Benutzerdefinierte Typen II

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
```

magic operator


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

def filter (l, op):
    ll = []
    for el in l:
        if op(el):
            ll.append(el)
    return ll

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

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
```

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)`

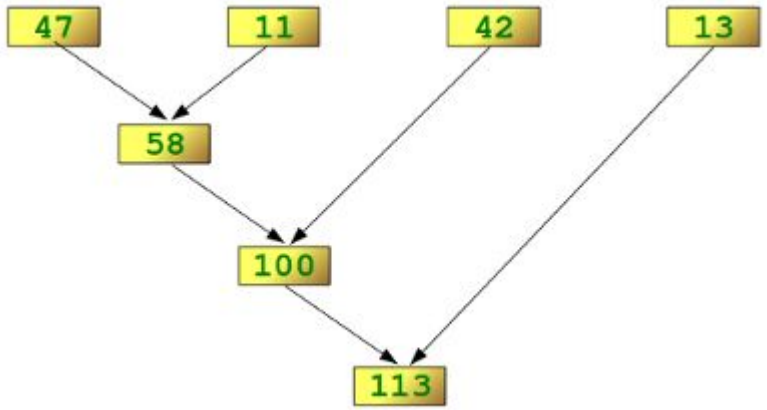
- wendet die Funktion `func` fortlaufend auf eine Sequenz an und liefert einen einzelnen Wert zurück.
- zuerst wird `func` auf die beiden ersten Argumente s_1 und s_2 angewendet.
- das Ergebnis ersetzt die beiden Elemente s_1 und s_2:

  $[func(s_1, s_2), s_3, \ldots, s_n]$

- 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

eine elegante Methode Listen in Python zu definieren oder zu erzeugen

```
l = [1,2,3,4,5]
[el*2 for el in l] #wie map
[el for el in l if el%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)]
```

Anwendung: Autos

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

Anwendung: Autos

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

Anwendung: Autos

- Ein Autohaus hat Autos zu verkaufen
  - Auto:
    - Attribute: Modell, Farbe, Baujahr
    - Methoden: tanken, anlassen

- Kunden haben Geld
  - Kunde:
    - Name, Betrag
    - Methoden: verdienen

- Kunden können vom Autohaus Autos kaufen

Anwendung: Autos

- Ein Autohaus hat Autos zu verkaufen
  - Auto:
    - Attribute: Modell, Farbe, Baujahr
    - Methoden: tanken, anlassen

- Kunden haben Geld
  - Kunde:
    - Name, Betrag
    - Methoden: verdienen

- Kunden können vom Autohaus Autos kaufen
  - Autohaus:
    - Autos, Sold
    - Methoden: add_auto, verkaufen

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
t.x=10
print (t.x) #ok
```

Slots

- `__slots__` wenn wir viele (Hunderte, Tausende) Objekte derselben Klasse instanziieren 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


```
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

```
def getPosY(self):
    return self.__yPos

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

Private Attribute?


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

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

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
```

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 Rechteck(object):
    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
```