Python ZF

Basistypen

bool

```
True
False
0
1

if (1):
   # always exectued

if (False):
   # never exectued
```

Zahlen

```
int     a = 2
float   a = 2.0       a = 2e-1
complex a = 1 + 2j

abs(-1) == 1
```

Sequenzen

```
str         a = "hallo"
list        a = [1,2,3]
tuple       a = (1,2,3)
bytes       a = bytes([255, 7, 0])
bytearray   a = bytearray([255, 7, 0])
dict        a = {"a" : 2, 3 : False}
set         a = {1, 1, 2}             a = set([1,1,3])
frozenset   a = frozenset([1,1,3])
```

rot = unveränderlich

MERKE → list set dict «LSD» veränderlich, Rest nicht.
(das bytezeug interessiert keinen *hust*)

Veränderlichkeit

Ein Tupel ist unveränderlich. Aber nur die Referenzen.
Das referenzierte Objekt kann ich ändern.

```
mylist = [1,2]
mytuple = (1, 'a', mylist)            → (1, 'a', [1,2])
mylist = [1,3]                        → (1, 'a', [1,3])
```

Integer sind Konstanten im Speicher und werden eigentlich nur referenziert, deshalb sind diese auch als unveränderlich klassifiziert.

Insbesondere ist string auch unveränderlich.

```
s = "abc"
s[1] = 'x'  → Error
```

Arithmetik

nichts überraschendes..

```
a = 10
b = 5
result = a - 5 * 2**3
#        a - 40
print(result)
```

Es gibt kein x++!!! x+=1 gäbe es Safe mit x = x+1

Type Checking

```
a = 1
type(a)  → int
type(21) → int
type(1.0)→ float
type([]) → list
type(()) → tuple
type("") → str

if type(a) == int:
    #wäre hier true
```

Bin, Oct, Hex

```
a = 0b1001                    Bsp:
a = 0o3177                    int i = 0o11
a = 0xA3F1                    print(i) → 9
```

Strings

Basics

Ein String ist in "" oder '' eingeklammert.
a = "I'm here"

Mehrzeilig:
a = '''abc
abc'''

b = "abc\nabc" a==b → true

r'Escaped nichts: \n bleibt als \n'

Slicing

mystring = 'abcdefg'

mystring[0]		'a'
mystring[-1]		'g'
mystring[2:]	ab & mit 2	'cdefg'
mystring[-2:]	vom 2tletzten bis Ende	'fg'
mystring[:2]	bis & ohne 2	'ab'
mystring[:-2]	Start bis&ohne 2tetztes	'abcde'
mystring[::2]	jeden zweiten	'aceg'
mystring[::-1]	reversed ;-)	'gfedcba'
mystring[-2:1:-2]	Beim Reversen vertauschen sich von/bis Indexe. Hier: Reversed jeden 2ten, von -2(f) bis und ohne 1(b)	'fd'
mystring[0:99:2]	hört Ende String einfach auf	'aceg'
mystirng[99]	IndexError	

Der string ist unveränderlich, wie ein Tupel:
a[0] = 'x' # Error

Methoden

```
x = mystring.upper()            'ABCDEFG'
x = mystring.lower()

x = mystring.split()            ['abcdefg']      #spaces!
x = mystring.split('e')         ['abcd', 'fg']
"Hallo du Sack".split()         ['Hallo', 'du', 'Sack']
x = mystring.splitlines()       Liste aller Zeilen

"  abc  \n".strip()             "abc"     → stript Whitespaces
".!,abc,!.".rstrip('.,')        ".!,abc,!"→ bleibt am ! hängen
```

"abcdef".partition("d")
→ Partitioniert in drei Teile, das links vom d, das d, und das rechts vom d:
["abc", "d", "ef"]

```
', '.join(['a','b','c'])  →    'a, b, c'


s = "ab"     d = "cd"
s += d   →   s == "abcd"
```

Mehr Methoden

Viele Prüfmethoden:
mystring.isalpha() mystring.isdigit() mystring.islower() …

Gross Klein:
lower(), uper(), swapcase(): Dürften klar sein.

capitalize(): Erstes Zeichen des Strings (nicht Wort) gross
title(): Erstes Zeichen jedes Wortes gross
casefold(): Wie lower, nur aggressiver (ß to ss)

```
str.count("Word") Zählt das Wort-Vorkommen
str.index("Word")  Wo kommt Wort erstmalig vor?
                   Wenn nicht -> ValueError
str.find("Word")   Wo kommt Wort erstmalig vor?
                   Wenn nicht -1
```

Alle Methoden auf Zusammenfassung!

String Format

```
"Wuff: {}   Meow: {}" .format("dog", "cat")
"Meow: {1}  Wuff: {0}".format("dog", "cat")
"One: {x}  Two: {y}   One: {x}".format(x="a", y="b")

Zeichenlänge modifizieren:
"{var:>8.2f}".format(var=somenumber)

var    Variable, die in die {} kommt, wie oben.
       0,1,… oder leer ginge auch.
:      Anzeiger, dass jetzt Formatbullshit kommt
>      rechtsbündig
^      centered
<      linksbündig
=      Vorzeichen ganz links, Zahl ganz Rechts
{:8}   Platz insgesamt mit allem inkl .
{:.2}  Präzision nach dem Komma
f      Float: 12.34 nicht 1.234e2
e      Exponential: 1.234e2
d      Dezimal
g      General - Schaut selbst was passt (?)
s      String
```

Zusätzlich gäbe es noch +, - oder " " das man für ein Vorzeichen(Platzhalter) nutzen könnte, oder ne 0, die dann 000123 schreiben würde.

Es gibt zur Formatierung noch Stringmethoden, etwa

```
"Python".center(14, '=')   →  ====Python====
"Python".ljust(14, '=')    →  Python========
```

Zahlen <-> Strings

Es gibt Konvertierungsoperatoren
int() float() str() btw: und auch list() set()

str()

Strings mit Zahlen / Listen addieren geht nicht implizit

```
a = 2
b = [1,3]
c = "hallo"

print(a + b + c)              # Error <int + list + string>
print(str(a) + str(b) + c)    # ok: "2[1,3]hallo"

print( "Soviel Autos: " + str(anzahl) )
```

String -> Zahl

Der String muss ganz streng eindeutig sein

```
int("2")     → 2
float("2")   → 2.0
int("2.0")   → Error
int("2e0")   → Error
float("2e0") → 2.0

int("101", 2)  → 2er Basis  4 + 0 + 1 = 5
```

Listen

```
simple = [1,2,3]
mylist = [1,2, "abc", [3.0, "hallo"], 3, True]

len(mylist) → 6          # Sublist ist ein einzelnes Element
```

mylist[0]		1
mylist [-1]		True
mylist [4:]	ab & mit 4	[3, True]
mylist [:2]	bis & ohne 2	[1,2]
mylist [::2]	jeden zweiten	[1, "abc", 3]
mylist [3]	Drittes Element	[3.0, "hallo"]
mylist [3:4]	List with List!	[[3.0, "hallo"]]

Modifizieren

```
mylist[0] = False
mylist = [False, 2, "abc", …]

mylist.insert(2, "Y")  #schafft mehr platz, kein overwrite
mylist = [False, 2, "Y", "abc", … ]

mylist.append("new")
mylist = [… 3, True, "new"]

mylist.append([1,2])   #List als ein Item
mylist = [… 3, True, "new", [1, 2]]

mylist.extend([3,4])   #Liste wird unwinded
mylist = [… 3, True, "new", [1, 2], 3, 4]

mylist += [5,6]        #wie extend
mylist = […,"new", [1, 2], 3, 4, 5, 6]

[1,2] + [3,4] = [1,2,3,4]
[1,2] * 3     = [1,2,1,2,1,2]
```

Methoden

```
mylist = [1, 2, 3, 4, 5, 6]

a = mylist.pop()
   a      → 6
   mylist → [1, 2, 3, 4 ,5]


a = mylist.pop(0)          #0 ist der index
   a      → 1
   mylist → [2, 3, 4, 5]

mylist.remove(5)           #5 ist das Element
mylist → [2, 3, 4]

mylist.reverse()  → [4,3,2]   #inplace
mylist.sort()     → [2,3,4]   #inplace
mylist.index(4)   → 2
mylist.count(4)   → 1

max(mylist)
min(mylist)

sorted(mylist)               #Gibt sortierte Kopie retour
```

Referenz

```
mylist = [1,2]
l = mylist     → l ist referenz.
                 Änderungen an l auch in mylist sichtbar.

safe = mylist.copy()
safe = mylist[:]         Range-Access macht ne Kopie.
                         Warnung: Aber nicht bei Numpy-Arrays!
```

Nested List

```
mylist = [1,2,['x','y','z']]

'y' == mylist[2][1]
```

Keyword 'in'

```
4 in [1,2,3]       → False
2 in [1,2,3]       → True
"bc" in "abcde"    → True
```

Dictionary: Prüft, ob der Key vorhanden ist:

```
d = {1:2, 3:4}         1 in d → True        2 in d → False
```

Unpacking

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

(x,y,z) = a    # weist x,y,z entsprechenden wert (s.o.) zu
```

Swap:

```
a,b = b,a      # Hax!!
```

List Comprehension

```
matrix = [[1,2,3],[4,5,6],[7,8,9]]

a = [ item for item in matrix ]
→ a = matrix           # item entspricht je einer subliste

a = [ item[1] for item in matrix ]
→ a = [2, 5, 8]
```

Beispiel:

```
x = [1,2,3,4]
out = []
for num in x:
   out.append(num**2)                  # uncool

→   out = [num**2 for num in x]    # cool
→   out = [num for num**2 in x]    # Error!
```

if-Statement:

```
list = [-1, 0, 5, -2]
a = [i for i in list if i < 0]
→ a = [-1, -2]
```

Mehrfach:

```
l1 = [2, 3]
l2 = [5, 7]

a = [ i*j for i in l1 for j in l2 ]
a → [10, 14, 15, 21]  # alle Combos "Kreuzprodukt"-mässig

a = [ i*j for i in l1 for j in l1 ]  #geht auch
a → [4, 6, 6, 9]
```

zip()

```
l1 = [1,2,3,4]    l2 = [9, 8, 7, 6]
```

Zip nimmt je ein Element von l1 und eins von l2, und macht daraus ein Tupel und fügt es in eine Liste:

```
list( zip(l1, l2) ) = [ (1, 9),
                        (2, 8),
                        (3, 7),
                        (4, 6)  ]
```

zip(l1, l2) → zip object, iterable, also der For-Loop darf direkt auf dem Zip-Object arbeiten ohne list-cast:

```
for (a,b) in zip(l1, l2):
    print("1234:" + str(a) + "9876:" + str(b))
```

Enumerate

Enumerate macht aus einer Liste eine Liste aus Tupeln, wo jeweils noch eine Numeration dabei ist:

```
list ( enumerate(früchte) )
[(0, 'apfel'), (1, 'banane'), (2, 'mango')]

for (n, frucht) in enumerate(früchte, start=7):
   print(n, frucht)

→  7 apfel
   8 banane
   9 mango
```

Tupel, Set, Dict

Tupel

Wie Liste, aber unveränderlich!

```
t = (1,"a",3)
t[0] = 4  #Error
```

Set

```
Syntax:  s = set( <list> )
  oder:  s = {1, 1, 2, "x"}

Vorsicht: type({}) = dict!
Leeres Set mit set() machen

Jedes Element ist nur einmal vorhanden:

s = set([1, 1, 2, "x"])
s.add(1)
s.add(3)
s.add("x")

print(s)
 → {1,2,3,'x'}   #unsorted, unique elements

s.add([1,2])  s.add({1,2})  s+=[1,2]     #je ein Error
```

Dictionary

```
Dictionaries sind Key-Value Pairs.
Der Key muss unveränderlich sein, z.B. keine Liste.

Definition
my_stuff = { "key1": 123,
             2:      3,
             "?":    "was?",
             5:      {
                         "i'm nested": True,
                         "magic":      "Grab me"
                     }
           }

Access
my_stuff["key1"]       → 123
my_stuff[2]            → 3
my_stuff[5]["magic"]   → "Grab Me"


Modify
my_stuff["key1"] = 321    → ändert value
my_stuff["neu"]  = [1]    → added Key mit Value

print(my_stuff)

{'key1': 321,
 2: 3,
 '?': 'was?',
 5: {"i'm nested": True, 'magic': 'Grab me'},
 'neu': [1]}
```

Methoden

```
d.items()   (Eigenartige) Liste der Key-Value Paare
            Genutzt zum Iterieren
            for (k,v) in my_stuff.items():
                print("Key: {}, Value: {}".format(k,v))

d.keys()    Liste der Keys
            for k in my_stuff.keys():
             print("Key: {}, Value: {}".format(k,my_stuff[k]))

d.values()  Liste der Values

d.pop(key)  Entfernt Key (und Value). Returned den Value.

list(d.items()) == list(zip(dic.keys(), dic.values()))
→ Beide identisch, Normale Listen der K/V Paare als Tupel


d.update(other_dict) → Fügt other_dict dem d hinzu.
```

Sortieren

```
Listen bieten die inplace Methode .sort() an

mylist = [2,1,3]
mylist.sort()
mylist  → 123

Es gibt die globale Methode sorted. Sie ist nicht inplace.

mylist2 = sorted(mylist)
```

key und reverse

```
Man kann der sorted-Fn eine Sortierungs-Funktion angeben.

Bsp:
mylist = ["abc", "z", "ab"]
sorted(mylist)  → default lexicographisch ['ab', 'abc', 'z']

sorted(mylist, key=len) → Der Länge nach: ['z', 'ab', 'abc']

sorted(mylist, key=len, reverse=True) →   umgedrehte Order

entsprechend die Inplace Methode:
mylist.sort(key=len, reverse=True)
```

Custom Sortkey / Lambda

```
Einen eigenen Key zum Sortieren macht man am besten als
Lambda.

Nicht wie bei Java oder C++ muss man nicht ein int bzw bool
returnen, der zwei Elemente vergleicht.
Man muss einfach das "Element" returnen, was man als
Sortierelement haben will. Dieses muss __lt__ ( < )
unterstützen.

Bsp1 Länge:
mykey = lambda arg: len(arg)
"abc" → 3
"z"   → 1
sorted(mylist, key=mykey)  →   ['z', 'ab', 'abc']

Bsp2: mylist.sort(key=lambda a: a[-1])
→ Sortiert nach letztem Zeichen
mylist == ['ab', 'abc', 'z']

Bsp3: Sortieren einer Liste aus Tupeln - ganz analog:
mylist = [ ('John',  'Miller'  , 46, 18.67),
           ('Randy', 'DerDandy', 48, 27.99)]

mykey = lambda arg: arg[-1]  #sortiert nach der 18.67 Spalte
```

Dict sortieren

```
Am besten auf obigen Fall zurückführen.
Dazu die Values und Keys einzeln holen und per zip() in
"Liste aus Tupeln"-Form bringen:

zip(dic.keys(), dic.values())

Sortiert by values:
sorted(zip(dic.keys(), dic.values()), key=lambda a: a[1])

Hinweis: Hier kann man direkt auf dem zip-Objekt arbeiten.
         Beachte, dass man normalerweise das zip-Objekt noch
         verlistisieren muss: list(zip(…))

Hinweis2:
sorted(dic.items(), key=lambda arg: arg[1])
funzt nicht!
```

Control Flow

Operatoren

1 > 2	False
1 <= 1	True
1 == 1	True
1 == "1"	False int immer != string
1 == True	True
1 == 1.0	True
1 != 2	True
a and b	
a or b	
not 1<2	not True → False

Operator Chaining

Python erlaubt chaining:

```
0 < a < 10       # 0 <  a  and  a  <  10
a == b == c      # a == b  and  b  == c
1 != 2 != 1      # 1 != 2  and  2  != 1   → True

BTW: Auch erlaubt:  a = b = 5;  → a = 5, b = 5
     aber           a,b   = 5 #Error, rechts müsste ein
                              iterable der Länge 2 sein
```

if else

```
if a>b:
    print("a is greater")
elif a<b:
    print("a is smaller")
    if a < 0:
        print("a is even negative")
else:
    print("a is b")
```

Inline if else

```
absolut = a if a >= 0 else -a
print('x positiv' if x >= 0 else 'x negativ')
```

Hinweis: Dies ersetzt die ?: Syntax anderer Sprachen.

Switch Case

Gibt es nicht! WTF.

for

Range:

```
for i in range(5):
    print(i)            # 0 bis 4


for i in range(2,5):    # NICHT range(2:5)
    print(i)            # 2 bis 4
```

```
Sidenote: Auch Collection aus range() machbar:
list(range(5))        →   [0, 1, 2, 3, 4]
set (range(0,10,2))   →   {0, 2, 4, 6, 8}
list(range(20,10,-2)) →   [20, 18, 16, 14, 12]
```

List:

```
mylist = [1,1,2,3,5,8,13]
for i in mylist:
    print(i)            # 1; 1; 2; 3; 5; 8; 13
```

Set:

```
myset = {1,2,3}
for i in myset:
    print(i)            # 1; 2; 3
```

Dict:

```
for (k,v) in my_stuff.items():
    print("Key: {}, Value: {}".format(k, v))

for k in my_stuff.keys():
    print("Key: {}, Value: {}".format(k, my_stuff[k]))

for k in my_stuff:
    print("Key: {}, Value: {}".format(k, my_stuff[k]))
```

→ Bei direkter Iteration über das Dict wird über den Key iteriert. Der Key ist immer das "Hauptelement" des Dicts.

BTW-Bsp:

```
a,b = {1:'a', 2:'b'}
→  a = 1, b = 2
```

Tuple Unpacking

Nicht nur bei Dict und Zip, sondern ganz allgemein nutzbar:

```
mypairs = [(1,2,3), (4,5,6), (7,8,9)]

for (i1, i2, i3) in mypairs:
   print("{}, {}, {}".format(i1, i2, i3)) #1, 2, 3 ; …
```

While

```
i=1
while i<5:
    print(i)          #1, 2, 3, 4
    i = i+1
```

Loop - Else

Nach einem While- oder For Loop kann ein Else Block kommen. Er wird ausgeführt, wenn die Schlaufe regulär beendet wird, ohne break.

```
while True:
    break
else:
    print("never exectued")  #weil der Loop per break beendet

while False:
    print("dead code")
else:
    print("always exectued")  #weil der Loop regulär beendet
```

Merke: Break bricht das while-Statement komplett ab inklusive des zugehörigen else-Blocks.

Funktionen

Syntax

```
def my_fn(param1, param2='default'):
    """
    Docstring
    """
    print(param1)
    print(param2)
    return param1 + param2


a = my_fun(1,2)       → a = 3
a = my_fun(1)         → Error 1 + "default"     int + string
a = my_fun("1", "2")  → a = "12"
```

Default-Argumente müssen jeweils am Schluss der Deklaration sein. Named Aufrufe ebenso.

Lambda

Syntax:
lambda param1, param2: <return-expression>

```
mylist = [1,2,3,4,5,6,7,8]

def is_even(num):
    return num%2 == 0

ist äquivalent mit

is_even = lambda num: num%2 == 0
```

filter, map, reduce

Standard-Methoden von Python, die Lambdas als Argumente nehmen

Filter: Liste mit Prädikat filtern.
filter(predicate, list)

```
out = filter(is_even,            mylist)   →   [2,4,6,8]
out = filter(lambda i: i%2 == 0, mylist)   →   [2,4,6,8]
```

Map: Liste auf etwas anderes ändern:

```
map( function(one_arg)   , one_list    )
map( function(arg1, arg2), list1, list2)

map ( lambda i: i*i, [3, 5] )
    → [9, 25]
map ( lambda i, j: i*j, [3, 5], [7, 11] )
    → [21, 55]
```

➔ Vgl List Comprehension. Ähnlich bzw gleich für ein Argument, aber nicht identisch bei zwei Argumenten.

Reduce: Akkumulieren

```
from functools import reduce
reduce(lambda x,y: x+y, [1,2,3])
1 + 2 + 3 = 6
```

* und ** als Paramter und Argument

*

Als Argument / Unwinding:

```
def foo(a, b, c):
   return (a,b,c)

foo(1,2,3)     → ok
foo([1,2,3])   → error, one argument given, 3 expected
foo(*[1,2,3])  → ok. Die Liste wird unwinded.
```

Als Parameter / Variadische Fn:

```
def foo(*args):
   return args[::-1]
```

args steht für beliebig viele Parameter. Es ist in der Funktion als TUPEL verfügbar(unveränderlich!).
Darum wird wieder ein Tupel returned bei unserem Bsp.

```
foo(1)         → (1)           #args = (1)
foo(1,2,3)     → (3,2,1)       #args = (1,2,3)
foo(*[1,2,3])  → (3,2,1)       #args = (1,2,3)
foo([1,2,3])   → ( [1,2,3] )   #args = ( [1,2,3] )
```

**

Als Parameter:

```
def bar(**d):
   for k in d:
      print(k, d[k])
```

d ist in diesem Fall ein ganz normales Dictionary. Der Aufruf von bar ist wie folgt möglich:

```
bar(salami="lecker", gewicht=100)
```

Wir können also random Argumente mit random Names angeben, die in bar als Dictionary als Key/Value Paare verfügar sind.

Beachte den Unterschied:

```
def bar(d):                  #ein argument, ein einziges
   for k in d:
      print(k, d[k])
bar({"salami":"lecker", "gewicht":100})  #ein dict als arg
```

Beim unteren haben wir ein einzelnes Dictionary Argument, beim oberen hauen wir direkt named random Argumente in variadischer Anzahl rein.

Als Argument:
Etwas abgefuckt. Hier wird das Dictionary unwinded.

```
def foo(name, age, size):
   print(name, age, size)

Normaler Aufruf:
foo("Tom", 32, 1.79)         #Alter Sack! Holy Shit.

der_tom = {"name":"Tom", "age":32, "size":1.79}
foo(der_tom)   #Error, 1 Arg given, 3 Expected
foo(**der_tom) #ok
foo(name="Tom", age=32, size=1.79)  #identischer Aufruf

käse = {"a":1, "b":2, "c":3 }
foo(**käse)         #Error, unexpected Keyword-Arg "a"
foo(a=1, b=2, c=3)  #Error, unexpected Keyword-Arg "a"
```

Scope

Lookup order:

1) lokal
2) innerhalb der Funktion
3) global
4) Built-In (len, str, …)

Beispiel

```
x=25

def foo()
   x = 50
   return x


print(x)       #25, global
print(foo())   #50, foo returned das function-local x aka 50

foo()
print(x)       #25, das Assignment findet geschlossen statt,
                kein Einfluss auf das global x
```

globale Variablen ändern

```
x=25

def foo()
   global x
   x = 50

print(x)       #25, global
foo()
print(x)       #50, global x ändert dann das globale x
```

Exceptions

```
try:
   file = open('nofile.txt', 'r')
   1/0
   a = undefinedname
   raise ValueError("So nicht!!")

except IOError:
   print("IO Error")

except ZeroDivisionError as e:
   print("Fehler: " + str(e))

except NameError:
   print("Name not found")

except:
   print("other error")

else:
   print("no error")

finally:
   print("always executed")
   if file:
      file.close();
```

Files

Simple

```
file = open("abc.txt", 'r')

text = file.read()
liste_der_zeilen = file.readlines()
  → Vorsicht: Readlines hat ein \n am Ende jedes Eintrags.

file.close()

f = open("abc.txt", 'w')

f.write('Hi')
f.writelines( [ "Line 1", "Line 2" ] )

f.close()

Modi: Create, Read, Write, Append
Diese sind mega strikt, also nur genau eins davon aufs mal!
```

With

```
with open("abc.txt", 'w') as file:
   file.write(…)
```

IOError

With macht das File auf jedenfall sauber zu, und man muss sich nicht kümmern. Ansonsten müsste man im else (noerror) f.close() aufrufen (nicht im finally)! Einfach with nehmen.

Bei with muss man einfach nur den IOError abfangen.

Zweimal Lesen

Wenn man ein File gelesen hat, ist der "Read-Pointer" am Ende des Files. Will man ihn erneut an den Beginn setze, um das File nochmal zu lesen:

```
sourcefile.seek(0)
```

Numpy Array aus File erstellen

Sei Text wie folgt:

```
   11,12
   21,22
```

Dann gibt np.loadtxt(text, delimiter=',') ein schönes Array:

```
[[11 12]
 [21 22]]
```

Iterator

KEIN PRÜFUNGSSTOFF

Eine Liste bietet als iterable die Methoden __iter__ und __next__ an.

```
liste = [1,2,3]
myiterator = iter(liste);
next(myiterator) → 1
next(myiterator) → 2
next(myiterator) → 3
next(myiterator) → StopIteration-Exception
```

Generator

KEIN PRÜFUNGSSTOFF

Ein Generator ist auch ein Iterator.

1. Funktion definieren, die yield enthält.
2. Die Ausführung der Fn einer Variablen zuweisen.
3. Mit next(var) oder for elem in var iterieren.

yield

```
def fibo():
    a = 0
    b = 1
    while True:         #Abbruchbedingung hier möglich
        yield b
        a, b = b, a + b

f = fibo()

next(f) → 1
next(f) → 1
next(f) → 2
…

f = fibo()  # Resets
```

```
i = 0
for elem in f:
    i+=1
    if (i > 100):         #Alternative Abbruchbedingung
        break
    print(elem)
```

Generator Expression

Genau wie List Comprehension, aber runde Klammern.
Der Generator sit danach direkt einsatzbereit, keine weitere Zuweisung zu einer Variable nötig.

```
gen = (i*i for i in range(1, 10))
next(gen) → 1
next(gen) → 4
next(gen) → 9
…
```

send

Mit send kann man sogar von aussen werte in den generator geben.
Man assigned das yield statement einer Variablen.

```
def counter():
    n = 0
    while True:
        wert = yield n
        if wert is not None:   #Also falls was gesendet wurde
            n = wert
        else:
            n += 1

f = counter()

next(f)    → 0
f.send(50) → 50
next(f)    → 51
```

Regex

Kein PRÜFUNGSSTOFF

```
import re

text = "I'm a random string in Python Code."
pattern = "in"
```

match = re.match(pattern, text)
- Prüft am String-Anfang
- none oder match-objekt

match = re.search(pattern, text)
- Sucht erstes auftreten
- none oder match-objekt

match = re.findall(pattern, text)
- Sucht alle Auftreten
- **Liste der Teilstrings** wird returend
- *['in', 'in']*

Regex Search Patterns

Da der \ ein Steuerzeichen ist muss der String ein r davor haben:
r'\d'

Platzhalterzeichen

\d	Digits	[0-9]
\D	not digits	[^0-9]
\s	Whitepsace	Space, \n, \t , …
\S	nicht Whitespace	
\w	Zahlen und Buchstaben	[a-zA-Z0-9_äöü…]
\W	nicht das obere	

Es gibt noch diverse weitere wie WortEnde, LinienEnde etc
→ Alle auf Prüfugnszusammenfassung

Steuerzeichen

*	0 bis n	sd*	s, sd, sdd, sddd…
+	1 bis n	sd+	sd, sdd, sddd…
?	0 bis 1	sd?	s, sd
{a}	a mal	sd{2}	sdd
{a,b}	a bis b	sd{1,3}	sd, sdd, sddd
[]	oder	s[dg]	sd, sg
[]	oder	s[a-c]	sa, sb, sc
^	not	[^abc]	alles ausser a,b,c

Bsp: Hex Zeichen
pattern = r'[a-fA-F\d]+'

Gruppen

Gruppen im Pattern sind mit () einzudingsen.

pattern = r'(ab)+c'

→ abc, ababc, abababc

\1 Gibt die Rückwärtsreferenz auf "die vorherige Gruppe"
Mit ?P<hallo> kanni ch die Gruppe benennen

(?P<name>ab) → ab gibt der Gruppe einen Namen
(?P<name>ab)\1 → abab ref per zahl
(?P<name>ab)(?P=name) → abab ref per name

Match Objekt

match.group(1) gibt die erste Gruppe zurück

```
m = re.search(r'(\d+) ([a-z]+)', '123 hallo welt!')
                Grp1: Zahlen
                      Grp2: Kleinbuchstaben

if m is not None:
    print('groups():', m.groups())
    print('group(0):', m.group(0))
    print('group(1):', m.group(1))
    print('group(2):', m.group(2))
else:
    print('keine Übereinstimmung')
```

```
groups(): ('123', 'hallo')   Liste aller Gruppen
group(0): 123 hallo          Ganzer Match
group(1): 123                Gruppe 1
group(2): hallo              Gruppe 2
```

---

match.group("name") gibt die benannte Gruppe zurück

```
m = re.search(r'(?P<zahl>\d+) (?P<wort>\w+)', '123 hallo
welt!')

print(m.group('zahl'))    → 123
print(m.group('wort'))    → hallo
```

Flags

```
match = re.search(
r'(?P<zahl>\d+)\.(png|jpg|gif)', text, re.IGNORECASE)
```

Die Flags kommen beim search als letztes ARgument hin.
Alle Flags sind auf ZF.

Substituieren

```
re.sub(pattern, substituat, text)
re.sub(r'\bschön\b', 'herrlich',
        'Das Wetter ist schön oder unschön.')
```

→ schön mit herrlich ersetzten
\b ist eine Wortgrenze

→ Das WEtter ist herrlich oder unschön.

Split

```
re.split(r'du', "Hallo du Sack")
['Hallo ', ' Sack']
```

Look-around Assertion

Man kann vorangehende oder nachfolgende Patterns definieren, die dann kommen müssen, aber nicht zum Match selbst zählen.

```
Bsp: Nach dem Match muss .doc folgen
re.findall(r'\w+(?=.doc)', 'bericht.doc dokument.doc')

Bsp: Nach dem Natch düfen keine Zahlen sein
re.findall(r'\w+(?!\d+)', 'abc123 cde')

Bsp: vor dem Match muss ein # sein
re.findall(r'(?<=#)\d+', '#10, #25, 66')

Bsp: vor dem Match darf kein # sein
re.findall(r'(?<!#)\d+', '#10, #25, 66')
```

Klassen

Grundlagen

```
class Dog():
   """ Docstring """

   ##Statischer Member
   species = "mammal"

   ##Statische Methode - kein self Argument
   @staticmethod
   def translate (word):
      return "Wuff" + word + "Wuff"

   ##Konstruktor
   def __init__(self, breed, name):
      self.breed = breed          ##Membervariablen
      self.name = name            ##unlike C++/C#/Java
                                  ##nicht vorab zu deklarieren
   ##Methode
   def bark(self):
       print("The {} {} barks!".format(self.breed, self.name))


mydog = Dog("Labrador", "Gini")

mydog.breed           → Labrador
mydog.bark            → "bound method"
mydog.bark()          → The Labrador Gini barks!
Dog.bark(self=mydog) → The Labrador Gini barks!

mydog.species         → mammal
Dog.species           → mammal

mydog.translate("wau2")        → Wuffwau2Wuff
Dog.translate("wau2")          → Wuffwau2Wuff

mydog.breed = "Husky"
mydog.breed             → Husky
mydog.randomshit = 1
mydog.randomshit        → 1

type(myodg) == Dog      → True
```

Access Modifier

Member und Methoden können private/prot/public sein:

```
class Foo():
    def __init__(self):
        self.pub = 'Ich bin öffentlich.'
        self._prot = 'Ich bin protected.'
        self.__priv = 'Ich bin privat.'

    def pub_funktion(self):
        print(self.pub)

    def _prot_funktion(self):
        print(self._prot)

    def __priv_funktion(self):
        print(self.__priv)
```

Properties

Trivialer Ansatz

- Private Member
- Public Access Methods

```
class Bank:
    def __init__(self):
        self.__guthaben = 0

    def get_guthaben(self):
        return self.__guthaben

    def set_guthaben(self, n):
        self.__guthaben = n

Contra: Ich muss immer set_ oder get_ schreiben.

b = Bank()
b.__guthaben       → bank has no attribute __guthaben
b.__guthaben = 2   → ok, but no effect
b.get_guthaben()   → still 0
```

Trivial enhanced

- Private Member
- Private Methoden
- Property explizit definiert
- Einfach zu merken

```
class Bank:
    def __init__(self):
        self.__guthaben = 0    #sollte eig setter hier schon usen

    def __get_guthaben(self):
        return self.__guthaben

    def __set_guthaben(self, n):
        self.__guthaben = n

    geld = property(__get_guthaben, __set_guthaben)

a = Bank()
a.geld = 5
print(a.geld)
```

@property

- Private Member
- Public Methoden mit Annotations
- Etwas schwieriger zu merken

```
class Bank:
    def __init__(self):
        self.geld = 0    #kann setter hier schon usen

    @property
    def geld(self):
        return self.__guthaben

    @geld.setter
    def geld(self, n):
        self.__guthaben = n

    def method_using_prop(self):
        self.guthaben = 0
        return self.guthaben

a = Bank()
a.geld = 5
print(a.geld)
```

```
class OnlyGetter:
    def __init__(self):
        pass

    @property
    def getit(self):
          return 5;
```

→ @getit.setter einfach weglassen

Special Member Functions

Special Member sind jeweils mit __<name>__ umgeben, z.B.

```
def __str__(self):
   return <mystringrepresentation>
```

Einige Special Members

- init #CTOR
- del #DTOR
- len #für len(…)
- abs #für abs(…)
- str #Konvertierungs-operatoren
- int
- float

Operatoren

- add #a + b
- iadd #a += b
- sub, mul, …
- truediv #a / b
- eq, lt, gt, lte, gte, … == < > etc Operatoren
- not
- xor, lfshit usw usf… gibt Tausende!!

Achtung: Wahrscheinlich möchtee man bei bank1+bank2 ein Bank-Objekt als Return. Das vergisst man gerne. Man darf allerdigns durchaus auch einfach ein int oder string returnen.

Beispiel:

```
def __add__(self, other):
    return Bank(self.__balance + other.__balance)
def __iadd__(self, other):
    return self + other   # nutze __add__
    # returned ein neues Objekt, modifizert nicht self [weird]

def __int__(self):
    return len(self.mymember)
def __str__(self):
    if type(self.mymember) == str:
        return self.mymember
    else:
        return "No String Representation available."



class Book():
   def __init__(self, name, pages):
        self.name = name
        self.pages = pages

   def __str__(self):                          #toString()
       return "Book " + self.name

   def __len__(self):
       return pages

   def __add__(self, other):
       newname = self.name + " and " + other.name
       newpages = self.pages + other.pages
       return Book(newname, newpages)          #new object

a = Book("Wetlands", 150);

print(a)    → Book Wetlands
len(a)      → 150

b = Book("random sheet", 1)
c = a+b
c.name → Wetlands and random sheet
c.pages → 151
```

Vererbung

```
class Animal():
   def __init__(self):
      print("Animal Created")
      self.eat()

   def eat(self):
      print("Eating")

   def walk(self):
      print("Walking")



class Dog(Animal):
   def __init__(self):
     print("Dog Created")

   def walk(self):              #implizites Override
      super().walk()            #super call
      super(Dog, self).walk()   #Alternativ
      Animal.walk(self)         #Alternativ
      print("Running")



d=Dog()   → Dog Created
            CTOR von Animal wird nicht gecalled!!
            Grund: Wird nicht aufgerufen mit
            super().__init__()
d.eat()   → Eating                          geerbt, funzt
d.walk()  → Walking Walking Walking Running     overriden
```

Super-CTOR-Call

Man sollte im CTOR der Subklasse unbedingt den Super-CTOR callen (in den meisten Fällen)

Das self Argument wird bei super-calls nicht mitgegeben, da wir ja aus dem objekt raus callen

Im folgenden Beispiel würde der Name nicht gesetzt, würde man den super().__init__(name) call weglassen.

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

class Angestellter(Person):
    def __init__(self, name, personalnummer):
        # Initialisierungsmethode der Superklasse aufrufen
        super().__init__(name)
        # oder Person.__init__(self, name)
        self.__persnomalnummer = personalnummer
```

Multiple Vererbung

Syntax:

```
class Foo(A, B, C):
   pass
```

Wenn man nun super().__init__ aufruft, sind alle argumente am besten named anzugeben

```
class A habe CTOR Arg aaa
class B habe CTOR Arg bbb und ccc

class Foo(A, B):
   def __init__(self, arg1, arg2, arg3, arg4):
      super().__init__(aaa=arg1, bbb=arg2, ccc=arg3)
      self.foomember = arg4
```

Da an die init-Fn von A und B zu viele Argumente mitgegeben werden, muss man dort noch ein **kwarg Parameter drin haben. Wenn die überflüssigen Args nicht nutzt ist es ja egal.

Module

Basics

In mymodule.py:

```
def func_in_module():
    print("haha");
```

In programm.py:

```
import mymodule
mymodule.func_in_module()
```

Advanced

Named import:

```
import mymodule as mm
mm.func_in_module()
```

Single Function import:

```
from mymodule import func_in_module
func_in_module()
```

Single Class import:

```
from mymodule import MyClass
a = MyClass()
```

Wildcard item import [better us import mymodule]

```
from mymodule import *
func_in_module();
```

Seite 13 bringt Unglück, also schnell weiter!

Numpy

```
import numpy as np

"Matlab für Python"

Trigo:  sin, cos, sqrt, exp, …
Linalg: Vektoren, Matrizen und deren Arithmetik
```

Grundsätzliche Funktionen

- np.pi π
- np.sin()
- np.exp()
- np.cumsum()
- np.log10()
- np.mean()
- …

Numpy Arrays

Klasse für Matrizen

Erzeugen

```
arr = np.array( [1,2,3] )

arr = np.array( [ [11,12,13], [21,22,23] ] )
```

Das Argument für np.array() ist eine Liste.
Gut: np.array([1,2,3])
Schlecht: np.array(1,2,3)
Bei 2dim Arrays ist es eine Liste aus Listen. [[…], […]]

Gängige Funktionen zur Erzeugung von arrays:

Funktion	Resultat
np.arange(3)	array([0, 1, 2])
np.ones((2,2))	array([[1., 1.], [1., 1.]])
np.ones_like(arr1)	array([1, 1, 1])
np.zeros((2,2))	array([[0., 0.], [0., 0.]])
np.zeros_like(arr1)	array([0, 0, 0])
np.full((2,2), 7.0)	array([[7., 7.], [7., 7.]])
np.full_like(arr1, 7)	array([7, 7, 7])
np.eye(2)	array([[1., 0.], [0., 1.]])
np.identity(2)	array([[1., 0.], [0., 1.]])
np.linspace(0, 1, 5)	array([0., 0.25, 0.5, 0.75, 1.])
np.logspace(0, 1, 4)	array([1., 2.1544, 4.6416, 10.])
np.random.randn(3)	array([0.7576, 0.0135, -0.8934])

Linspace: Von, Bis, Anzahl
Von und Bis sind included

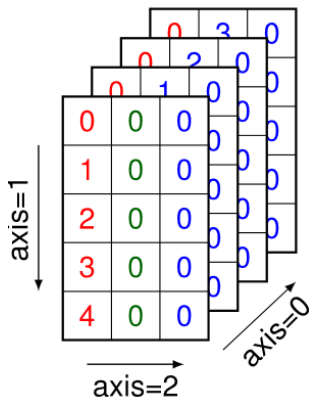
np.linspace(0,1,3) → [0, 0.5, 1]

Achsen und Dimension

```
arr.ndim  = 2        Vektor 1, Matrix 2, Würfel 3
arr.shape = (2, 3)   2 Reihen, 3 Spalten
```

Index:
arr[row, col]
arr[hinten, row, col]

Bei 3dimensionalen Arrays
ist die erste Dimension
diejenige nach hinten




```
arr  = np.arange(9)        →  [0,1,2, … , 8]
arr2 = arr.reshape(3,3)    →  [0,1,2],[3,4,5],[6,7,8]
```

Slicing

Slicing ist wie gewohnt auf die einzelnen Achsen anwendbar.
Sei shape(arr) == (3,3)

```
arr[:2, 1:]     Reihen 0 und 1
                Spalten 1 bis ende

arr[2]          nur die zweite Reihe        shape: (3)
arr[2,:]        ebenso                      shape: (3)
arr[2:,:]       Reihen ab 2                 shape: (1,3)
                alle Spalten
                → das gleiche wie oben aber als Matrix

arr[:, :2]      alle Reihen
                Saplten 0 und 1

arr[-1, :2]     Letzte Reihe
                Spalten 0 und 1
```

Dimensionen Advanced

```
s3 = np.arange(24).reshape(2, 4, 3)
                           2 hintereinander
                              4 Reihen
                                 3 Spalten
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]],

       [[12, 13, 14],
        [15, 16, 17],
        [18, 19, 20],
        [21, 22, 23]]])
```

s3[:,:,0] = ?
→ Alle "Karten" (hintereinander, siehe Bild links)
→ Alle Reihen (dim 2)
→ nullte Spalte

➔ 0,3,6,9 und 12,15,18,21
➔ [[0,3,6,9],[12,15,18,21]]

s3[…, 0] = ?
➔ Das Gleiche

Array Entwursteln:
np.ravel(s3) == np.arange(24)

Index in Dimensionen Codieren:
np.unravel_index(n, s3.shape)

```
n=0  →  (0,0,0)
n=3  →  (0,1,0)    # return ist tupel!
n=12 →  (1,0,0)
n=13 →  (1,0,1)
```

Arrays Stacken

Wenn ich Vektoren habe, wie entscheide ich, ob sie Reihenweise oder Spaltenweise in eine Matrix kommen??

```
a = np.array([1,2,3])

M = np.array([a,a,a])        ➔      Zeilenweise

                                array([[1, 2, 3],
                                       [1, 2, 3],
                                       [1, 2, 3]])
```

```
M = np.column_stack([a,a,a])  ➔   Columnweise

                                array([[1, 1, 1],
                                       [2, 2, 2],
                                       [3, 3, 3]])
np.c_[a,a,a]
```

Universelle Syntax:
`np.stack([a,a,a], axis=0)`

axis=0 nimmt die Reihen als Achse, also Reihenweise
axis=1 nimmt die Spalten als Achse, also Spaltenweise

Hab ich nun Arrays, und möchte sie als Karten hintereinanderlegen:
"Hintereinander"-Dimension ist die Dimension 0.
→ `np.stack([M,M,M], axis=0)`

Im Zweifel axis=-1 an der Ex!

Referenzen

Arrays sind auch bei np immer Referenzen.

```
arr2 = arr1          //änderungen in arr2 in arr1 sichtbar
```

Neu: Auch geslicete Arrays sind Referenzen:

```
arr2 = arr1[0:1, 2]  //änderungen in arr2 in arr1 sichtbar

arr_standalone = arr1[0:1, 2].copy()
```

Simples Rechnen mit Arrays

Identische Dimensionen
+, -, * rechnet komponentenweise
Achutng: Das * ist also kein Matrixprodukt!

Beispiele:

```
arr = np.array([1,2,3])
arr * arr → array([1,4,9])
arr > 2   → array([False, True, True])
```

zum Vergleich:
`[0, 1, 2] > 0  → TypeError: > not defined for  "list > int"`

Verschiedene, aber "passende" Dimensionen:
Das kleinere Element wird einfach ausgeweitet.
Die letzte Dimension muss übereinstimmen.

z.B.

```
1 1              3 4
1 1   +   2 3 =  3 4
1 1              3 4
```

Shape(3,2) + Shape(2) -> Letzte Dimension stimmt überein.

```
arr = np.arange(7)
arr[2:4] = 0 →    [0 1 0 0 3 4 5 6]
```

Funktionen:
Funktionen werden wie Operatoren komponentenweise angewendet:

```
x = np.linspace(0,10,100)
y = np.sin(x);
```

Numpy Rechenfunktionen

```
np.dot(M, v)       → Matrixprodukt / Skalarprdoukt
np.cross(v, w)     → Vektorprodukt
np.transpose(M)    → Transponieren   short: M.T
np.linalg.inv(M)   → M invertieren
np.linalg.norm(v)  → Norm von v = (a,b,c)
```

np.linalg.solve(A,b) → Löst Ax=b

`np.dot( np.linalg.inv(A) , b )` → Löst $Ax = b \Leftrightarrow x = A^{-1} \cdot b$

Sum unc Co entlang Achsen

```
np.sum(arr)  → eine Zahl

np.sum(arr, axis=0)  → Summiert entlang Reihen [bei 2d Matrix]
np.sum(arr, axis=1)  → Summiert entlang Spalten[bei 2d Matrix]
```

Vorsicht: Bei axis = 0 würd man ja denken (Achse=0 sind bei 2 Dimensionalen Matrizen die Reihen → Reihenweise). Aber man stelle sich ein Pfeil entlang aller Reihen vor (siehe Bild eine Seite vorher), und entlang dieses Pfeils wird dann summiert.

Genau gleich funzen np.max() und np.mean()

Wenn man etwa Reihenweise summiert, erhält man einfach einen Vektor. Wenn man schön die Reihen behalten möchte, keepdims=True setzen.

Bsp: Sei A = np.arange(9).reshape(3,3)

```
[[0, 1, 2],
 [3, 4, 5],
 [6, 7, 8]]

np.sum(A, axis=1)   →    [3, 12, 21]



np.sum(A, axis=1, keepdims=True)   →    [[3],
                                         [12],
                                         [21]]
```

Die Norm kann man auch entlang einer Achse berechnen, z.B. entlang der Reihen einer 3x3 Matrix:

`np.linalg.norm(s, axis=1)`

```
1 2 3
4 5 6     ➔   3.7, 8.8, 13.9
7 8 9
```

Matplotlib

```
import matplotlib.pyplot as plt
```

Grundsatz:

1. X-Werte mit linspace oder arange
2. Y-Werte daraus rechnen
3. plt.figure()
4. plt.plot(x,y, '.', label="Bla")
5. plt.show() oder plt.savefig("bla.pdf")

plt.plot()

Achtung, nicht auf ZF!

plt.plot() nimmt als Argumente:

1. x
2. y
3. style inkl farbe
4. label="Bla"

ohne x-Argument sind die x-Werte einfach 0, 1, 2, 3, 4, …

Mods vor dem plt.show()

```
plt.xlabel("x-Achsenbschriftung")

plt.grid(True)   → Gitternetz

plt.legend()         → Die Labels anzeigen
plt.tight_layout()   → Rand minimieren
```

Subplots

plt.subplot(2, 1, 1)
arg1: Wieviel Reihen?
arg2: Wieviel Spalten?
arg3: Wer bin ich?

statt plt… den return davon dann nehmen.

```
plt.figure()

ax1 = plt.subplot(2, 1, 1)
ax1.plot(np.random.randn(10));
ax1.set_xlabel('X1')
ax1.set_ylabel('Y1')

ax2 = plt.subplot(2, 1, 2, sharex=ax1)
ax2.plot(np.random.randn(10));
ax2.set_xlabel('X2')
ax2.set_ylabel('Y2')

plt.show()
```

Mit nur einem Statement:
fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)

Universeller Ansatz

Oft schreibt er in den Beispielen stat plt.figure() drum auch:

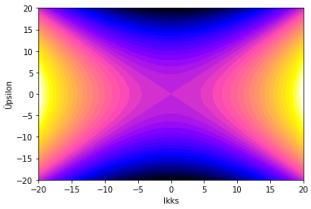
```
fig, ax = plt.subplots()
ax.plot(..)
ax.grid(..)
```

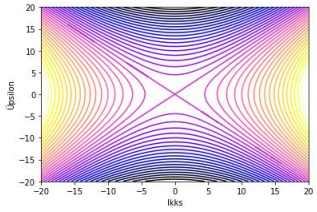
contour

Zum Printen von Höhenlinien und so.

Bsp: f(x,y) = x^2 - y^2

```
# 1) Alle Kombinationen von x und y mit meshgrid realisieren.

x = y = np.linspace(-20, 20, 1000)
X,Y = np.meshgrid(x,y)

# 2) Z ausrechnen mittels X und Y

Z = X**2 - Y**2

# 3) Figure wie vorhin
fig, ax = plt.subplots()

# 4) statt plt.plot() neu plt.contourf()
ax.contourf(X, Y, Z);
```





contourf() malt so mit Farben
contour zeichnet nur die contour-Lines (Höhenlinien)

Colorbar

```
# Kommt mit an Sicherheit grenzende Wahrscheinlichkeit nicht.
```

→ Die Contour Zeichnung gibt ein Return. Davon davon kann man dann die color-Skala holen. Diese kann man wiederum mit einem Label versehen.

```
zeichnung = plt.contourf(X,Y,Z)
colorbar = plt.colorbar(zeichnung)
colorbar.ax.set_ylabel("Zett")
```

Streamplot

Zeichnet Pfeile
Arg 1 & 2: Wo muss ich zeichen (X, Y vom Meshgrid)
Arg 3 & 4: Wo geht der Pfeil hin

```
plt.streamplot(X,Y,x,y)
```

loglog

plt.loglog() macht logarithmische Achsen, ansonsten aber gleich wie plt.plot()

Twinx

Wenn man zwei Kurven zeichnet, aber verschiedene Achsenbeschriftungen haben will:

```
fig, ax1 = plt.subplots()
ax2 = ax1.twinx()

ax1.plot(x, y1)
ax2.plot(x, y2)

ax1.set_ylabel("Bla")
ax2.set_ylabel("Other")
```

Scipy

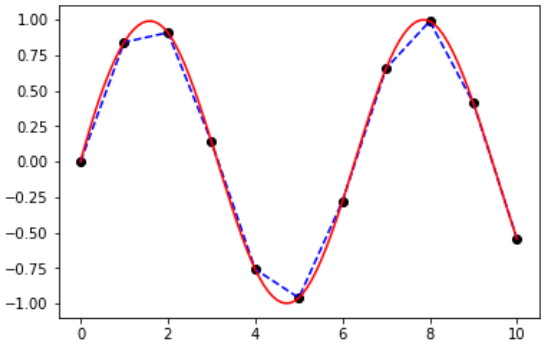
```
from scipy import integrate
from scipy import interpolate
from scipy import …

# Kaum Prüfungsrelevant

Library zum interpolieren, integrieren und so weiter.
```

trapz

```
trapz(x,y) interpoliert die Fläche unterhalb einer Kurve
(aka Integral)
```

interp1d

```
Interpoliert 1d-Daten

Beispiel:
x = np.linspace(0,10,11)
y = np.sin(x)

x_f = np.linspace(0,10,101)


f_inter = interp1d(x,y)   # return ist eine Funktion!
y_inter = f_inter(x_f)

f_inter_c = interp1d(x,y,kind='cubic')
y_inter_c = f_inter_c(x_f)

fig, ax = plt.subplots()
ax.plot(x,   y,         'ko',  label='Stützzwerte')
ax.plot(x_f, y_inter,   'b--', label="Linear")
ax.plot(x_f, y_inter_c, 'r-',  label="Cubic")

fig.show()
```



Ableiten

```
from scipy.misc import derivate

x = np.linspace(0, 2*np.pi, 100)
d = derivative(np.sin, x, dx=1e-6)  # func, x werte, dx
```

RectBivariateSpline

```
Interpolation für x,y,z Daten.
x,y sind gegeben und z sind die Werte


spline = RectBivariateSpline(self.X[0, :], self.Y[:, 0],
self.Z.T)

Evaluieren:

spline.ev(x_pos, y_pos)
```

Keyfindings aus den Übungen

Wichtigste / Interessante / Prüfungsrelevante Übungen

Palindrome Checker

In Python ein Einzeiler:

```
is_palindrome = lambda word: word == word[::-1]
```

Über Text Ierieren

Zeichen für Zeichen

```
for char in text:
```

Wort für Wort:

```
for word in text.split():
```

Line für Line

```
for line in text.splitlines():
   print(line)
```

Line für Line V2

Wenn man ein File readed, kann man gleich die Linien einlesen:

```
   for line in file.readlines():
```

statt erst readen und dann splitlines

Zeichen Zählen

Mit Dictionary geht es sehr leicht.

```
with open('text.txt', encoding="utf-8") as source:
    counts = {}
    with open ('text_result.txt', 'w') as target:
        for char in source.read():
            if char in counts:
                counts[char] += 1
            else:
                counts[char] = 1;
        target.write(str(resultdict));  #allenfalls noch schön
                                         iterieren für format.
```

Wörter zählen

Selber ansatz wie oben empfohlen. Wir machen es zur Übung aber anders: Zuerst legen wir ein set aller Wörter an, und danach zählen wir jedes vorkommende Wort.

```
with open('text.txt', encoding="utf-8") as source:
    content = set();
    with open ('text_result2.txt', 'w') as result:
        text = source.read()
        for word in text.split():
            content.add(word.strip(',.;:!?(){}[]<>'))

        for word in content:
            result.write(word + ": " + str(text.count(word)) + '\n')
```

List Comprehension

```
limit = 12
result = [(n1, n2) for n1 in range(1,limit+1)
                   for n2 in range(1,limit+1)
                   if n2%n1==0 and n1!=n2]
print(result)
```

What is the output?

→ Iteration über alle Paare n1, n2 zwischen 1 und 12
→ Ausgabe wenn man n2 durch n1 teilen kann und sie nicht gleich sind.

Also:

```
[(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1,
9), (1, 10), (1, 11), (1, 12),
(2, 4), (2, 6), (2, 8), (2, 10), (2, 12),
(3, 6), (3, 9), (3, 12),
(4, 8), (4, 12), (5, 10), (6, 12)]
```

Unique List

"Element soll in Liste nicht mehrfach vorkommmen"
→ Via set

```
mylist = [1,1,2,2,3]
temp = set(mylist)
mylist = list(temp)       → [1,2,3]
```

Generator Madness

```
def rr(gen):                #Generator als Argument
    while True:
        for elem in gen(): #Beachte: gen() startet bzw
                           #resettet den Generator
            yield elem

def mygen():
    yield 1
    yield 2
    yield 3

for elem in rr(mygen):
    print(elem)
```

Was Passiert?
Unser Generator ist zwar nach 3 Elementen fertig, aber rr führt ihn in der True-Schlaufe immer wieder aus. mit dem for elem in gen() wird der gen() immer neu generiert.
Output ist 123123123123123123123123 endlos

String.join mit List Comprehension

```
matrix = [
        [1, 2, 3, 4, 5, 6],
        [10, 20, 30, 40, 50, 60],
        [100, 200, 300, 400, 500, 600],
        ]

def matrix_to_string(matrix):
    result = "\n".join(
                    ";".join(str(zahl) for zahl in zeile)
                for zeile in matrix)

    result += "\n"
    result += ";".join(
       [str(sum(item)) for item in list(zip(*matrix))] )
    return result;



print(matrix_to_string(matrix))
1;2;3;4;5;6
10;20;30;40;50;60
100;200;300;400;500;600
111;222;333;444;555;666
```

Erklärung:
Oberer Teil:
Zuerst wird jede Zahl in jeder Zeile per ; getrennt dargestellt.
Jede Zeile wird im äusseren Layer mit \n gejoint.

Unterer Teil:

*matrix unwinded die matrix. Es gibt dann für zip drei Arugmente, und zwar

```
  arg1      [1, 2, 3, 4, 5, 6],
  arg2      [10, 20, 30, 40, 50, 60],
  arg3      [100, 200, 300, 400, 500, 600],
```

zip verwuselt das jetzt so, dass immer das erste von jedem Argument zusammen kommt, dann immer das zweite etc.
(1, 10, 100)
(2, 20, 200) etc

list macht dann ne liste draus:
[(1, 10, 100), (2, 20, 200), etc]

Jedes Tupel wird dann summiert und ausgegeben.

String.join mit List Comprehension Bsp2

```
"\n".join([
               " : ".join([item for item in line])
               for line in self.board
          ])
```

Regex Übung

```
import re

def replacer(word):
    l = len(word.group())
    result = ""
    for i in range(l):    #einfacher: r = "."
        result += "."     #           r*=len(word.group())

    return result;

text = "In diesem Beispiel werden alle Wörter, die bis zu vier
Zeichen lang sind, mit ebenso vielen Punkten ersetzt."

s = re.sub(r'\b\w{0,4}\b', replacer, text)

print(text);
print(s);
```

Regex Findall und Gruppen

```
import re

re.findall(r"(\w+)=(\w+)", "Ort=Rapperswil , PLZ=8640")
Out[103]: [('Ort', 'Rapperswil'), ('PLZ', '8640')]
```

→ Return ist eine Liste aller Matches. Jeder Match wird als Tupel returned wobei das Tupel die Gruppen enthält. Nicht-gruppen-zeichen (das =) kommen gar nicht erst zurück.

```
re.findall(r"\w+=\w+", "Ort=Rapperswil , PLZ=8640")
Out[104]: ['Ort=Rapperswil', 'PLZ=8640']
```

→ Return ist wieder eine Liste. Da wir keine Gruppen haben kommt aber jeder Match kommt direkt als String zurück inkl =

```
match = re.search(r"(\w+)=(\w+)", "Ort=Rapperswil , PLZ=8640")
→ Findet nur einmal das (erste) Paar!

match.groups()    →   ('Ort', 'Rapperswil')   #1 und 2, s.ff.
match.group(0)    →   'Ort=Rapperswil'   #ganzer match
match.group(1)    →   'Ort'
match.group(2)    →   'Rapperswil
```

Sorted List

Aufgabe Sorted List:

- Von List erben
- Alle Special Member Methoden werden einfach delegiert: super().__xy___(arg)
- Nach jedem Aufruf noch ein super().sort() und wir habens schon.

```
super().__xy___(arg):
Self gibt man hier nicht als Argument an!

class SortedList(list):

    def __init__(self, args):
        super().__init__(args)  ##argumente einfach weiterreichen
        super().sort()

    def __setitem__(self, i,x):
        super().__setitem__(i,x)
        super().sort()


    def __iadd__(self, b):
        super().__iadd__(b)
        super().sort()
        return self

    def __imul__(self, n):
        super().__imul__(n)
        super().sort()
        return self

    def append(self, x):
        super().append(x)
        super().sort()

    def extend(self, b):
        super().extend(b)
        super().sort()

    def insert(self, i, x):
        super().insert(i,x)
        super().sort()

    def reverse(self):
        pass

    def sort(self, key=lambda x: x, reverse=False):
        pass
```

Numpy und Zeichnen

```
import numpy as np
import matplotlib.pyplot as plt

# Zeitachse / x-Werte. Drei Möglichkeiten
t = np.linspace(0,1,101)    # 1 ist dabei
t = np.arange(0, 1, 0.01)   #1 wäre nicht dabei
t = np.arange(0,100)/100    #dito

# Zwei verschiedene y Werte
# Mit dem t einfach rechnen als wärs ne Zahl.
y1 = 5 * np.exp(-t/0.2) * np.cos(2*np.pi*10*t - np.pi/2)
y2 = 5 * np.exp(-t/0.2)

plt.figure()
plt.plot(t, y1,  'b-', label="Schwingung")
plt.plot(t, y2,  '--', label="Hüllkurve", color="grey")
plt.plot(t, -y2, '--', color="grey")
plt.xlabel('Zeit (s)')
plt.ylabel('Amplitude')
plt.legend(loc='center right')
plt.grid()
plt.show()
```

Array Quick-Creation

So ein Array einzugeben ist manchmal doof:

```
a = np.array( [ [1,2,3],
                [4,5,6],
                [7,8,9] ]
```

Möglichkeit:

```
a = np.array( [1,2,3,4,5,6,7,8,9] )
a = a.reshape(3,3)
```

Und noch schneller:

```
a = np.arange(1,10).reshape(3,3)
```

Propeprüfung Keyfindings

Ein vollständiges Progamm schreiben

`import numpy as np` nicht vergessen (Ist auf ZF!)
auch bei Pyplot.

Achten Sie auf die Details

Bei einem Plot:

- alle haben Farbe schwarz (wahrscheinlich wegen Druck)
  `plt.plot(x,y,'k-', label="y")`
  `plt.plot(x,y,'-', label="y", color="black")`
- von/bis bei den Achsen mit plt.xlim(a, b) setzen
  `plt.xlim(-2, 3)`
- Platzierung der Legende:
  `plt.legend(loc="lower left")`

  Mögliche Strings (u.a.)
  - lower left
  - upper right
  - upper center
  - center left
  - center
  - best

Comprehension Styles

```
a = [i*i for i in list]
```
Macht Liste, normale List Comprehension

```
a = {i for i in list}
```
Macht Set, normale Set Comprehension

```
a = {i*i:i for i in list}
```
Macht Dictionary mit Key / Value
Wäre hier {1:1, 4:2, 9:3, 16:4}

```
a = (i*i for i in list)
```
Generator, erlaubt next(a) zu machen

Klasse: Init und Reset

Wenn eine Aufgabe kommt, wo man eine Klasse machen muss, die sich resetten kann (z.B. Tic Tac Toe Spielfeld):

→ Methode Reset muss sowieso geschrieben werden.
→ Diese Methode in __init__ aufrufen zum initialisieren, so kann man das eigentliche Initialisieren sparen.

Aufgabe 3 - Vermischt

Alles waren **Einzeiler**. Hat man in der eigenen Lsg viele Zeilen evtl nochmal drüber schauen, ob es einfacher geht.
Oft mit Comprehension was ganz einfaches machbar.

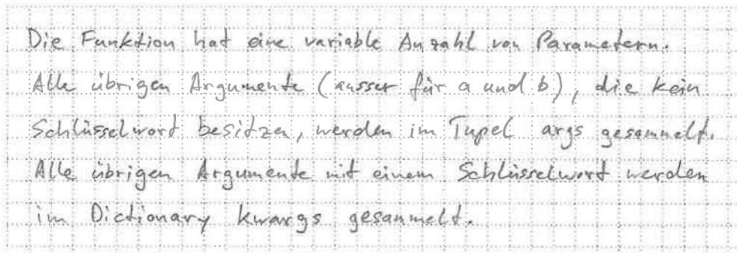
maxsplit

Maxsplit splittet n Elemente und tut den Rest in ne Gruppe
Total also n+1 Elemente

`"a b c d e f".split(maxsplit=3)` → `['a', 'b', 'c', 'd e f']`

* und **

```
def func(a, b, *args, **kwargs):
    print(a, b, args, kwargs)
```



Beachte:
args ist ein Tupel!
kwargs ein Dictionary.

Array Initialisiern

Muss man ein Array aka 9 Felder, alle mit "" initialisieren, den *-Operator nutzen:

```
["a", "b"]*3 = ['a', 'b', 'a', 'b', 'a', 'b']
["a"]*3      = ['a', 'a', 'a']
 [""]*3      = ['','','']
```

Falsch wäre:
`[[""]*3 ]*3 = [['', '', ''], ['', '', ''], ['', '', '']]`

auch falsch:
```
a = [""]*3
final = [a,a,a]
```

Dies ist jeweils eine Liste, die dreimal die gleiche innere Liste referenziert!

Richtig ist:
`final = [ [""]*3 for i in range(3) ]`
Im Zweifel manuell erstellen:
`final = [["","",""],["","",""],["","",""]]`

Python Syntax!

Doppelpunkte net vergessen:
`if self.board[row][col] != '':`
→ Manchmal recht versteckt.

Keine ; nach Zeilen machen ;-)

Keine Klammern bei if und whiles und so!

self.

net vergessen!
Wenn man eine Methode der eigenen Klasse aufruft, ist das self auch zu benutzen.

```
self.get_winner()
self.reset();
```

Denke immer an TicTacToe.reset(self);

nicht auf fig zeichnen

```
fig = plt.figure()
```

Der fig return braucht man da gar nicht!!

es ist trotzdem dann **plt**.plot(…)

ansonsten gibt es noch:
```
fig, ax = plt.subplots()
ax.plot()
```
Dann wird plt.xlabel('..') aber zu ax.set_xlabel('--') und so, darum lieber Finger davon!

Buchstabensalat

Wichtig für Palindrom- / Anagramm-Aufgaben.

```
s = "bcab"
sorted(s)        → ["a", "b", "b", "c"]
s.sort()         → AttributeError: 'str' has no sort method

x = list(s)      → ["b", "c", "a", "b"]
x.sort()         → ["a", "b", "b", "c"]

set(s)           → {"a", "b", "c"}
```
sorted(<set>) macht aus einem Set eine sortierte Liste
list(<set>) gibt nicht zwingend ne sortierte Liste aus!!

Achtung: Bei diesen Aufgaben ist wahrscheinlich der Case zu ignorieren!!

```
→ return sorted(a.lower()) == sorted(b.lower())
→ return a.lower() == a.lower()[::-1]
```