I Objektorientierte Programmierung

1. Programmierpradigmen

- bisher: Imperative Programmierung
  - Zustand wird durch Zusammensetzung der Variablen bestimmt.
  -
- neu: Objektorientierte PRogrammierung
  - alles Dinge sind Objekte
  - Objekte kommunizieren untereinander
  - Objekte bilden Teilzustände ab

2. Objekte und Attribute

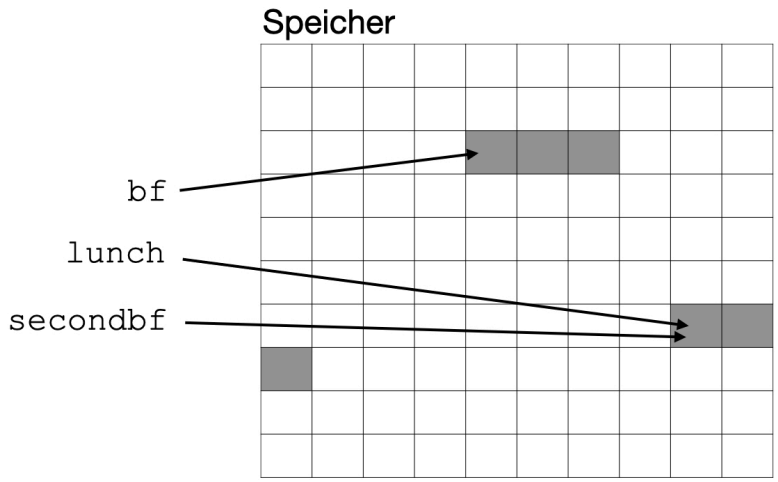
- Alle Werte in Python sind in Wirklichkeit Objekte
- Objekte bestehen aus Attributen (= Eigenschaften der Objekte) und Methoden(= Funktionen für die Objekte)
- Objekt = Instanz

1.1 Objekte anlegen

- Jedes Objekt besitzt eine eigene Identität.
- Beim Anlegen eines Objekts wird Speicherplatz im Speicher reseviert. Der Name des Objekts ist ein Zeiger auf diesen Speicherbereich.

```
bf, lunch = ["ei", "speck", "toast"],["Suppe", "Salat", "Bohnen"]
secondbf = lunch
print(bf)
print(lunch)
print(secondbf)
```

```
['ei', 'speck', 'toast']
['Suppe', 'Salat', 'Bohnen']
['Suppe', 'Salat', 'Bohnen']
```



1.2 Objekte auf Gleichheit überprüfen

- Operatoren `is` und `is not` überprüfen auf Identität.
- `lunch is bf` liefert `True`, wenn `lunch` und `bf`dasselbe (Anmerkung: nicht das Gleiche) Objekt sind, ansonsten `False`
- `is not` liefert umgekehrte Ergebnisse.

```
print(lunch is bf)
print(bf is secondbf)
print(lunch is secondbf)
```

```
False
False
True
```

Wichtig:
- Test auf Identität erfolgt mit `is` und `is not`. Getestet wird, ob es sich um dasselbe Objekt handelt. - Test auf Gleichheit erfolgt mit `==` Operator. Getestet wird, ob es sich um den gleichen Typ handelt, ob sie gleich lang sind und ob ise die gleichen Werte haben.

Faustregel:
Verwende den Gleichheitstest.

```
print(lunch == bf)
print(bf == secondbf)
print(lunch == secondbf)
```

```
True
True
True
```

`None-Type` - Der `NoneType` hat nur den einzigen Wert `None` - Vergleiche können daher mit Gleicheit oder Identität erfolgen. - Aber: Vergleiche sollten mit `is None` oder mit `is not None` erfolgen.

1.3 Vorsicht Nebeneffekte

- Veränderbare Objekten (insbesondere Listen)
  - Attribute können modifiziert werden können
  - Bei einer Zuweisung x = y beeinflussen Operationen auf x auch y und umgekehrt.
- Nichtveränderbare Objekte (Zahlen(int, float, complex), Strings und Tupel)
  - Objekt kann durch die Zuweisung x = y nicht verändert werden.

1.4 Attribute

- Eigenschaften von Objekte heißen Attribute.
- Auf die Attribute von Objekten kann man mit der Punktnotation zugreifen.
- Syntax:

  `expression.attribut`

- Beispiel:
  - Komplexe Zahlen haben einen Realteil und einen Imaginärteil.
  - $2 + 3i, \quad 5 + 0i, \quad 0 - 3i$
  - Komplexe Zahlen die nur einen Realteil besitzen, entsprechen den reellen Zahlen.
  - $5 + 0i = 5$ - Komplexe Zahlen gibt es als Datentyp auch in Python. Der Imaginärteil wird in Python mit j bezeichnet.
  - Attribute heißen `real` und `imag`
  - Zugriff auf die einzelne Teil einer komplexen Zahl, d.h. auf die Attribute der Komplexen Zahl wie folgt möglich

```python
c = 3+4j
print(c)
print("Der Realteit ist:", c.real)
print("Der Imaginärteil ist:", c.imag)
```

```
(3+4j)
Der Realteit ist: 3.0
Der Imaginärteil ist: 4.0
```

1.5 Records und Klassen

- bisher kamen nur vorgefertigte Objekte zum Einsatz
- eigene sollen entworfen werden.
- benötigt wird ein Bauplan, eine **Klasse**

1.5.1 Definieren von Klassen

- Syntax: `python class ClassName: attributes`
- die Klassendefintion muss mindestens einmal aufgerufen werden.

Definition: - Ein Record ist ein Objekt, das mehrer untergeordnete Objekte, die sogenannten Atttribute enthält. - Ein Klasse definiert (zunächst nur), welche Attribute vorhanden sein sollen. - Objekte heißen auch Instanzen.

Beispiel:

In einem Rollenspiel werden die Gamecharacter durch Attribute beschrieben.

[!]{images/dndchar.jpg}

Dafür gibt es character sheets, welche die Eigenschaften des Pbjekts beshreiben.

Vereinfachung:

`name : str` Name des Charakters
`intelligence : int` IQ des Charakters
`strength : int` Die Stärke des Charackters ≥ 0

Die Klasse GameCharacter soll erzeugt werden.

```python
class GameCharacter:
    pass
```

Konvention:

Neue Klassennamen werden mit Großbuchstaben begonnen und in CamelCase geschrieben. In Funktionennamen werden mit Kleinbuchstaben gegonnen und gegebenenfalls "_" verwendet

1.5.2 Erzeugen von Instanzen

- Der Klassenname ist gleichzeitig auch ein Funktion (Konstruktor) der Klasse
- mit Aufruf der Klasse als Funktion wird eine neue Instanz angelegt.
- Beispiel:

```
orgor = GameCharacter()
leelah = GameCharacter()
print(orgor)
print(leelah)
print(orgor is leelah)
print(orgor == leelah)
print(isinstance(orgor, GameCharacter))
print(isinstance(0, float))
```

```
<__main__.GameCharacter object at 0x105b8bef0>
<__main__.GameCharacter object at 0x105b89f10>
False
False
True
False
```

```
orgor.name = "Orgor van Hauten"
orgor.strength = 20

leelah.name = "Leehlah Butterblume"
leelah.strength = 14
```

Problem:
- Es können weitere Attribute zu jedem Objekt ergänzt werden. Damit kann es Objekte geben mit unterschiedlichen Atrributen. - Instanzen sind dynamische Objekte, die sich nicht nur in den Werten der Atrribute sondern in der Zusammensetzung der Attribute unterscheidet.

```
orgor.intelligence = 4
print(orgor.intelligence)
leelah.intelligence = 12
print(leelah.intelligence)
leelah.appereance = 20
print(leelah.appereance)
```

```
4
12
20
```

1.6 Dataclasses

- import der Bibliothek Dataclasses
- Syntax: “‘python from dataclasses import dataclass @dataclass class DnDCharacter: name: str intelligence: int strength : int
- Bei Aufruf der Klasse als Funktion (Konstruktor) erzeugt eine Instanz mit garantiert den drei Atrributen.
- Instanzen der Klasse sind gleich, wenn die Attributwerte übereinstimmen.

```
from dataclasses import dataclass
@dataclass

class DnDCharacter:
    name : str
    intelligence : int
    strength : int

sigmund = DnDCharacter("Sigmund", 15, 21)
sigmund
```

```
DnDCharacter(name='Sigmund', intelligence=15, strength=21)
```

```
sigmund.ttt = 2
print(sigmund)
```

```
DnDCharacter(name='Sigmund', intelligence=15, strength=21)
```

1.6 Funktionen auf Records anwenden

Die Charaktäre der Klasse DnDCharacter sollen hochgestuft werden. Sie erhalten zhen Prozent mehr auf Ihre Attributwerte. Dazu wird eine Funktion benötigt.

1. Bezeichner und Datentypen

- level_up ist der Funktionsname
- Paramter sind
  `dndcharacter : DnDCharacter` der D&D-Charakter
  `percent : int`Prozentsatz, der den Atrributen zugerechnet werden soll.

2. Funktionsgerüst

```
    def levelup(
        dndcharacter : DnDCharacter,
        percent : int
    ):
    # to fill with code
```

3. Funktionsdefinition

```
    def levelup(
            dndcharacter : DnDCharacter,
            percent : int
         ):
        dndcharacter.strength*(1+percent/100),
        dndcharacter.intelligence*(1+percent/100)]
```

```
def levelup(
            dndcharacter : DnDCharacter,
            percent : int
         ):
        dndcharacter.strength= int(dndcharacter.strength*(1+(percent/100)))
        dndcharacter.intelligence= int(dndcharacter.intelligence*(1+(percent/100)))

Terha = DnDCharacter("Thera", 10, 22)
Mbala = DnDCharacter("M'Bala", 20,20)
Gerosh = DnDCharacter("Gerosh d'Oorc", 17,18)

print(Terha)
levelup(Terha, 10)
print(Terha)
```

```
DnDCharacter(name='Thera', intelligence=10, strength=22)
DnDCharacter(name='Thera', intelligence=11, strength=24)
```