

Zwischenprüfung

- 05. 12. um 18:00 Uhr, NTTSocrate
- 3-4 Aufgaben
- Code-Fragmente
- Sequenzielle Datentypen
- Dateien
- Matrix
- ...

Inhalt

- magic Operators
- lambdas
- List Comprehensions
- private/public in Python
- Klassenvariablen

magic operator



Kann man in Python eine Funktion an eine andere Funktion als Parameter übergeben?



magic operator

```
def s (a, b): return a+b
```

```
def p (a, b): return a*b
```

magic operator

```
def s (a, b): return a+b
```

```
def p (a, b): return a*b
```

```
def op (a, b, mop):  
    r = mop(a,b)  
    return r
```

```
print(op(1, 2, s)) #-> 3
```

```
print(op(2, 3, p)) #-> 6
```


Ich möchte die gerade Zahlen aus einer Liste filtern



magic operator

```
def filter (original_list):  
    filtered_list = []  
  
    for elem in original_list:  
        if elem % 2 == 0:  
            filtered_list.append(elem)  
  
    return filtered_list  
  
print(filter([1, 2, 3, 4]))
```

magic operator

```
def par (t): return t%2 == 0

def filter (original_list, op):
    filtered_list = []

    for elem in original_list:
        if op(elem):
            filtered_list.append(elem)

    return filtered_list

print(filter([1, 2, 3, 4], par))
```

Lambdas

- wenn man eine bestimmte Funktionalität benötigt
- → definiert man eine Funktion
- und man kann diese wiederverwenden
- aber was passiert, wenn man die Funktionalität nur einmal braucht?
- lohnt sich eine Funktion zu definieren?

Lambdas

- Lambda-Funktionen kommen aus der funktionalen Programmierung
- mit Hilfe des Lambda-Operators können **anonyme Funktionen**, d.h. Funktionen ohne Namen erzeugt werden

lambda Argumentenliste: Ausdruck

```
>>> s = lambda x, y : x + y
```

```
>>> s(1,2) #3
```

magic operator

```
def filter (original_list, op):  
    filtered_list = []  
  
    for elem in original_list:  
        if op(elem):  
            filtered_list.append(elem)  
  
    return filtered_list  
  
print(filter([1, 2, 3, 4], lambda el: el%2 == 0))
```


Lambdas

einfache Verarbeitung von Listen

- map
- filter
- reduce

map

```
r = map(func, seq)
```

- `func` ist eine Funktion und `seq` eine Sequenz (z.B. eine Liste)
- `map` wendet die Funktion `func` auf alle Elemente von `seq` an und schreibt die Ergebnisse in eine neue Liste

```
a = [1, 2, 3]
```

```
b = [4, 5, 6]
```

```
def mal2(el): return el*2
```

```
list(map(lambda el: el*2, a)) #[2, 4, 6]
```

```
list(map(mal2, a)) #[2, 4, 6]
```

```
list(map(lambda ela, elb: ela+elb, a, b)) #[5, 7, 9]
```

filter

`filter(funktion, liste)`

bietet eine elegante Möglichkeit diejenigen Elemente aus der Liste `liste` herauszufiltern, für die die Funktion `funktion` `True` liefert

```
a = [1, 2, 3, 4]
```

```
list(filter(lambda el:el%2 == 0, a)) #[2, 4]
```

reduce

`r = reduce(func, seq)`

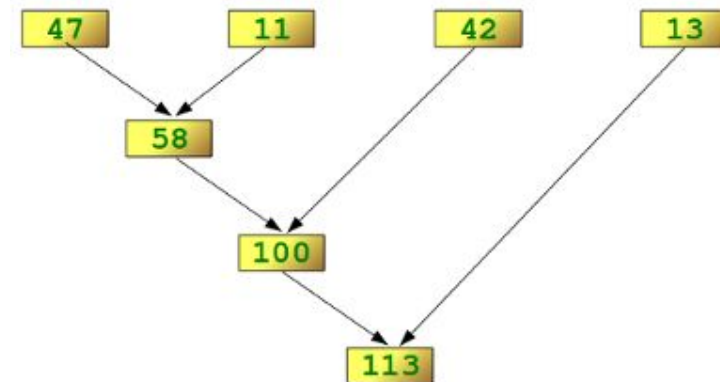
- wendet die Funktion `func` fortlaufend auf eine Sequenz an und liefert einen einzelnen Wert zurück.
- zuerst wird `func` auf die beiden ersten Argumente s_1 und s_2 angewendet.
- das Ergebnis ersetzt die beiden Elemente s_1 und s_2 :

`[func(s_1 , s_2), s_3 , ... , s_n]`

- im nächsten Schritt wird `func` auf `func(s_1 , s_2)` und s_3 angewendet.
- dies wird solange fortgesetzt bis nur noch ein Element übrig bleibt

`import functools`

```
functools.reduce(  
    lambda x, y: x + y, [47, 11, 42, 13]  
) #113
```



List Comprehension

- typischer Anwendungsfall
- man hat eine Liste und anhand dieser Liste möchte eine neue erstellen
- old-school Approach (was wir bis dato gemacht haben)

```
def filter (original_list, op):  
    filtered_list = []  
  
    for elem in original_list:  
        if op(elem):  
            filtered_list.append(elem)  
  
    return filtered_list
```

List Comprehension

eine elegante Methode Listen in Python zu definieren oder zu erzeugen

```
l = [1, 2, 3, 4, 5]
[e1*2 for e1 in l] #wie map
[e1 for e1 in l if e1%2 == 0] #wie filter
```

```
[(a, b, c) for a in range(1,30)
    for b in range(a,30)
    for c in range(b,30)
    if a**2 + b**2 == c**2]
# die pythagoreischen Tripel
```


List Comprehension

```
l = [0]*5
```

```
a = [1,2,3,5]
```

```
b = [1,4,3,4]
```

```
s = [i+j for i in a for j in b]
```

```
l = [i for i in range(len(a)) for j in range(len(b)) if a[i] == b[j]]
```

```
m = [ [0 for i in range(5)] for j in range(5)]
```

```
m = [[0] * 5] * 5 #problematisch
```

```
m = [[1 if i == j else 0 for i in range(5)] for j in range(5)]
```

Exkurs: Autos (mit List Comprehensions)

- Ein Autohaus hat Autos zu verkaufen
- Kunden haben Geld
- Kunden können vom Autohaus Autos kaufen



Exkurs: Autos (mit List Comprehensions)

- Ein Autohaus hat Autos zu verkaufen
 - Auto:
 - Attribute: Modell, Farbe, Baujahr, Preis
 - Methoden: tanken, anlassen
- Kunden haben Geld
- Kunden können vom Autohaus Autos kaufen



Exkurs: Autos (mit List Comprehensions)

- Ein Autohaus hat Autos zu verkaufen
 - Auto:
 - Attribute: Modell, Farbe, Baujahr, Preis
 - Methoden: tanken, anlassen
- Kunden haben Geld
 - Kunde:
 - Name, Betrag, Auto
 - Methoden: verdienen, kaufen
- Kunden können vom Autohaus Autos kaufen



Exkurs: Autos (mit List Comprehensions)

- Ein Autohaus hat Autos zu verkaufen
 - Auto:
 - Attribute: Modell, Farbe, Baujahr, Preis
 - Methoden: tanken, anlassen
- Kunden haben Geld
 - Kunde:
 - Name, Betrag, Auto
 - Methoden: verdienen, kaufen
- Kunden können vom Autohaus Autos kaufen
 - Autohaus:
 - Autos, Sold
 - Methoden: add_auto, verkaufen, sold_autos in 2024, sold_sum



Slots

- wie definiert man Attribute für eine Klasse
- innerhalb `__init__`

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

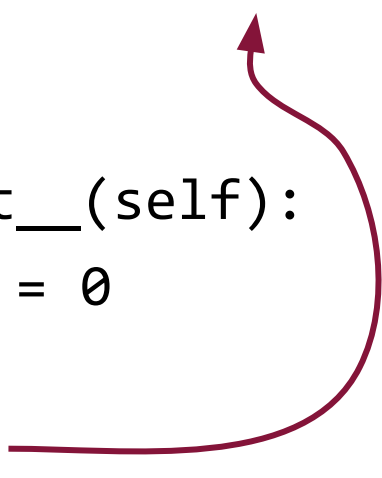
```
bob = Student("Bob")  
print(bob.name)
```

- aber man könnte auch später, nach der Definition der Klasse, Attribute dynamisch *definieren*

Slots

- Jedes Python-Objekt hat ein Attribut `__dict__`
 - das ein Dict ist und alle anderen Attribute der Klasse enthält
- z.B. für `self.attr` macht Python tatsächlich
 - `self.__dict__['attr']`

```
class T:
    def __init__(self):
        self.t = 0
t = T()
print (t.t)
print(t.x) #fehler, da x nicht definiert ist
t.x=10 #man definiert x
print (t.x) #ok
```



Slots

- `__slots__` wenn wir viele (Hunderte, Tausende) Objekte derselben Klasse instanziiieren möchten
- `__slots__` gibt es nur als Tool zur Speicheroptimierung
- `__slots__` soll nicht zur Einschränkung der Attribut-Erstellung verwendet sein

```
class T:  
    __slots__ = 'a', 'b'  
  
    def __init__(self, a, b):  
        self.a = a  
        self.b = b
```

Öffentliche Attribute

- alles ist in Python öffentlich

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

```
bob = Student("Bob")  
print(bob.name)
```

- man kann die Attribute direkt zugreifen...

Öffentliche Attribute

- wieso könnte das ein potenzielles Problem sein?

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

    def pay_10 (self):
        self.pay -= 10

bob = Student("Bob", 100)
bob.pay_10()
print(bob.name, bob.geld) # → 90
```

Öffentliche Attribute

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

    def do_bad_stuff(person):
        mine = person.geld
        person.geld = 0

bob = Student("Bob", 100)
dob = BadGuy("Dob")
dob.do_bad_stuff(bob)
print(bob.name, bob.geld) # → 0, poor bob :(
```

Öffentliche Attribute

- ein weiteres potenzielles Problem

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

    def pay_10 (self):
        self.pay -= 10

bob = Student("Bob", 100)
bob.geld = - 1000 # ← wie können wir sowas vermeiden?
print(bob.name, bob.geld) # → -1000
```


Öffentliche Attribute

```
class Student:
    def __init__(self, name):
        self.name = name
        self.noten = []

    def pruefung (self, note):
        self.noten.append(note)
```

Öffentliche Attribute

```
bob = Student("Bob")
```

```
bob.pruefung(10)
```

```
x = bob.noten
```

```
x.append(4)
```

```
print(x) # → [10, 4]
```

```
print(bob.noten) # → [10, 4]
```

Öffentliche Attribute

Klasse.Attribut = <Wert>

- können von außen zugegriffen werden
- als Klassenvariable: `klasse.variable`
- als Objektvariable: `objekt.variable`
- Klassenvariablen sind standardmäßig öffentlich

Private Attribute?

Klasse.__Attribut = <Wert>

- beginnen mit zwei Unterstrichen
- ein Zugriff von außen auf diese Attribute ist theoretisch nicht möglich*.
- werden mit Hilfe von get-Methoden zugegriffen.
- werden mit Hilfe von set-Methoden verändert.

***results may vary... (name mangling)**

Private Attribute?

Get/Set-Methoden

```
class Figur:
    def __init__(self):
        self.__yPos = 0

    def getPosY(self):
        return self.__yPos

    def setPosY(self, pos):
        self.__yPos = pos

f = Figur()
print(f.__yPos) # error
print(f.getPosY())
```

Private Attribute?

Get/Set-Methoden

```
class Figur:
    def __init__(self):
        self.__yPos = 0

    def getPosY(self):
        return self.__yPos

    def setPosY(self, pos):
        self.__yPos = pos

f = Figur()
f.__yPos = 10 # error
f.setPosY(10)
```

Private Attribute?

```
class T:  
    def __init__(self, a):  
        self.__a = a  
  
t = T(10)  
t.__a = 10 #error
```

Private Attribute?

```
class T:  
    def __init__(self, a):  
        self.__a = a
```

```
t = T(10)  
t.__a = 10 #error  
t._T__a = 10
```

das bedeutet, dass man Attribute weiterhin zugreifen könnte, solange man diese Konvention kennt

Schwache private Attribute

Klasse._Attribut = <Wert>

- beginnen mit einem Unterstrich
- nur als Info für den Aufrufer
- können von außen zugegriffen werden
- werden **nicht** durch die Anweisung `from ... import *` in eine Datei **importiert**

The Python way

```
class T:
    def __init__(self,x):
        self.__x = x

    @property
    def x(self):
        return self.__x

    @x.setter
    def x(self, x):
        self.__x = x
```

The Python way

- was bedeutet das für uns
 - wie werden wir Code schreiben?
- alle Attribute mit `__` ergänzen
- `@property` und `@setter` für jedes Attribut
- was gewinnt man
 - Information Hiding
 - Philosophie in Python ist ja unterschiedlich
 - wir sind hier alle Erwachsene
 - aber wichtig für andere Sprachen in der objektorientierten Welt

Attribute - Fazit

- man definiert alle Attribute innerhalb `__init__`
- jedes Objekt besitzt die gleiche Attribute, aber mit evtl. verschiedenen Werten
 - als Parameter für `__init__`
- um ein Attribute verwenden bzw. zugreifen zu können, muss man erstens ein Objekt erzeugen
 - d.h. die Attribute sind mit einer Instanz verbunden
- aber es gibt auch eine art von Attribute, die nicht mit einer Instanz verbunden

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 besitzen
- können von jedem Objekt der Klasse verändert werden
- sind globale Attribute eines Objekts

Klassenvariablen

```
class Student:
    #Klassenvariablen
    schule = 'Schule 101'

    def __init__(self, name, mat_nr):
        self.name = name
        self.mat_nr = mat_nr

bob = Student('Bob', 10)
print(bob.name, bob.mat_nr, Student.schule)

dob = Student('Dob', 20)
print(dob.name, dob.mat_nr, Student.schule)
```

Klassenvariablen

```
1 class Rechteck(object):
2     ANZAHL = 0
3
4     def __init__(self, b = 10, h = 10):
5         self.__hoehe = h
6         self.__breite = b
7         Rechteck.ANZAHL = Rechteck.ANZAHL + 1
8
9 r1 = Rechteck(10, 6)
10 r2 = Rechteck()
11 print(Rchteck.ANZAHL)
12
13 r3 = Rechteck()
14 print(Rchteck.ANZAHL, r1.ANZAHL, r2.ANZAHL, r3.ANZAHL)
15
16 r1.ANZAHL = 100
17 print(Rchteck.ANZAHL, r1.ANZAHL, r2.ANZAHL, r3.ANZAHL)
18
19 r4 = Rechteck()
20 print(Rchteck.ANZAHL, r1.ANZAHL, r2.ANZAHL, r3.ANZAHL, r4.ANZAHL)
```


Klassenvariablen

```
1 class Rechteck(object):
2     ANZAHL = 0
3
4     def __init__(self, b = 10, h = 10):
5         self.__hoehe = h
6         self.__breite = b
7         Rechteck.ANZAHL = Rechteck.ANZAHL + 1
8
9 r1 = Rechteck(10, 6)
10 r2 = Rechteck()
11 print(Rchteck.ANZAHL)
12
13 r3 = Rechteck()
14 print(Rchteck.ANZAHL, r1.ANZAHL, r2.ANZAHL, r3.ANZAHL)
15
16 r1.ANZAHL = 100
17 print(Rchteck.ANZAHL, r1.ANZAHL, r2.ANZAHL, r3.ANZAHL)
18
19 r4 = Rechteck()
20 print(Rchteck.ANZAHL, r1.ANZAHL, r2.ANZAHL, r3.ANZAHL, r4.ANZAHL)
```

```
2
3 3 3 3
3 100 3 3
4 100 4 4 4
>
```