Unveränderbare Listen: Tupel

- Moment mal: Alle Variablen in Python sind veränderbar!
- Richtig, Tupel sind veränderbar aber Elemente eines Tupels nicht
- Syntax: Wie Listen aber mit runden Klammern () statt eckigen []

```
tuple_1 = ()              # leeres Tupel
tuple_2 = (7, 8, 9)   # Tupel mit 3 Werten vom Typ Integer
tuple_3 = ("x", "y")# Tupel mit 2 Werten vom Typ String
tuple_4 = (10, 53.12, "abc", [8, 9], (1, 2, 3))
# Tupel mit verschiedenen Datentypen
```

Ausnahme: Tupel mit einem Element

- Bei Tupel mit einem Element, gibt es ein seltsames Verhalten:
- So nicht:

```
my_tuple = ("hello")
print(type(my_tuple))  # <class 'str’>
```

- Tupel mit einem Element erstellen:

```
my_tuple = ("hello",)
print(type(my_tuple))  # <class 'tuple'>
```

Tupelelemente addressieren & verändern

- Genau wie bei Listen: negativer/positiver Index, Slice
- Anmerkung: Auch eckige Klammern verwenden!
- Elemente verändern: Nicht ganz so leicht!
- Tupelelemente lassen sich nicht löschen
  - Ganze Tupel aber schon!
- Ansonsten gilt:
  - Jede Listenmethode, die keine Werte verändert, lässt sich auch für Tupel nutzen

Beispiel

```
tup = (4, 2, 3, [6, 5])


# Tupelelemente lassen sich nicht verändern:
tup[1] = 5 # TypeError


# Listen in Tupel lassen sich verändern:
tup[3][0] = 3
print(tup) # Output: (4, 2, 3, [3, 5])


# Tupel lassen sich überschreiben:
tup = (4, 5, 3, [6, 5])
print(tup)
```

Warum sollte ich Tupel statt Listen nutzen?

- Tupel sind kleiner als Listen -> Steigerung der Performance
- Wenn über eine Menge von Werten **nur** iteriert werden soll
- Ein gewisser Schreibschutz
- Tupel lassen sich leicht zu Dictionaries umwandeln
  - Dazu später mehr
- Anwendungsfälle von Tupel:
  - Daten, die zusammen gehören:
    z.B. Informationen einer Person:
    p = ("Fred", "Sterling", 14, 6, 1981, "Developer")
    first_name = p[0]
    last_name = p[1]
    birth_year = p[4]

Übung

- 11 – Übung.py