





# OOP

- Statische Elemente in Python
- Operatoren in Python
- Code Organisation / Design Prinzipien

# SEMINAR: $2+2=4$

KLAUSUR: DANIEL HAT EINEN APFEL.  
BERECHNE DIE MASSE DER SONNE

# Beispiel

```
class Konto(object):  
    def __init__(self, inhaber, kontonummer,  
                  kontostand,  
                  kontokorrent=0):  
        self.Inhaber = inhaber  
        self.Kontonummer = kontonummer  
        self.Kontostand = kontostand  
        self.Kontokorrent = kontokorrent  
  
    def ueberweisen(self, ziel, betrag):  
        if(self.Kontostand - betrag < -self.Kontokorrent):  
            # Deckung nicht genugend  
            return False  
        else:  
            self.Kontostand -= betrag  
            ziel.Kontostand += betrag  
            return True  
  
    def einzahlen(self, betrag):  
        self.Kontostand += betrag  
  
    def auszahlen(self, betrag):  
        self.Kontostand -= betrag  
  
    def kontostand(self):  
        return self.Kontostand
```

```
>>> from konto import Konto  
>>> K1 = Konto("Jens", 70711, 2022.17)  
>>> K2 = Konto("Uta", 70813, 879.09)  
>>> K1.kontostand()  
2022.17  
>>> K1.ueberweisen(K2, 998.32)  
True  
>>> K1.kontostand()  
1023.85  
>>> K2.kontostand()  
1877.41
```



# Klassenvariablen

Eine Klassenvariable kann nur mit Hilfe des Klassennamens verändert werden (theoretisch)

Der Klassenname und das Attribut werden durch einen Punkt miteinander verbunden

```
Modul.Klasse.Attribut = Wert
```



# Klassenvariablen

- werden innerhalb der Klasse, aber außerhalb einer Methode definiert
- werden häufig zu Beginn des Klassenrumpfes aufgelistet
- sind Attribute, die alle Objekte teilen
- **können von jedem Objekt der Klasse verändert werden**
- sind globale Attribute eines Objekts



# Klassenvariablen

```
class Rechteck:
    ANZAHL = 0
    FARBE_KANTE = "Black"
    FARBE_FÜLLUNG = "White"

    def __init__(self, b = 10, h = 10):
        self.__xPos = 0
        self.__yPos = 0
        self.hoehe = h
        self.breite = b
        Rechteck.ANZAHL = Rechteck.ANZAHL + 1
```



## weiteres Beispiel

```
class RationalNumber:
    numberOfInstances = 0
    def __init__(self,a,b):
        self.n = a
        self.m = b
        RationalNumber.numberOfInstances += 1

def testNumberInstances():
    assert RationalNumber.numberOfInstances == 0
    r1 = RationalNumber(1,3)
    assert r1.numberOfInstances == 1
    r1.numberOfInstances = 8
    assert r1.numberOfInstances == 1
testNumberInstances()
```





## weiteres Beispiel

```
1 class T:
2     t = 0
3     def __init__(self, x):
4         self.x = x
5         T.t += 1
6         self.x += 1
7
8
9     T.t = 1000
10
11     print('here')
12     assert 1 == 1
13
14     t = T(10)
15     print('t.x=', t.x)
16
17
18
19     t1 = T(10)
20     print(T.t)
21
22     t.t = 10
23     #T.t = 101
24
25     print(T.t, t.t, t1.t)
```



# Statische Methoden

```
class T:
    __counter = 0

    def __init__(self):
        type(self).__counter += 1

    @staticmethod
    def TotalInstances():
        return T.__counter

>>> T.TotalInstances()
>>> x = T()
>>> x.TotalInstances()
```

# Statische Methoden

```
class RationalNumber:
    #class field, will be shared by all the instances
    numberOfInstances = 0

    def __init__(self, n, m):
        """
        Initialize the rational number
        n, m - integer numbers
        """
        self.n = n
        self.m = m
        RationalNumber.numberOfInstances += 1

    @staticmethod
    def getTotalNumberOfInstances():
        """
        Get the number of instances created in the app
        """
        return RationalNumber.numberOfInstances

def testNumberOfInstances():
    """
    test function for getTotalNumberOfInstances
    """
    assert RationalNumber.getTotalNumberOfInstances() == 0
    r1 = RationalNumber(2, 3)
    assert RationalNumber.getTotalNumberOfInstances() == 1

testNumberOfInstances()
```



# Hooks

- bisher wurden einige Operatoren vorgestellt: +, -, ...
- in Python gibt es aber gar keine Operatoren, sondern nur Operationen:
  - der „\*“-Operator ruft beispielsweise intern die `__mul__` Methode des ersten Operanden auf
  - diese speziellen Methoden kann man selbst definieren, um damit die Funktionalität zu ändern oder zu erweitern



# Hooks

```
class Rational:
    def __init__(self, num, den):
        self.num = num
        self.den = den
```

```
    def __mul__(self, other):
        num = self.num * other.num
        den = self.den * other.den
        return Rational(num, den)
```

```
    def __repr__(self):
        return "R("+ str(self.num)+", "+ str(self.den) + ")"
```

```
    def __str__(self):
        return str(self.num) + "/" + str(self.den)
```

```
>>> r1 = Rational(1,2)
>>> r2 = Rational(3,4)
>>> r1 * r2
R(3, 8)
>>> print(r1 * r2)
3/8
```

# Hooks

- Vergleichsoperatoren (Rückgabe: True / False):
  - `__eq__` → `==`
  - `__ge__` → `>=`
  - `__gt__` → `>`
  - `__le__` → `<=`
  - `__lt__` → `<`
  - `__ne__` → `!=`
- `__bool__` : gilt das Objekt als Wahr oder Falsch? (Gibt True oder False zurück)



# Hooks

- Numerische Operationen:

- `__add__` → +
- `__div__` → /
- `__mul__` → \*
- `__sub__` → -
- `__mod__` → %

- Element-Zugriff für Sammeltypen:

- `__getitem__(self, index)` → `x[i]`
- `__setItem__(self, index, value)` → `x[i] = 10`



# Hooks

```
class Data:
    def __init__(self):
        self.data = []

    def add_element(self, elem):
        self.data.append(elem)

    def print_elements(self):
        for el in self.data:
            print(el)

    def __getitem__(self, index):
        return self.data[index]

    def __setitem__(self, index, value):
        self.data[index] = value

data = Data()

for i in range(10):
    data.add_element(i)

print ('second element', data[1])
data[1] = 101
data.print_elements()
```

```
second element: 1
0
101
2
3
4
5
6
7
8
9
```





# Beispiel

```
class R:
    def __init__(self, a, b):
        if b == 0:
            raise ZeroDivisionError("b cannot be 0")
        if isinstance(a, int) and isinstance(b, int):
            self._a = a
            self._b = b
        else:
            raise ValueError("args should be int")

    def __add__(self, o):
        return T (self.a*o.b+self.b*o.a, self.b*o.b)

    def __lt__(self, o):
        return self.a/self.b < o.a/o.b

    def __eq__(self, o):
        return self.a == o.a and self.b == o.b
```

```
def __str__(self):
    if self.b == 1:
        return str(self._numerator)
    return "%i/%i" % (self.a, self.b)

@property
def a(self):
    return self._a

@property
def b(self):
    return self._b
```



# Beispiel

```
def main():
    r = R(1,2)

    print(R(1,2) == R(1,2))

    print (r + R(1,2))

    print (r < R(2,3))

    try:
        p = R(1,0)
    except ZeroDivisionError:
        print(Fraction not Valid')
    except ValueError:
        print(Types not Valid')

    print('here')
main()
```

```
10 import * as ProtectedRoute from './ProtectedRoute'
11 import * as serviceWorker from './serviceWorker'
12
13 ReactDOM.render((
14   <BrowserRouter>
15     <Switch>
16       <Route path="/login" component={Login} />
17       <ProtectedRoute exact={true} path="/" component={Dashboard} />
18       <ProtectedRoute path="/settings" component={Settings} />
19     </Switch>
20   </BrowserRouter>
21 ), document.getElementById('root'));
```

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL



# Codestruktur

- was ist das genau?
- man benutzt einige Design Prinzipien, um Code besser strukturieren zu können
- Single Responsibility Prinzip
- Separation of Concerns
- Dependencies
- Coupling and Cohesion

# Single-Responsibility-Prinzip

- Jede Funktion sollte für eine Sache verantwortlich sein
- Jede Klasse sollte eine Entität darstellen
- Jedes Modul sollte einem Aspekt der Anwendung entsprechen





# Single-Responsibility-Prinzip

- lass uns das folgende Beispiel nehmen

```
def filterScore(scoreList):  
    st = input("Start score :")  
    end = input("End score:")  
    for score in scoreList :  
        if score [1] > st and score [1] < end:  
            print(score)
```



# Single-Responsibility-Prinzip

- lass uns das folgende Beispiel nehmen

```
def filterScore(scoreList):  
    st = input("Start score :")  
    end = input("End score:")  
  
    for score in scoreList :  
        if score [1] > st and score [1] < end:  
            print(score)
```

- liest etwas von der Tastatur ein
- berechnet was
- gibt das Ergebnis aus



# Single-Responsibility-Prinzip

- kann diese `filterScore` Funktion sich verändern?
- ein komplett anderes Format für Input
  - Konsoleanwendung (Menu)
  - GUI
  - Webseite
- ein neuer Filter
- ein anderes Format für Output
- → die Methode hat 3 Verantwortungen





# Single-Responsibility-Prinzip

- der gleiche gilt für Module
- Module stellen Methoden zusammen, die thematisch miteinander passen
- mehrere Verantwortungen sind schwer zu
  - verstehen
  - verwenden
  - testen
  - warten
  - weiterentwickeln



# Separation of Concerns

- man soll das Programm in verschiedenen Abschnitte aufteilen
- jeder Abschnitt adressiert ein bestimmtes Problem
- Concerns - Informationen, die auf Code auswirken
  - z. B. Computerhardware, auf der das Programm ausgeführt wird, Anforderungen, Funktionen und Modulnamen
- richtig implementiert führt zu einem Programm, das einfach zu testen ist und einfach wiederverwendet werden können



# Separation of Concerns

- die gleiche Methode

```
def filterScore(scoreList):  
    st = input("Start score :")  
    end = input("End score:")  
    for score in scoreList :  
        if score [1] > st and score [1] < end:  
            print(score)
```



## Separation of Concerns - UI

- nur UI Funktionalität
- der Rest sind an `filterScore` delegiert

```
def filterScoreUI(scoreList):  
    st = input("Start score :")  
    end = input("End score:")  
  
    result = filterScore(scoreList, st, end)  
  
    for score in result :  
        print(score)
```



## Separation of Concerns - der Rest

- die Methode hat nur eine Verantwortung

```
def filterScore(scoreList, st, end):  
    rez = []  
  
    for p in lst:  
        if p[1] > st and p[1] < end:  
            rez.append(p)  
  
    return rez
```

# Separation of Concerns - das Testen

- die `filterScore()` Funktion kann so getestet werden

```
def filterScoreTest():  
    lst = [["Anna", 100]]  
  
    assert filterScore(lst, 10, 30) == []  
    assert filterScore(lst, 1, 300) == lst  
  
    lst == [["Anna"], 100], ["Ion"], 40], ["P"], 60]]  
    assert filterScore(lst, 3, 50) == [["Ion"], 40]
```



# Dependency/Abhängigkeit

- was ist eine Abhängigkeit?
- **Funktionen** - Eine Funktion ruft eine andere Funktion auf
- **Klassen** - Eine Klassenmethode ruft eine Methode einer anderen Klasse auf
- **Module** - Eine Funktion eines Moduls ruft eine Funktion eines anderen Moduls auf
- Gegeben sei die folgenden Funktionen **a**, **b**, **c** und **d**.
  - a ruft b an, b ruft c an und c ruft d an
- Was kann passieren, wenn wir die Funktion **d** ändern?



# Kohäsion

- wie gut eine Programmeinheit (eine Funktion/ein Modul) eine logische Aufgabe oder Einheit abbildet
- **starke Kohäsion:** alle Teile eines Moduls sollten mit anderen Teilen des Moduls zusammenhängen und voneinander abhängig sein.
- **schwache Kohäsion:** Teile eines Moduls haben keinen Bezug zu anderen Teilen
- viele Teile -> schwer zu verstehen!





# Kopplung

ein Maß, das die Stärke die Verknüpfung von verschiedenen Systemen, Anwendungen, oder Softwaremodulen beschreibt

Formen von Kopplung. Am Beispiel von der Klasse X zur Klasse Y:

- X ist direkte oder indirekte Unterklasse von Y
- X hat Attribut bzw. Referenz von Typ Y
- X hat Methode, die Y referenziert (Abhängigkeit)

# Generalisierung – Vererbung – Die Ist-Beziehung

- beschreibt eine Beziehung zwischen einer allgemeinen Klasse (Basisklasse) und einer speziellen Klasse
- die spezialisierte Klasse ist vollständig konsistent mit der Basisklasse, enthält aber zusätzliche Informationen (Attribute, Methoden, Assoziationen)
- ein Objekt der spezialisierten Klasse kann überall dort verwendet werden, wo ein Objekt der Basisklasse erlaubt ist





# Vererbung

- In objektorientierten Sprachen kann man (normalerweise) Klassen von anderen Klassen ableiten
- Die abgeleitete Klasse erbt Variablen und Methoden von der Basisklasse
  - Somit unterstützen die abgeleiteten Klassen die gleichen Methoden/Variablen wie die Basisklassen
  - und können überall dort benutzt werden, wo die Basisklasse benutzt werden kann
- So lange die abgeleiteten Klassen Methoden nicht überschreiben, verhalten sie sich in der abgeleiteten Klasse genauso wie in der Basisklasse



# Vererbung

```
class Face:
    def smile(self):
        print(": -)")
    def kiss(self):
        print(": - *")

class BabyFace(Face):
    def pokeTounge(self):
        print(": - P")
```

```
>>> boy = BabyFace()
>>> boy.pokeTounge()
: - P
>>> boy.kiss()
: - *
>>> boy.smile()
: - )
```



# Vererbung

```
class Person:  
    def __init__(self, name):  
        self.name = name
```

```
class KlingonGuy(Person):  
    def sayHello(self):  
        print("nuqneH " +  
              self.name)
```

```
class GermanGuy(Person):  
    def sayHello(self):  
        print("Hallo " +  
              self.name)
```

```
>>> g =  
GermanGuy('Stefan')  
>>> g.sayHello()  
Hallo Stefan  
>>> k =  
KlingonGuy("D'utoz")  
>>> k.sayHello()  
nuqneH D'utoz
```