b = 3
```

Aber Vorteil bei der Mehrfachzuweisung ist: Die Zuweisung erfolgt gleichzeitig. Damit kann man vertauschen:

In [54]:
```
a,b=2,3
a, b = b,a
```

```
print ("a ist ", a, "b ist ", b)
```

```
a ist  3 b ist  2
```

Beispiel 7:

In [55]:
```
[a, (b,c), (d,e),f]= (42, (6,9), "do", [1,2,3])

print(a , " | ", b, " | ", c ," | ", d, " | ", e, " | ",f )
```

```
42  |  6  |  9  |  d  |  o  |  [1, 2, 3]
```

11.4 Gemeinsamkeiten von String, Listen und Tupel

- enthalten untergeordnente Objekte
- bestimmte Reihenfolge der Objekte
- Typen mit dieser Eigenschaft heißen Sequenztypen
- allgmeiner Containertypen.

11.5 Operationen auf Sequenzen

Verkettung: `"Hallo " + "Kurs" == ""Hallo Kurs"`

Wiederholung: `2*"Hallo " == "Hallo Hallo "`

Indizierung: `"Python"[1] == "y"`

Mitgliedschaftstest: `11 in [2,3,5,7,11,13,17]`

Slicing: `"Friedrich Gymnasium"[10:18] == "Gymnasium"`

Typkonvertierung `list("Hallo")=["H", "a", "l", "l", "o"]`

Iteration: `for x in "python"`

11.5.1 Verkettung

- "+"-Operator

Beispiel 8:

In [56]:
```
mylist = ["hallo", "welt"]
yourtuple = (2,5,7,9) #Tupel
print(["hallo"]+mylist)

print(yourtuple + yourtuple)#Tupel verkettet

#print(yourlist + mylist) #Tupel mit Liste verketten geht nicht

print(list(yourtuple)+ mylist)
```

```
['hallo', 'hallo', 'welt']
(2, 5, 7, 9, 2, 5, 7, 9)
[2, 5, 7, 9, 'hallo', 'welt']
```

11.5.2 Wiederholung

- "*"-Operator

Beispiel 9:

In [57]:
```
print("*" * 20) # * wird 20 mal wiederholt

print([None, 2,3]*3)# Die Liste wird 2 mal wiederholt

print(2*("Marcus", ["in" , " colloseum", "est"]))

print("!" * (-2))
```

```
********************
[None, 2, 3, None, 2, 3, None, 2, 3]
('Marcus', ['in', ' colloseum', 'est'], 'Marcus', ['in', ' colloseum', 'es
t'])
```

11.5.3 Sequenzierung

- "[]"-Schreibweise
- Zugriff auf die Position der Elemente einer Sequenz möglich über einen Index
- Sequenzen können von vorne oder von hinten indiziert werden
- die Indizierung von vorne beginnt mit dem Index 0 !
- zur Indizierung von hinten dienen negative Indizes. Das hinterste Element hat den Index -1

Beispiel 10:

In [58]:
```
primes = (2,3,5,7,11,13)
print (primes [1]) #das zweite Element

print (primes[-1])#das letzte Element, die 13

text="Es tut mir in der Seele weh"
text[-2] # das e
text[30] #String nicht so lang, folgich Fehlermeldung
```

```
3
13
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[58], line 8
      6 text="Es tut mir in der Seele weh"
      7 text[-2] # das e
----> 8 text[30] #String nicht so lang, folgich Fehlermeldung

IndexError: string index out of range
```

11.5.4 Test auf Mitgliedschaft

- `in`-Operator
- Syntax: `item in seq`
- True, wenn `seq` das Element `item` enthält
- `seq` muss Tupel oder Liste sien, sonst such man einen substring
- Syntax: `substring in string`

Beispiel 11:

In [ ]:
```
print(2 in [1,2,4])

if "pizza" in ("Ei", "Omlette", "Croissant"):
    print("lecker")
```

```
True
```

In [ ]:
```
print("i" in "pizza", "pozz" in "pizza", "izza" in "pizza")
```

```
True False True
```

11.5.5 Slicing

- Teile ausschneiden
- Syntax: `seq[start:end]`

Beispiel 12:

In [ ]:
```
primes = (2,3,5,7,11,13, 17, 19)

print(primes[1:4])#beginnt an der 2 Position (Position 1) und hört vor Posit

print(primes[:2])#beginnt am Anfang und geht bist vor die Position 2

len(primes) #gibt die Länge des Tupels primes an.

print(primes[-3:])# drittletzte Position bis Ende

print(primes[:])#von Anfang bis Ende

#print("Schnitzel, Pommes und Salat")
```

```
(3, 5, 7)
(2, 3)
(13, 17, 19)
(2, 3, 5, 7, 11, 13, 17, 19)
```

Bemerkung:

- Beim Slicing gibt es keine Indexfehler
- Bereiche außerhalb sind einfach leer.

Typkonvertierung:

- `tuple` -Operator wandelt eine Liste in ein Tupel um
- `list` -Operator wandelt ein Tupel in eine Liste um
- `str` -Operator wandelt in eine "Druckversion"

In [ ]:
```
# Liste in Tupel umwandeln
tuple([2,4,6])
```

Out[ ]:
```
(2, 4, 6)
```

In [ ]:
```
#Tupel in eine Liste umwandeln
list((3,5,7))
```

Out[ ]:
```
[3, 5, 7]
```

In [ ]:
```
#gibt einen ausgedrucket Repräsenation zurück
str(["a", "b", "c"])
```

11.5.7 Liste zu einem String zusammenfügen

- `join` -Operator
- Anwendung auf einen leeren String
- einzelne Elemente einer Sequenz werden zu einem String zusammenfeführt

Beispiel 13:

In [ ]:
```
#Liste von Strings wieder in einen String überführen
"".join(["h", "e", "l","l", "o"])
```

Out[ ]:
```
'hello'
```

In [ ]:
```
#Tupel von Strings in einem String zusammenführen, angewandt auf ein nicht l
"*".join(("t","o","p"))
```

Out[ ]:
```
't*o*p'
```

11.5.8 Summe berechnen

- `sum` -Operator
- einzelne Elemente einer Sequenz werden zusammengerechnet

Beispiel 14:

```
In [ ]: sum((2,3,5,7,11))
```

```
Out[ ]: 28
```

11.5.9 Minimum bzw. Maximum einer Sequenz berechnen

- `min`-Operator, oder `max`-Operator
- gibt das Minimum oder Maximum einer Sequenz an.
- besteht die Liste selbst aus Sequenzen, dann wird die lexikographische Ordnung verwendet.
- nicht vergleichbare Typen führen zu einer Fehlermeldung

Beispiel:

```
In [ ]: min([6,13,65,2,34,-2])
```

```
Out[ ]: -2
```

```
In [ ]: max([6,13,65,2,34,-2])
```

```
Out[ ]: 65
```

```
In [ ]: min(("ah", "beh", "ce", "de"))
```

```
Out[ ]: 'ah'
```

11.5.10 Länge der Sequenz

- `len`-Operator

Beispiel 15:

```
In [ ]: len("Hallo")
```

```
Out[ ]: 5
```

```
In [ ]: len([1,1,2,3,5,8,13])
```

```
Out[ ]: 7
```

11.5.10 (stabil) sortierte Liste erzeugen

- `sorted`-Operator
- liefert eine Liste mit den selben Elementen, aber sortiert.

Beispiel 16:

In [ ]:
```
sorted([5, 5, 3, 4, 1])
```

Out[ ]:
```
[1, 3, 4, 5, 5]
```

11.5.11 iteritertes OR auf Listen

- `any` -Operator
- entspricht: `Objekt 0 OR Objekt 2 OR ...`

Beispiel 17:

In [ ]:
```
any((2,3,4,5))
```

Out[ ]:
```
True
```

11.5.11 iteritertes AND auf Listen

- `all` -Operator
- entspricht: `Objekt 0 AND Objekt 1 AND ...`

Beispiel 18:

In [ ]:
```
all([4,88,99,45,22,1])
```

Out[ ]:
```
True
```