

Einsteiger, WS 2015/2016

Version: 17. Januar 2017, 19:38

Inhaltsverzeichnis

1	Linux Einführung	5
2	Rechtliches	6
3	Python	6
3.1	Erste Schritte	6
3.2	Rechenoperationen	7
3.3	Hilfe	7
3.4	Module	7
3.5	Mathematik	7
3.6	Turtle-Grafik	7
4	Grundlagen der Programmierung	8
4.1	Variablen	8
4.2	Verzweigungen	8
4.3	Die while-Schleife	9
4.4	Die for-Schleife und Listen	10
4.5	Dictionaries	11
4.6	Verschachteln	12
4.7	break und continue	12
4.8	Funktionen	13
5	Minimales Programm mit Tkinter	20
6	Klassen	21
6.1	Self	22
6.2	Der Konstruktor	22
6.3	Vererbung	23
6.4	Operatoren überladen	24
7	Plots mit matplotlib/pylab	25
7.1	plot	25
7.2	ndarray	25
7.3	Subplots	26

8	Dateien	27
8.1	Lesen und Schreiben von Textdateien	27
8.2	Binäres Lesen und Schreiben	28
9	Ranges und List Comprehensions	30
10	Mehr zu Matplotlib	31
10.1	Hilfe zu diversen Funktionen oder Modulen	31
10.2	Weitere plot-Funktionen	31
10.3	Achsenbeschriftung	31
10.4	Histogramme	32
10.5	Fits	33
10.6	Beliebige Funktionen	33
10.7	Einschub: Lambdas	35
10.8	Komplizierte Fits	36
10.9	Ausführliches Beispiel	36
11	3D-Graphik mit Matplotlib und Axes3d	39
11.1	Flächen	40
12	3D-Graphik mit Mayavi	44
12.1	Plots mit Mayavi	44
12.2	Animationen mit Mayavi	44
13	Einschub: Dekoratoren	47
14	Einschub: yield	49

Index

`=`, 8
`==`, 8

A

Abgeleitete Klasse, 21
Achsenbeschriftung, 31
`and`, 8
`arange`, 25
`*args`, 17

B

`pylab.bar`, 31
Basisklasse, 21
benannte Parameter, 17
Betriebssystem, 5
binäre Daten, 28
`break`, 12

C

callback, 19
`class`, 21
`continue`, 12
copy-on-write, 14, 15
`scipy.optimize.curve_fit`, 33

D

Dateien, 27
default-Parameter, 17
`dict`, 11

E

`elif`, 9
`else`, 9

F

Fakultät, 16
`False`, 8
`for`, 10
`array.frombytes`, 29
`array.fromlist`, 29
Funktionen, 13

G

`pylab.gca()`, 31

`pylab.gcf()`, 31
`global`, 18
globale Variable, 18

H

`help()`, 7
Hilfe, 7
`pylab.hist`, 32
Histogramme, 32

I

`if`, 9
`import`, 7
Indizierung, 10
`__init__`, 22
Instanz, 21
Interpreter, 6
`isinstance`, 23

K

Klassen, 21
Konstruktor, 22
`**kwargs`, 17

L

`lambda`, 35
`linspace`, 25
list comprehensions, 30

M

`math`, 7
Mathematik, 7
`matplotlib`, 25
Member, 22
Methode, 21
Module, 7

N

`ndarray`, 25
`not`, 9

O

`scipy.odr`, 36

`scipy.optimize`, 36
`or`, 8

P

Parameter, 13
`pylab.pie`, 31
`pylab.plot`, 25
`print()`, 6
`pylab`, 25

R

`readline()`, 27
`readlines()`, 27
Rekursion, 16
Rekursionstiefe, 16
return, 15

S

Schleifen, 9
`self`, 21, 22
Skriptsprache, 6
slices, 30
`str.split`, 27
`pylab.subplot`, 26

T

tkinter, 20
`array.tobytes`, 29
`array.tolist`, 29
`True`, 8
`turtle`, 7

U

überladen, 24

V

Variablen, 8
variadische Funktionen, 17
Vererbung, 23

W

`while`, 9

1 Linux Einführung

Linux¹ ist ein freies Betriebssystem, das von Linus Torvalds initiiert wurde. „Linux“ bezeichnet dabei den Kernel des Systems, also den Teil der für den Ablauf der Prozesse, die Verwaltung der Hardware etc. zuständig ist.

Die Tools (das „Userland“) kommen zumeist vom Betriebssystem GNU² der „Free Software Foundation“ FSF³.

Somit müsste das im Kurs verwendete Betriebssystem eigentlich „GNU/Linux“ heißen.

Das Betriebssystem wird zusammen mit vielen Anwendungen (meist ebenfalls freie Software) von anderen Organisationen bzw. Unternehmen verteilt. Dies wird *Distribution* genannt.

Das im Kurs verwendete Regensburger Linux „ReX“ basiert auf der Debian-Distribution⁴.

„Frei“ bedeutet hier nicht nur „umsonst“ sondern „Freiheit“. Je nach Lizenz kann die Software weitergegeben und verändert werden.

Genauer bitte der jeweiligen Lizenz entnehmen!

Interessante Befehle auf der Kommandozeile

Befehl	Nutzen	Beispiel
man	Hilfe für Befehle	man ls
info	Hilfe für Befehle	info rsync
chmod	Rechte ändern	chmod o-wx datei.py
ls	Dateien anzeigen	ls -la
cd	Verzeichnis wechseln	cd /media
mkdir	Verzeichnis anlegen	mkdir Beispiele
less	Text anzeigen lassen	less datei.py
jobs	Hintergrundprozesse	jobs
ps	Prozesse anzeigen	ps aux
ssh	Remote-Zugriff	ssh abc12345@phy300
rsync	Remote-Kopieren	
who	Benutzer anzeigen	who
cat	Dateien ausgeben	cat datei.py
top	Prozessorlast	top
find	Dateien suchen	find . -name '*.py'

¹ www.kernel.org

² www.gnu.org

³ www.fsf.org

⁴ www.debian.org

2 Rechtliches

Verschiedene Lizenzen:

Oft kommerzielle Lizenzen, oder freie Lizenzen („Copyleft“), z.B.”

- BSD
- GPL
- LGPL

und viele andere.

Manche dieser Lizenzen erlauben eine teilweise kommerzielle Nutzung.

Andere (vor allem die GPL) sind darauf ausgelegt, dass freie Software immer frei bleibt.

3 Python

Python ist eine freie Skriptsprache. Homepage ist www.python.org

Skriptsprache bedeutet, dass ein Python-Programm alleine nicht lauffähig ist, sondern dass man immer einen sogenannten *Interpreter* benötigt, um die Programme auszuführen.

Aktuell empfohlene Version: *Python 3.5*

3.1 Erste Schritte

Ein erstes Programm:

```
1 print("Hallo Welt")
```

In Python 3 ist `print` eine Funktion (daher die runden Klammern).

In Python 2 wäre `print` eine Anweisung, und das Programm würde so aussehen:

```
1 print "Hallo Welt"
```

Auf der Kommandozeile startet

`python3`

den interaktiven Interpreter,

`python3 dateiname.py`

führt das Programm `dateiname.py` im Interpreter aus. (Dabei ist `dateiname.py` eine normale Textdatei.)

3.2 Rechenoperationen

Mathematische Grundoperationen sind bereits definiert, die üblichen Vorrangregeln (Punkt vor Strich) werden beachtet. Vorrang wird ansonsten (in der üblichen Notation) mit Klammern geregelt.

```
1 7 + 3
2 1 + 2 * 3
3 (1 + 2) * 3
```

3.3 Hilfe

Mit der Funktion `help()` im Interaktiven Interpreter bekommen Sie Hilfe zu eingebauten Funktionen von Python.

Mit der Taste `q` verlassen Sie die Hilfe. Mit dem Vorwärtsslash `/` können Sie in der Hilfe nach Textteilen suchen.

3.4 Module

Module erweitern die Funktionalität des Interpreters. Mit

```
from modulname import *
```

machen Sie alle Objekte des Moduls verfügbar, mit

```
import modulname
```

machen Sie das Modul *an sich* verfügbar.

3.5 Mathematik

Das Modul `math` stellt die üblichen mathematischen Funktionen wie `sin`, `cos`, `sqrt` usw. zur Verfügung.

Im interaktiven Interpreter würden Sie folgendermaßen Hilfe zur Wurzelfunktion erhalten:

```
from math import *
help(sqrt)
```

Wenn Sie Hilfe zum ganzen Mathematikmodul brauchen, machen Sie folgendes im interaktiven Interpreter:

```
import math
help(math)
```

3.6 Turtle-Grafik

Das Modul `turtle` beinhaltet einfach zu bedienende Grafikfunktionen.

Beispiele für Zeichenfunktionen sind:

forward, shape, undo, right, left, penup, color, pendown, circle

Hilfe bekommt man im Interpreter, z.B. so:

```
from turtle import *  
help(color)
```

4 Grundlagen der Programmierung

4.1 Variablen

Variablen dienen z.B. dem Speichern von Zwischenergebnissen, können aber beliebige Objekte speichern. (In Python ist alles ein Objekt, und Python bestimmt den Typ von Variablen automatisch. Mehr dazu in einem anderen Kapitel.)

```
1 laenge = 80 * 3  
2 print(laenge)  
3 laenge = laenge / 7  
4 print(laenge)  
5 from math import *  
6 wuzwei = sqrt(2)  
7 print(wuzwei)
```

Hierbei dient der Operator = als *Zuweisung*. Der Ausdruck rechts vom = wird zuerst vollständig ausgewertet, danach wird das Ergebnis auf die Variable links vom = kopiert.

4.2 Verzweigungen

4.2.1 Vergleiche

Um zwei Werte oder Variablen miteinander zu vergleichen, gibt es in Python zahlreiche Operatoren. Beispielsweise:

Operator	Bedeutung
==	Gleichheit
!=	Ungleichheit
>	größer
<	kleiner
>=	größer oder gleich
<=	kleiner oder gleich

Alle diese Operatoren haben als Ergebnis einen logischen Wert, entweder **True** oder **False**. Logische Werte können in Python mit den Operatoren **and** und **or** kombiniert werden. Ergebnis

ist wieder **True** oder **False**. Der **or**-Operator ist das *logische* oder, also das inklusive oder. Die logische Negation geschieht mit dem Operator **not**.

4.2.2 if-else-Anweisung

Die if-else-Anweisung stellt eine Verzweigung im Programm dar, und ermöglicht die bedingte Ausführung von Code, z.B.:

```
1 a = float(input())
2 b = float(input())
3
4 if a > b:
5     print("a ist groesser als b")
6 else:
7     print("a ist nicht groesser als b")
```

Dabei steht nach **if** ein beliebiger Ausdruck, der als Ergebnis **True** oder **False** haben muss. Der eingerückte Block nach **if** wird ausgeführt, wenn der Ausdruck **True** ergibt. Andernfalls wird er eingerückte Block nach **else** ausgeführt.

In Python werden nach einer guten Konvention⁵ Einrückungen immer mit 4 Leerzeichen ausgeführt. Andere Konventionen sind möglich, sollten aber nur mit gutem Grund eingesetzt werden.

Eine Mehrfachverzweigung erstellt man mit dem Ausdruck **elif**. Beachte, dass Python die if-elif-else-Anweisung nach dem *ersten* Treffer verlässt:

```
1 a = 7
2 b = 8
3 if a > b:
4     print("eins")
5 elif b > a:
6     print("zwei")
7 elif b == 8:
8     print("drei")
9 else:
10    print("vier")
```

4.3 Die while-Schleife

Die while-Schleife bietet die Möglichkeit, einen Block abhängig von fast beliebigen Bedingungen zu wiederholen:

⁵ www.python.org/dev/peps/pep-0008/#indentation

```

1 while Bedingung:
2     Anweisung1
3     Anweisung2

```

Hierbei werden die Anweisungen im eingerückten Block wiederholt, solange die **Bedingung** als Ergebnis **True** liefert. Genauer: Vor jedem Durchlauf wird die Bedingung ausgewertet. Ist das Ergebnis **True**, wird der eingerückte Block ausgeführt, ist das Ergebnis **False**, wird der eingerückte Block übersprungen und die while-Schleife ist beendet.

```

1 a = 7
2 while a < 100:
3     print("a ist", a)
4     a = float(input())
5 print("fertig")

```

Obiges Beispiel bleibt in der **while**-Schleife, solange der Benutzer eine Zahl kleiner als 100 eingibt.

4.4 Die for-Schleife und Listen

Die for-Schleife erledigt Wiederholen, wenn die Art und Anzahl der Schritte bekannt sind. Beispiel:

```

1 for lauf in [1, 5, 3, 7, 3, 9, 2]:
2     print("quadrat", lauf ** 2)
3     print("kubik ", lauf ** 3)
4     print("---")

```

Dabei wird der Variablen **lauf** nacheinander jeder Wert der Liste zugewiesen, und der eingerückte Block ausgeführt.

Auch bei Schleifen wird der zur Schleife gehörende Block durch Einrückung gekennzeichnet. Auch hier empfiehlt sich die Konvention, mit vier Leerzeichen einzurücken.

Nach **for** steht also ein Variablenname, und nach **in** ein Ausdruck, dessen Ergebnis eine Sequenz ist. In obigem Beispiel ist dies ein sogenanntes „Array“.

Man kann sich Sequenzen auch automatisch erzeugen lassen, z.B. mit den Ausdrücken **range** (Python 2.x und Python 3.x) oder mit **xrange** (Python 2.x). Hierbei ist das erste Argument der erste Wert in der Sequenz und der Zweite derjenige, der gerade nicht mehr erreicht wird:

```

1 for lauf in range(2, 12):
2     print(lauf, " quadrat ist ", lauf * lauf)

```

Warum verhält sich **range** so seltsam? Die Indizierung von Arrays beginnt bei Null:

Die Funktion **len** gibt die Länge eines Arrays zurück, so dass es zwei Varianten gibt, über das Array zu iterieren:

```

1 arr = ["foo", "bar", "baz"]
2 print(arr[0]) # --> foo
3 print(arr[2]) # --> baz
4 print(arr[3]) # FEHLER!

```

Direkt über das Array:

```

1 arr = ["foo", "bar", "baz"]
2 for var in arr:
3     print(var)

```

oder indirekt über die Indices:

```

1 arr = ["foo", "bar", "baz"]
2 for i in range(0, len(arr)):
3     print(arr[i])

```

In der For-Schleife steht nach **in** immer ein sogenanntes „Iterable“. Als Iterable zählen Arrays (vom Typ **list**) die durch die eckigen Klammern gekennzeichnet sind, und Tupel, die durch runde Klammern gekennzeichnet sind. Dabei sind Tupel unveränderlich, während Arrays im Programm geändert werden können. (Z.B. kann man Elemente hinzufügen oder Löschen)

```

1 leer = [ ] # ein leeres Array
2 meus = ["foo", "bar", "baz"] # ein Array
3 deins = [3, 2.1, "bla", [3, "baz"] ] # gemischt!
4 for d in deins:
5     print(d)
6
7 tu = (3, 5, 'bla') # Tupel
8 for x in tu:
9     print(x)

```

Dabei können in einem Array verschiedene Datentypen gemischt werden.

4.5 Dictionaries

Dictionaries (auch assoziative Arrays genannt) sind vom Typ **dict**. Sie kennzeichnen sich durch eine Zuordnung von Schlüsseln (*keys*) zu Werten (*values*). Der Zugriff auf Elemente ist nahezu $\mathcal{O}(1)$.

Sowohl keys als auch values können beliebige Objekte sein. Es ist aber üblich, als keys nur Zeichenketten (strings) zu verwenden.

```

1  ha = { }                                # leeres Dictionary
2  h2 = { "bla": 7, 'nochwas': 67.89 }     # key/value Paare
3  print( h2['bla'] )                      # Zugriff auf Element
4  h2.update( { 'etwas':[1,2,3] } )       # Element ändern/einfügen
5  h2['bla'] = h2['bla'] + 1               # wie bei normalen Variablen
6
7  for k in h2:                            # (Siehe Text darunter)
8      print("key:", k, "value:", h2[k])

```

Verwendet man ein Dictionary als Iterable für eine for-Schleife (Zeile 7), so nimmt die Laufvariable (hier **k**) nacheinander die einzelnen *keys* an.

4.6 Verschachteln

Schleifen und Verzweigungen können beliebig verschachtelt werden. Die logische Zuordnung geschieht dabei durch Einrückung:

```

1  var = 77
2
3  while var < 100:
4      print("var ist", var)
5      print("nochwas")
6      print("noch viel mehr")
7      if var < 0:
8          print("negativ")
9      else:
10         print("positiv")
11     var = int(input())
12
13 print("FERTIG")

```

4.7 break und continue

Mit den Anweisungen **break** und **continue** lassen sich Schleifen noch feiner steuern.

Dabei bewirkt **continue**, dass eine Schleife (while oder for) sofort mit dem nächsten Schleifendurchlauf fortgesetzt wird. Nachfolgende Anweisungen in derselben Schleife werden übersprungen.

```

1  # Alle Zahlen ausgeben,
2  # die durch 17 teilbar sind
3  for i in range(0,1000):
4      if i % 17 != 0:
5          continue

```

```

6     print(i)
7     print("Ende")

```

break bewirkt, dass die Schleife sofort Verlassen wird, und der Programmablauf bei der ersten Anweisung welche nicht mehr zur Schleife gehört, fortgesetzt wird.

```

1  while True: # Endlosschleife!
2      # ganzzahligen Wert Einlesen:
3      variable = int(input())
4      if variable == 13:
5          break
6      print("das war", variable, "und ist ok")
7
8  print("Programmende")

```

4.8 Funktionen

4.8.1 Funktionsnamen

In Python sind Funktionen eigenständige Objekte. Der Funktionsname selbst bezeichnet die Funktion „an sich“, der Funktionsname mit nachgestellten Klammern bedeutet, dass die Funktion ausgeführt werden soll. Beachte den Unterschied bei:

```

1  from math import *
2  print(sqrt)
3  print(sqrt(2))

```

4.8.2 Parameter

Funktionen können verschieden viele Parameter haben. Wir kennen schon:

```

1  from turtle import *
2  forward(100)
3  circle(200, 120)

```

Auch selbst definierte Funktionen können Parameter haben:

```

1  def dreieck(laenge, farbe):
2      color(farbe)
3      forward(laenge)
4      left(120)
5      forward(laenge)
6      left(120)
7      forward(laenge)

```

Dabei werden die Parameter entweder anhand ihrer Reihenfolge identifiziert, oder anhand ihres Namens:

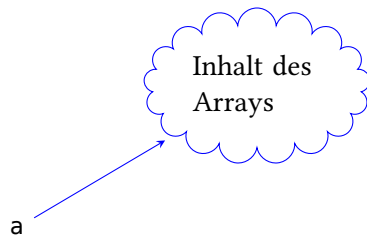
```
1 dreieck(170, "red")  
2 dreieck(farbe="green", laenge=150)
```

4.8.3 Parameter und „copy-on-write“

Python verwendet für Variablen und Parameter eine Technik, die manchmal „copy on write“ genannt wird. Dies wird besonders deutlich an Variablen mit einer komplizierten inneren Struktur, z.B. Arrays. Betrachten wir eine Arrayvariable a:

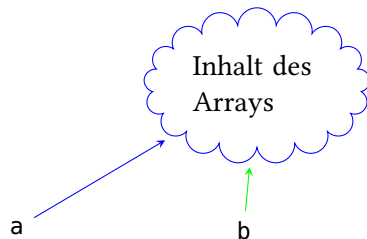
```
a = [1,2,3]
```

Hier wird im Speicher eine komplizierte Struktur angelegt, der Variablenname a verweist dann auf diese Struktur:



Macht man nun eine Zuweisung dieser Art: **b = a**

So bewirkt dies, dass nun die Variable b auf dieselbe Struktur hinweist:



Nun kann man sowohl über **a.append(7)**

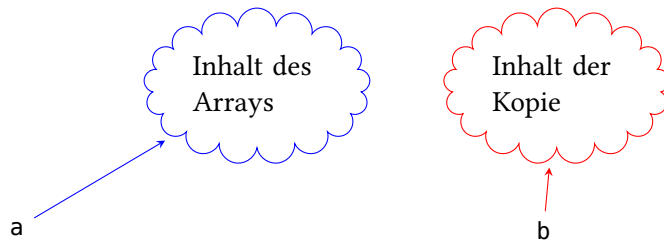
als auch so **b.append(8)**

Zugriff auf dieselbe Struktur erhalten.

Allerdings bewirkt diese Anweisung:

```
b = a + [7]
```

dass zuerst (wegen `a+[7]`) eine neue Struktur angelegt wird. Auf diese neu angelegte Struktur verweist dann b, und a und b verweisen nun tatsächlich auf verschiedene Strukturen.



Wegen dieser copy-on-write-Eigenschaft kann man zwar etwas an Parameter innerhalb von Funktionen etwas zuweisen, aber diese Zuweisung bleibt ohne Auswirkung außerhalb der Funktion, weil innerhalb der Funktion eine *Kopie* angelegt wird. So sind die Variable **name** in Zeile 4 und der Parameter **name** in Zeile 1 zwei unterschiedliche Objekte. Zwar wird in Zeile 5 eine Referenz übergeben, aber in Zeile 2 wird wegen copy-on-write eine Kopie angelegt, so dass das **name** innerhalb der Funktion eine Referenz auf die Kopie ist, die mit dem ursprünglichen **name** (also dem außerhalb der Funktion) nicht mehr verbunden ist.

```
1 def func(name):
2     name = 42
3
4     name = 77
5     func(name)
6     print(name) # gibt "42" aus
```

4.8.4 Rückgabewerte

Funktionen können Werte zurückgeben:

```
1 def kubik(wert):
2     return wert ** 3
3
4 z = kubik(7)
5 print(z)
```

Das ist nicht dasselbe, wie eine Ausgabe auf dem Bildschirm. Beachte den Unterschied:

```
1 def rueckgabe():
2     return "Irgendwas"
3
4 def ausgabe():
5     print("Etwas anderes")
6
7 x = rueckgabe()
8 y = ausgabe()
9 print("x ist", x, "y ist", y)
```

Will man mehrere Werte zurückgeben, so muss die Funktion ein Tupel zurückgeben. Dieses Tupel kann man bei der Zuweisung gleich wieder auflösen:

```
1 def wuerfel(seite):
2     flaeche = 6 * side**2
3     volumen = side**3
4     return flaeche,volumen
5
6 A,V = wuerfel(4.2)
```

4.8.5 Rekursion

Rekursion bedeutet, dass in der Definition einer Funktion die Funktion selbst wieder auftauchen kann.

Bekannt ist die rekursive Definition der Fakultät:

$$n! \mapsto \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

Dies lässt sich direkt in Python umsetzen:

```
1 def fakultaet(n):
2     if n == 0:
3         return 1 #Abbruchbedingung
4     else:
5         return n * fakultaet(n - 1) # Rekursionsschritt
```

Man sieht hier einerseits den Rekursionsschritt in Zeile 5, und andererseits die Abbruchbedingung in Zeile 3.

4.8.6 Rekursionstiefe

Die Rekursionstiefe ist im Allgemeinen begrenzt.⁶ Standardmäßig ist Python oft auf eine Rekursionstiefe von 1000 begrenzt, dies lässt sich jedoch ändern:

```
1 import sys
2 print "aktuelle maximale Rekursionstiefe ist", sys.getrecursionlimit()
3 sys.setrecursionlimit(100000)
4 print "maximale Rekursionstiefe ist nun", sys.getrecursionlimit()
```

⁶ Bei manchen Sprachen kann die Rekursion unendlich tief sein, falls der Rekursionsschritt die letzte Anweisung in einer Funktion ist. Dies wird als „tail recursion elimination“ bezeichnet. Python ist dazu jedoch nicht in der Lage.

4.8.7 Benannte Parameter

Anstatt die Parameter anhand der Reihenfolge zu identifizieren, in der sie übergeben werden, kann man auch die Namen der Parameter verwenden. Falls die Funktion so definiert ist:

```
def fun(peter, paul, hans):
```

so sind die folgenden Aufrufe gleichwertig:

```
fun(1,2,3)
```

```
fun(peter=1, paul=2, hans=3)
```

```
fun(hans=3, paul=2, peter=1)
```

```
fun(1, hans=3, paul=2)
```

Man kann also auch benannte und nicht-benannte Parameter mischen.

4.8.8 default-Parameter

Parameter können default-Werte haben. Sprich: falls der Benutzer diesen Parameter ganz lässt, wird der default-Wert eingesetzt.

Ist die Funktion also so definiert:

```
def fun(peter, paul=2, hans=3)
```

so sind die nachfolgenden Zeilen äquivalent:

```
fun(1)
```

```
fun(1,2
```

```
fun(1,2,3)
```

Default-Werte können überschrieben werden:

```
fun(7,8,9)
```

```
fun(hans=9, paul=8, peter=7)
```

sind also auch gleichwertig.

4.8.9 variadische Funktionen

In Python gibt es zwei Möglichkeiten, variadische Funktionen (Funktionen mit einer veränderlichen Anzahl an Parametern) zu definieren.

Beginnt ein Parametername mit einem Stern * (üblicherweise nennt man diesen Parameter dann ***args**) so sammelt dieser Parameter alle überzähligen, unbenannten Parameter in einem Array.

Beginnt ein Parametername mit zwei Sternen ** (üblicherweise nennt man diesen Parameter dann ****kwargs**) so sammelt dieser Parameter alle überzähligen *benannten* Parameter in einem Dictionary. So wie in Zeile 6

Auch die Umkehrung ist möglich. Der Ausdruck `*itr` innerhalb der runden Klammern eines Funktionsaufrufes bildet die Elemente in dem Iterable `itr` auf die Parameter der aufzurufenden Funktion ab, wie in Zeile 12.

```
1 def fun2(x,y,z):
2     print('fun2x', x)
3     print('fun2y', y)
4     print('fun2z', z)
5
6 def fun(a,b,c,*args,**kwargs):
7     print(args)
8     print(kwargs)
9     print('x ist', kwargs['x'])
10    fun2(*args)
11    ary = ['ih', 'uh', 'ah']
12    fun2(*ary)
13
14 fun(1,2,3,9,"hallo", 'bla', nix='los', x='u')
```

4.8.10 Globale Variablen

Variablen können global, das heißt außerhalb von Funktionen definiert sein. Eine Zuweisung innerhalb einer Funktion an eine Variable gleichen Namens bewirkt, dass nun die lokale Variable die globale verdeckt. Will man explizit in einer Funktion auf eine globale Variable Bezug nehmen, kann man die Anweisung *global* verwenden:

```
1 x = 7
2
3 def f():
4     print x
5
6 def g():
7     x = 9
8
9 def h():
10    print x
11    x = 9
12
13 def hh():
14    global x
15    print x
16    x = 9
```

In Zeile 10 wird eine lokale Variable x angesprochen. Sie heißt zwar genauso wie die globale Variable x , die in Zeile 1 angesprochen wird, verdeckt („shadows“) aber diese. Zuweisungen an die lokale Variable haben keinen Effekt außerhalb der Funktion. Die Anweisung *global* in Zeile 14 bewirkt, dass Python nun keine lokale Variable anlegt, sondern dass sich nun in der Funktion *hh* das x auf das globale, in Zeile 1 definierte x bezieht. Somit hat auch die Zuweisung an x in Zeile 16 auch einen Effekt außerhalb der Funktion *hh*.

4.8.11 Funktionen als eigenständige Objekte

Funktionen sind in Python vollwertige Objekte. Sie können in Variablen gespeichert werden, und aus als Parameter an andere Funktionen übergeben werden:

```
1  def f2():
2      return 'ahaha'
3
4  var = f2
5  print( var() )
6
7  def aufruf(etwas):
8      print('Das ist', etwas() )
9
10 aufruf(f2)
```

In Zeile 4 wird die Variable **var** mit dem Objekt einer Funktion belegt. Beachte dass in dieser Zeile hinter **f2** *keine* runden Klammern stehen! Die Funktion **f2** wird hier nicht aufgerufen. Erst wenn man in Zeile 5 hinter das Funktionsobjekt (das jetzt **var** heißt) runde Klammern schreibt, geschieht der Aufruf.

Analog dazu nimmt die Funktion **aufruf** einen Parameter, hinter den dann in Zeile 8 die runden Klammern kommen. Wenn man dann nun wie in Zeile 10 eine Funktion als Parameter übergibt (beachte dass hier wieder hinter der Funktion **f2** *keine* runden Klammern stehen), dann wird in in Zeile 8 die Funktion ausgeführt. Diese Technik heißt auch *callback*.

5 Minimales Programm mit Tkinter

Das Modul **tkinter** verwendet diese *callback*-Technik, wenn man einem Button mit Funktionalität belegen will. Der nachfolgende Code ist ein minimales Beispiel für eine einfache Anwendung mit einem einzelnen Button.

```
1 from tkinter import *
2 status = 1
3 def tuwas():
4     global status
5     status = status + 1
6     print("ich tu was", status)
7
8 app = Tk()
9 b = Button(app)
10 b.configure(text="Na so was")
11 b.configure(command=tuwas)
12 b.pack()
13 mainloop()
```

In Zeile 2 wird eine Variable **status** definiert. Die in Zeile 3 definierte Funktion arbeitet mit dieser Variablen. Um anzuzeigen, dass die Variable außerhalb der Funktion definiert wurde (also nicht innerhalb der Funktion neu angelegt werden soll) schreibt man in Zeile 4 das Wort **global**.

Danach legt man (Zeile 8) für die Anwendung an sich eine Variable vom Typ **Tk** an. Zeile 9 legt einen Button an, wobei man die Variable, welche die Anwendung beinhaltet (hier **app**) als Parameter übergeben muss. Zeile 10 definiert, was auf dem Button steht, Zeile 11 definiert, was der Button macht, wenn man darauf klickt. Schließlich generiert man in Zeile 12 ein automatisches Layout der Anwendung. Zuletzt startet man in Zeile 13 die Anwendung.

Statt **.pack()** kann man auch **.grid()** verwenden, um ein rasterartiges Layout zu erzeugen. **.grid()** nimmt die benannten Parameter **column** und **row** um die Elemente anzuordnen.

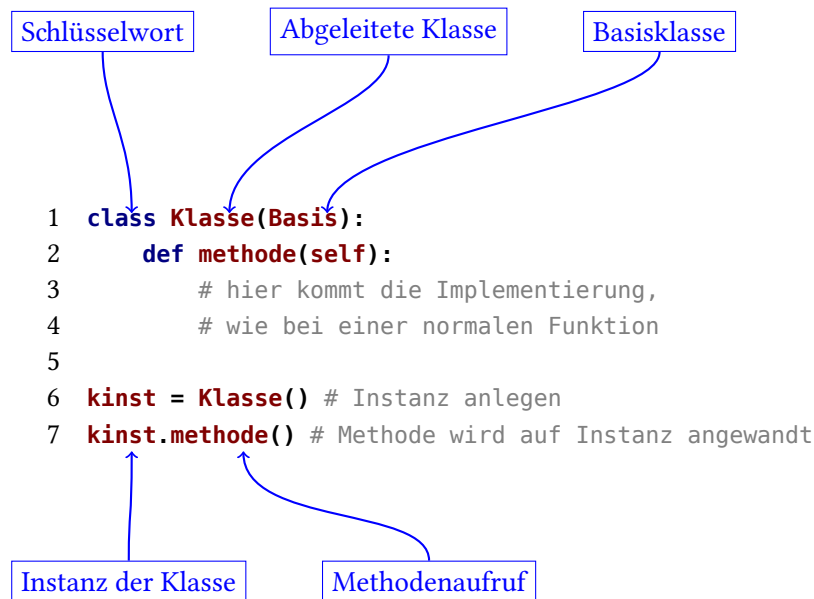
Eine gute Referenz finden Sie hier:

Allgemeine Themen zu Python: <http://effbot.org/>

Eine Anleitung zu Tkinter: <http://effbot.org/tkinterbook/>

Die Tkinter-Referenz: <http://effbot.org/zone/tkinter-index.htm>

6 Klassen



Klassen werden durch die `class`-Anweisung begonnen. Der Klassenname kann frei gewählt werden, nach denselben Regeln wie die Namen für Funktionen oder Variablen. Einer gängigen Konvention nach wählt man die Namen selbst definierter Klassen so, dass sie mit einem Großbuchstaben beginnen.

Funktionen, die innerhalb einer Klasse definiert sind, nennt man *Methode*. Dabei können bestehenden Klassen neue Methoden hinzugefügt, bzw. bestehende Methoden geändert werden. Die abgeleitete Klasse *erbt* hierbei alle Methoden der Basisklasse.

Hierbei werden Methoden stets mit einem Parameter mehr definiert, als später im eigentlichen Aufruf erscheinen, denn die Instanz selbst wird der Methode als erster Parameter übergeben. Die Konvention ist, diesen ersten Parameter stets `self` zu nennen. (Das ist eine reine Konvention, macht aber den Code für andere Programmierer deutlich lesbarer.)

Für neue Klassen soll die Basisklasse immer **`object`** sein!

6.1 Self

Methoden können über **self** hinaus noch weitere Parameter haben. Beim Aufruf einer Methode wird **self** durch die Variable ersetzt, die vor dem Punkt steht:

```
1 class Klasse(object):
2     def pension(self):
3         return 67 - self.alter
4     def geburtstage(self,n):
5         self.alter = self.alter + n
6
7 einer = Klasse()
8 einer.alter = 55
9 einer.geburtstage(2)
10 print einer.pension()
```

Hier wird in Zeile 9 das **self** aus Zeile 4 mit **einer** belegt, und **n** wird mit **2** belegt. In Zeile 10 wird das **self** aus Zeile 2 mit **einer** belegt.

6.2 Der Konstruktor

Der Konstruktor ist eine besondere Methode namens **__init__** (wirklich mit zwei führenden und zwei folgenden Unterstrichen!) die immer dann automatisch aufgerufen wird, wenn eine neue Variable vom Typ einer Klasse aufgerufen wird.

Dies wird zumeist dafür verwendet, Felder (Member) der Klasse mit sinnvollen Werten zu belegen:

```
1 class Klasse(object):
2     def pension(self):
3         return 67 - self.alter
4     def __init__(self,alter):
5         print "Hallo!"
6         self.alter = alter
7
8 einer = Klasse(37)
9 print einer.pension()
```

Hier hat z.B. jede Klassenvariable (Instanz) automatisch ein Feld **alter**, das im Konstruktor belegt wird.

6.3 Vererbung

Klassen können voneinander erben. Im nachfolgenden Beispiel „erbt“ die Klasse **Mitarbeiter** von der Klasse **Person**. Mitarbeiter ist somit auch eine Art Person, aber mit Mehr Information und spezielleren Aufgaben.

Damit hat die Mitarbeiter-Klasse alle Member und Methoden, die auch die Person-Klasse hat. Erhält jedoch die abgeleitete Klasse (hier „Person“) auch ihren eigenen Konstruktor, so muss dieser den Konstruktor der Basisklasse explizit aufrufen. Dies geschieht hier in Zeile 12. Die Funktion **super()** sucht dabei die richtige Basisklasse.

```
1 class Person(object):
2     def __init__(self,n,a):
3         self.name = n
4         self.alter = a
5         print("konstruktor für", n)
6
7     def pension(self):
8         return 67 - self.alter
9
10 class Mitarbeiter(Person):
11     def __init__(self,n,a,tel):
12         super().__init__(n,a) #c'tor der Basisklasse
13         self.telefon = tel
14
15 ich = Person('stefan', 37)
16 er = Mitarbeiter('egon', 38, 2097)
17 print("er wird pensioniert", er.pension() )
18 print(ich.pension() )
```

Ob eine Klasse von einer anderen erbt, kann man mit der Funktion **isinstance()** feststellen. So wäre im obigen Beispiel:

```
isinstance(ich,Person) ist True,
isinstance(ich,Mitarbeiter) ist False,
isinstance(er, Mitarbeiter) ist True,
isinstance(er, Person) ist True.
```

Interessant ist vor allem der letzte Fall: Das die Klasse **Mitarbeiter** von **Person** erbt, zählt die Instanz **er** also auch als **Person**. Die Klasse einer Instanz kann man auch mit der Funktion **type** bestimmen. Diese prüft dann aber nur exakte Übereinstimmungen:

```
type(ich) == Person ist True,
type(ich) == Mitarbeiter ist False,
type(er) == Mitarbeiter ist True,
type(er) == Person ist False.
```

6.4 Operatoren überladen

Klassen erlauben es, die üblichen vorhandenen Operatoren (z.B. `+` `-` `*` `%` `[]` `()`) zu überladen. (Es können keine neuen Operatoren definiert werden, und auch der Operatorvorrang und die Anzahl der Parameter ist fix.) Dies geschieht dadurch, dass in der Klasse Methoden mit speziellen reservierten Namen definiert werden.

Vollständige Liste: <https://docs.python.org/3/library/operator.html>

Beispiele:	<code>a + b</code>	<code>a.__add__(b)</code> oder <code>b.__radd__(a)</code>
	<code>a * b</code>	<code>a.__mul__(b)</code> oder <code>b.__rmul__(b)</code>

Zudem existieren von den meisten Operatoren eine rechtswirkende Variante. Falls `a` und `b` Instanzen von verschiedenen Klassen sind, so wird ja `a+b` als `a.__add__(b)` übersetzt. Falls das aber nicht definiert ist, so versucht Python dies als `b.__radd__(a)` zu übersetzen.

```
1 class Vektor(object):
2     def __init__(self,a,b,c):
3         self.data = [a,b,c]
4     def __mul__(self,other):
5         if isinstance(other, Vektor):
6             ergebnis = 0.0
7             for i in range(0,3):
8                 ergebnis = ergebnis + \
9                     self.data[i] * other.data[i]
10            return ergebnis
11        else:
12            return Vektor(other*self.data[0],
13                           other*self.data[1],
14                           other*self.data[2])
15    def __rmul__(self,other):
16        return self.__mul__(other)
17
18 if __name__ == '__main__':
19     a = Vektor(1,2,3)
20     b = Vektor(0.1, 0.01, 0.001)
21     x = 3*a           # a.__rmul__(3)
22     y = a*3           # a.__mul__(3)
23     s = a * b         # a.__mul__(b)
```

Zeile 18 sorgt dafür, dass der nachfolgende Code nur dann ausgeführt wird, wenn die Datei direkt mit Python ausgeführt wird (aber nicht wenn die Datei als Modul geladen wird). In den Zeilen 19 und 20 werden zwei Instanzen der Klasse `Vektor` angelegt. Zeile 21 kann nicht als `3.__add__(a)` übersetzt werden, weil für die Zahl 3 die Addition mit unseren selbstgeschriebenen Vektoren nicht definiert ist. Stattdessen wird Zeile 21 also als `a.__rmul__(3)` übersetzt. Zeile 22 hingegen kann als `a.__mul__(3)` übersetzt werden. Genau Zeile 23, die als `a.__mul__(b)` übersetzt wird.

7 Plots mit matplotlib/pylab

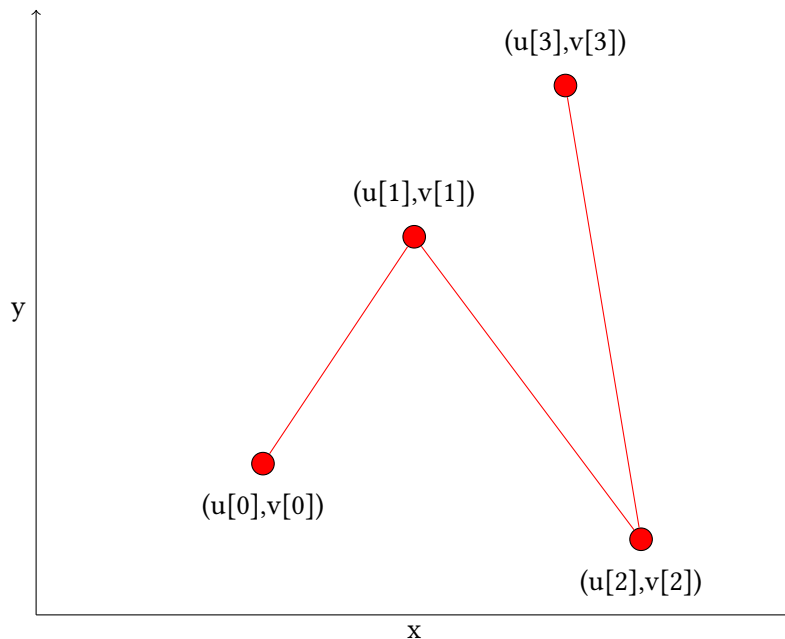
7.1 plot

Das Modul **pylab** bietet einen komfortablen Einstieg in Graphik mit Python. Es importiert seinerseits verschiedene Module, unter anderem **numpy** für Numerik und **matplotlib** für Graphik.

Eine zentrale Funktion ist **plot()**. So verbindet beispielsweise dieser Aufruf:

```
plot(u,v, 'r-')
```

nacheinander die Punkte $(u[0], v[0])$, $(u[1], v[1])$, $(u[2], v[2])$, $(u[3], v[3])$ mit einer roten Linie:



Plot nimmt also als ersten und zweiten Parameter etwas, das sich wie ein Array indizieren lässt. Das kann ein normales Python-Array sein, oder auch ein **ndarray**.

7.2 ndarray

Der Typ **ndarray** ist ein wichtiger Datentyp (aus **numpy**).

Oft braucht man gleichmäßig forlaufende Punkte für die unabhängige Variable (ähnlich dem Python-Range).

Dafür gibt es zwei praktische Funktionen: **linspace** und **arange**.

linspace nimmt als Parameter *Startwert, inklusiver Endwert, Anzahl der Unterteilungen*

arange nimmt als Parameter *Startwert, exklusiver Endwert, Schrittweite*.

linspace(0,2,5) ergibt **array([0. , 0.5, 1. , 1.5, 2.])**

arange(0,2.5,0.5) ergibt ebenfalls **array([0. , 0.5, 1. , 1.5, 2.])**.

Bei Instanzen von **ndarray** kann man einzelne Elemente mit den eckigen Klammern ansprechen (zum lesen und zum schreiben), ähnlich wie in normalen Python-Arrays:

```
1 x = arange(0,1,101)
2 x[7] = -7.0
3 print(x[0], x[7], x[100])
```

Zudem überlädt **numpy** (und damit auch **pylab**) die gängigen mathematischen Funktionen, so dass diese auf ganzen Arrays operieren können. Es wird dann jede Aktion Element für Element ausgeführt.

Hier ein Beispiel, wie man die Sinusfunktion zeichnet:

```
1 from pylab import *
2 x = linspace(0,2*pi,1000)
3 y = sin(x)
4 plot(x,y, 'r-')
5 show()
```

Und hier das Beispiel für eine Lissajous-Figur.⁷

```
1 from pylab import *
2 x = linspace(0,2*pi,1000)
3 y = sin(x)
4 plot(sin(7*x),cos(11*x), 'b-')
5 show()
```

7.3 Subplots

Die Funktion **subplot()** bewirkt, dass man in ein Fenster mehrere Plots zeichnen kann. Dabei nimmt **subplot** entweder drei Parameter (Zeilen, Spalten, Plotnummer) oder einen Parameter $N = 100z + 10s + n$, wobei z die Anzahl der Zeilen von Subplots, s die Anzahl von Spalten der Subplots und n die Plotnummer sind.

Hier werden zwei Plots übereinander gezeichnet (Zwei Zeilen, eine Spalte):

```
1 from pylab import *
2 x = linspace(0,10,1001)
3 subplot(211)
4 plot(x,sin(x), 'r-')
5 subplot(212)
6 plot(x,cos(x), 'b-')
7 show()
```

⁷ <https://de.wikipedia.org/wiki/Lissajous-Figur>

8 Dateien

8.1 Lesen und Schreiben von Textdateien

Um auf eine Datei zuzugreifen, verwendet man die Funktion `open()`. Diese gibt eine Instanz zurück, über die dann weitere Dateioperationen möglich sind. Der erste Parameter ist der Dateiname, der zweite der „Modus“. Als Modus ist `"r"` (read), `"w"` (write) oder `"a"` (append) möglich. Verwendet man `"w"`, so wird eine bereits bestehende Datei überschrieben! Die Modi `"rb"`, `"wb"` und `"ab"` bewirken ein binäres read, write bzw. append. Binär bedeutet, dass die Daten ohne Interpretation als Text, also ohne Umkodierung, direkt geschrieben werden.

```
1 # -*- coding: utf8 -*-
2 text = """ Irgend was mit Unicode: äußerst merkwürdig
3     noch eine Zeile
4     und noch eine
5     """
6
7 f = open("textdatei.txt", "w")
8 f.write(text)
9 f.close()
10
11 g = open("textdatei.txt")
12 neu = g.read()
13 g.close()
14 print(neu)
```

In diesem Beispiel wird ein String in eine Datei geschrieben und danach wieder gelesen. Die erste Zeile in diesem Skript erlaubt die Verwendung von Umlauten und anderen Sonderzeichen in Strings. (Python 2.x hat als Standardcodierung „latin1“. Verwendet Ihr Editor eine abweichende Codierung, muss man dies Python kundtun.)

Beim Einlesen von wissenschaftlichen Daten müssen im allgemeinen die gelesenen Daten – die hier ja als Zeichenketten (strings) gelesen werden – wieder in `float` oder `int` ingewandelt werden.

Eine einzelne Zeile erhält man mit der Methode `readline()`, während die Methode `readlines()` alle Zeilen der Datei als Iterable liefert.

Die einzelnen Worte einer Zeile erhält man z.B. mit der Methode `split()`:

```
1 zeile = f.readline()
2 worte = zeile.split()
```

Das nachfolgende Beispiel öffnet eine Datei zum Schreiben (die dann gegebenenfalls überschrieben wird, falls sie bereits existiert) und schreibt zehn Zeilen:

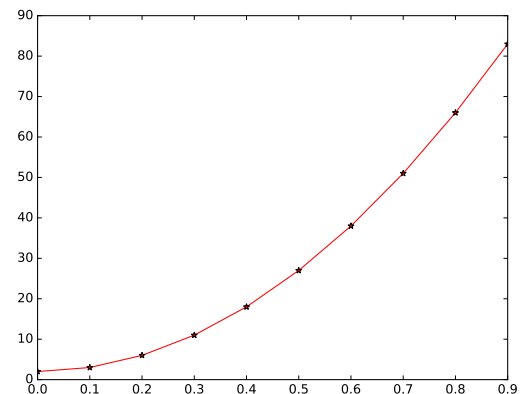
```
1 datei = open('meine', 'w')
2 for i in range(0,10):
3     print(i/10, 2+i**2, file=datei)
4 datei.close()
```

Die entstandene Datei sieht dann so aus:

```
1 0.0 2
2 0.1 3
3 0.2 6
4 0.3 11
5 0.4 18
6 0.5 27
7 0.6 38
8 0.7 51
9 0.8 66
10 0.9 83
```

Dann könnte man die Datei wieder einlesen und die erste Spalte als x-Werte und die zweite Spalte als y-Werte interpretieren, und diese dann Zeichnen.

```
1 datei = open('meine', 'r')
2 zeilen = datei.readlines()
3 datei.close()
4 u = [ ]
5 v = [ ]
6 for zeile in zeilen:
7     worte = zeile.split()
8     u.append( float(worte[0]) )
9     v.append( float(worte[1]) )
10 from pylab import *
11 plot(u, v, 'r*-')
12 show()
```



8.2 Binäres Lesen und Schreiben

Die Modi **"rb"**, **"wb"** und **"ab"** bei **open** bewirken ein binäres read, write bzw. append. Binär bedeutet, dass die Daten ohne Interpretation als Text, also ohne Umkodierung, direkt geschrieben werden.

Will man binäre, gleichartige Daten in Python-Objekte umwandeln, kann man z.B. das Modul **array** verwenden. Der Konstruktor der Klasse **array** im Modul **array** nimmt einen Parameter:

ein String der den Datentyp spezifiziert. (Z.B. **"d"** für Fließkommazahlen doppelter Genauigkeit oder **"B"** für vorzeichenlose Bytes.)


Die Methoden

- **frombytes**
- **tobytes**
- **fromlist**
- **tolist**

wandeln jeweils ein **array**-Objekt entweder in Binärdaten oder in ein normales Python-Array um.

[Datei ../Beispiele/07/binwrite.py als PDF-Attachment] 

```
1  from math import *
2  from array import *
3
4  f = open("pi.txt", "w")
5  print(pi, file=f)
6  f.flush()
7  f.close()
8
9  binaer = array('d') # "double": Zahlen mit
10 # Nachkommastellen doppelter Genauigkeit
11 binaer.fromlist( [pi] )
12
13 g = open("pibin.txt", "wb")
14 g.write(binaer.tobytes())
15 g.flush()
16 g.close()
```

[Datei ../Beispiele/07/binread.py als PDF-Attachment] 

```
1  f = open("pibin.txt", "rb")
2  data = f.read()
3  f.close()
4
5  from array import *
6
7  binaer = array('f')
8  binaer.frombytes(data)
9  for b in binaer:
10     print(b)
```

9 Ranges und List Comprehensions

Die *List Comprehensions* sind eine einfache Methode um normale Python-Arrays zu initialisieren. Der übliche code

```
1 a = [ ]
2 for i in iterable:
3     a.append(irgendwas_mit_i)
```

Kann ersetzt werden durch:

```
a = [irgendwas_mit_i for i in iterable]
```

Oder noch eleganter für Listen, bei denen die Elemente einer bestimmten Bedingung gehorchen müssen:

```
1 a = [ ]
2 for i in iterable:
3     if bedingung:
4         a.append(irgendwas_mit_i)
```

Kann ersetzt werden durch:

```
a = [irgendwas_mit_i for i in iterable if bedingung]
```

Um Ausschnitte aus Listen zu generieren, gibt es die *slices*. Dabei kann man einen Ausschnitt der Liste generieren durch:

```
ausschnitt = original[startwert:endwert]
```

Wobei nun **ausschnitt** die Elemente von **original[startwert]** bis **original[endwert-1]** enthält.

Für **endwert** sind auch negative Zahlen zulässig. Die Zählung beginnt dann im Array von hinten. So ist z.B. **a[7:-4]** gleichwertig zu **a[7:len(a)-4]**

10 Mehr zu Matplotlib

10.1 Hilfe zu diversen Funktionen oder Modulen

```
1 import pylab as pl
2 help(pl)      # Hilfe zu pylab selbst
3 help(pl.pie)  # Hilfe zu pie.
4 help(pl.bar)  # Hilfe zu bar.
5               #Beachte: ohne die runden
6               # Klammern nach bar oder pie.
7 help(pl.gca)  # Was ist gca überhaupt?
8 help(pl.gca())# Hilfe zum Achsenobjekt
```

10.2 Weitere plot-Funktionen

Weitere nützliche Plot-Funktionen sind z.B.:

- **bar** nimmt ähnliche Parameter wie **plot**, stellt die Funktion aber als Balkengraphik dar
- **pie** nimmt nur ein Array als Parameter, und produziert eine Tortengraphik. Ist die Summe der Arrayeinträge größer 1, so wird die Tortengraphik normiert, andernfalls nicht.

10.3 Achsenbeschriftung

Manche Features wie z.B. der Titel des Plots sind direkt aus **pylab** heraus aufzurufen. Andere (wie z.B. die Beschriftung der x-Achse) sind Funktionen es *Achsenobjects*. Dieses aktuelle Achsenobjekt bekommt man mit der Funktion **pylab.gca()** (steht für „get current axes“). Wieder andere Methoden gehören zur aktuellen *Figure*. Diese bekommt man mit der Funktion **pylab.gcf()** (steht für „get current figure“).

```
1 import pylab as pl
2 pl.title("Plot title")
3 pl.xlabel("x-Achsenbeschriftung")
4 pl.gca().set_xlabel("x-Achsenbeschriftung")
5 # Positionen auf der x-Achse, die beschriftet
6 # werden sollen
7 pl.gca().set_xticks( [1, 2.7, pl.pi, ] )
8 # Text, der an diesen Stellen erscheinen soll
9 pl.gca().set_xticklabels( ["eins", "e", "pi"] )
```

10.4 Histogramme

Meine Daten seien irgendwie verteilte reelle Zahlen. Dann stellt ein Histogramm dar, wie viele dieser Datenpunkte innerhalb eines bestimmten Intervalls liegen.

[Datei ../Beispiele/08/histo-bsp.py als PDF-Attachment] 

```
1 import pylab as pl
2 import random
3 random.seed(1001)
4 pl.title("Histogram")
5 data = [ ]
6 yy = [ ]
7 for i in range(0,30):
8     data.append(random.uniform(0,10))
9     yy.append(random.uniform(0.1,0.9))
10
11 pl.hist(data, range=(0,10), bins=10, color='yellow')
12 pl.plot(data,yy,'ro', markersize=10)
13 pl.show()
```

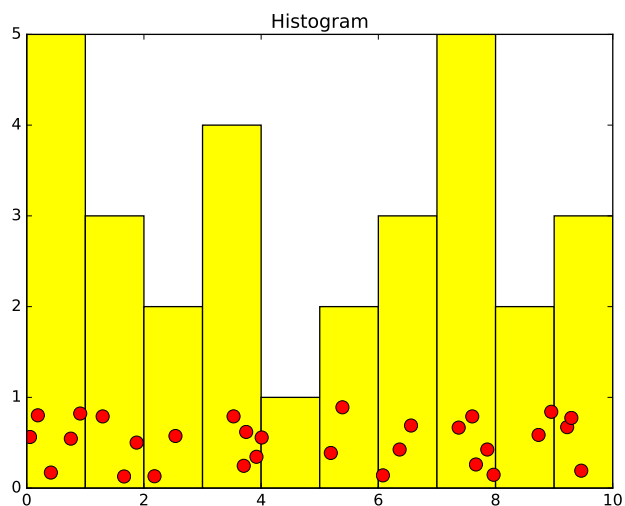


Abbildung 1: In diesem Histogramm entspricht die Höhe der gelben Balken der Anzahl der roten Punkte im entsprechenden Intervall auf der x-Achse. (Die y-Koordinate der roten Punkte ist unerheblich für das Histogramm und dient hier nur dazu, dass man die roten Punkte besser unterscheiden kann.)


10.5 Fits

10.6 Beliebige Funktionen

Die Funktion `scipy.optimize.curve_fit` optimiert beliebige Funktionen, ebenfalls nach dem minimalen Abstand in y-Richtung nach einem iterativen Algorithmus.⁸

Damit braucht die Funktion mindestens drei Parameter:

- Die zu fittende Funktion an sich,
- die unabhängige Variable,
- die abhängige Variable,
- den Parameter `p0`, der die Startwerte der Parameter der zu fittenden Funktion enthält. Da es sich um ein iteratives Verfahren handelt, hängt es tatsächlich von den Startwerten ab, ob das Verfahren gegen ein sinnvolles Ergebnis konvergiert, oder nicht.

[Datei ../Beispiele/08/linfit.py als PDF-Attachment] 

```
1 import pylab as pl
2 import scipy
3 import scipy.optimize as opt
4
5 V = [1, 2.2, 2.9, 4.1, 5.07]
6 A = [0.1, 0.2, 0.31, 0.3890, 0.55]
7
8 def strom(spannung,R,R0):
9     return R*spannung + R0
10
11 param,cov = opt.curve_fit(\
12     strom, V, A, p0=[1,1] )
13
14 print(param)
15
16 Aneu = [strom(v,param[0],param[1]) for v in V]
17
18 pl.plot(V,A,'r*')
19 pl.plot(V, Aneu, 'b-')
20 pl.show()
```

⁸ <https://de.wikipedia.org/wiki/Levenberg-Marquardt-Algorithmus>

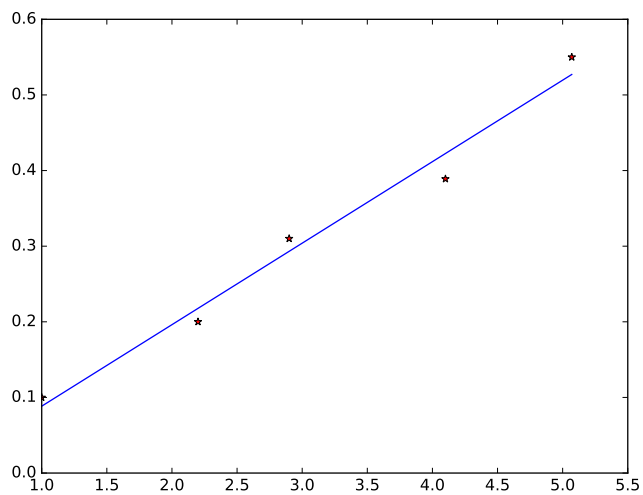



Abbildung 2: Hier sind die roten Sterne die ursprünglichen Daten und die blaue Linie stellt die gefittete Funktion dar.

Statt die neuen Daten über eine list comprehension zu berechnen, könnte man auch alle Daten als **ndarray** behandeln.

[Datei ../Beispiele/08/linfit2.py als PDF-Attachment] 


```

1  import pylab as pl
2  import scipy
3  import scipy.optimize as opt
4
5  V = pl.fromiter( [1, 2.2, 2.9, 4.1, 5.07], dtype='float' )
6  A = pl.fromiter( [0.1, 0.2, 0.31, 0.389, 0.55], dtype='float' )
7
8  def strom(spannung, R, R0):
9      return R*spannung + R0
10
11  param, cov = opt.curve_fit(\
12      strom, V, A, p0=[1,1] )
13
14  print(param)
15
16  pl.plot(V, A, 'r*')
17  pl.plot(V, strom(V, param[0], param[1]), 'b-')
18  pl.show()

```

10.7 Einschub: Lambdas

Manchmal ist es praktisch, Funktionen innerhalb einer Zeile zu konstruieren. Dann kann man mit der Anweisung **lambda** eine Funktion definieren, und das sogar innerhalb vom Ausdrücken. (**def** muss hingegen immer alleine stehen.)

[Datei ../Beispiele/09/lambda.py als PDF-Attachment] 

```
1 import pylab as pl
2 import scipy
3 import scipy.optimize as opt
4
5 V = [1, 2.2, 2.9, 4.1, 5.07]
6 A = [0.1, 0.2, 0.31, 0.3890, 0.55]
7
8 #def strom(spannung,R,R0):
9 #    return R*spannung + R0
10 #param,cov = opt.curve_fit(strom, V, A, p0=[1,1] )
11 strom = lambda s,R,R0: R*s+R0
12 param,cov = opt.curve_fit(strom, V,A, p0=[1,1] )
13
14 print(param)
15
16 Aneu = [strom(v,param[0],param[1]) for v in V]
17
18 pl.plot(V,A,'r*')
19 pl.plot(V, Aneu, 'b-')
20 pl.show()
```


10.8 Komplizierte Fits

Funktionen zum Fitten von Daten finden sich in den Modulen **scipy.optimize** oder **scipy.odr**.

Nützliche Funktionen sind unter Anderem:

- **scipy.optimize.leastsq** arbeitet nach der Methode der kleinsten Quadrate und verwendet den Levenberg-Marquardt-Algorithmus.⁹ Die Funktion erwartet, dass das Callable (erster Parameter) ein Array zurückgibt. Der Rückgabewert des Callables wird immer auf Null hin optimiert. Vorteil ist, dass man mit dieser Funktion auch komplizierte Fits an mehrere Datensätze gleichzeitig machen kann. Der Nachteil ist, dass man die Differenz von Modelldaten zu Messdaten selbst bilden muss. Beachte auch, dass diese Funktion nicht garantiert, dass eine gute Lösung gefunden wird. Es hängt viel von der Wahl der Startparameter (**x0**) ab, ob die beste Lösung gefunden wird.
- **scipy.optimize.curve_fit** Dies ist die einfachste Funktion, wenn man übliche Daten einer Veränderlichen fitten will. Sie ist eigentlich ein Wrapper für **scipy.optimize.leastsq**, jedoch übergibt man die Modellfunktion direkt und muss den Abstand zu den Messdaten nicht selbst bilden.
- **scipy.optimize.fmin** Arbeitet mit einem anderen Algorithmus, nämlich mit einem Simplex-Verfahren.¹⁰ Hier wird verlangt, dass das Callable nur eine Zahl zurückgibt. Diese wird minimiert.
- **scipy.odr** Bietet eine Variante des Levenberg-Marquardt-Algorithmus', jedoch kann ODR mit Daten umgehen, die auch Fehler in der unabhängigen Variablen vorweisen.

10.9 Ausführliches Beispiel

[Datei ../Beispiele/08/fitexample.py als PDF-Attachment] 

```
1 import matplotlib
2 import sys
3 if len(sys.argv) == 2:
4     matplotlib.use('pdf')
5
6 import scipy as sc
7 import scipy.optimize as opt
8 import scipy.odr as odr
9 import numpy as np
10 import pylab as pl
11 import random
12 import sys
13
14 random.seed(1)
15
16 #-----
17 # generiere die „Messdaten: Gaußkurve mit zufälligem
18 # Rauschen darauf.
19 #
```

⁹ <https://de.wikipedia.org/wiki/Levenberg-Marquardt-Algorithmus>

¹⁰ <https://de.wikipedia.org/wiki/Downhill-Simplex-Verfahren>

```

20 x = np.linspace(-2,2,41)
21 y = np.array( [0.7*np.exp(-(0.9*x[i]+0.2)**2)\
22               + random.uniform(-0.1,0.1) \
23               for i in range(0,len(x)) ] )
24 s = np.array( [random.uniform(0,0.2)\
25               for i in range(0,len(x)) ] )
26 z = np.array( [random.uniform(0,0.5)\
27               for i in range(0,len(x)) ] )
28
29 print(len(x), x)
30 print(len(y), y)
31
32 #-----
33 # zeichne die „Messdaten inklusive Fehlerbalken
34 pl.errorbar(x,y,yerr=s,xerr=z,fmt='r*-')
35
36 #-----
37 # Definiere die Modellfunktion als Gaußfunktion mit drei Parametern:
38 #   - Amplitude A
39 #   - Breite w
40 #   - Verschiebung phi
41 #
42 def fitfunc(x, A, phi, w):
43     return A*np.exp(-(w*x+phi)**2)
44 #-----
45
46 print("-----")
47 # curve_fit fittet in callabe mit der unabhängigen Variablen x
48 # and die Daten y
49 param,cov = opt.curve_fit(fitfunc, x, y,
50                           p0=[1,1,1], sigma=s)
51 print("fitted parameters", param)
52 print("covariance", cov)
53 print("-----")
54
55 pl.plot(x,fitfunc(x,param[0],param[1],param[2]), 'b-',
56         linewidth=3)
57
58 print("-----")
59 # Die funktion leastsq optimiert immer auf auf Null,
60 # wobei das Callable errfunc ein Array zurückgeben muss.
61 errfunc = lambda pr,x,y,s: (y-fitfunc(x,*pr))/s
62 optparam,covar,infodict,mesg,ier =\
63     opt.leastsq(errfunc, [1,1,1], args=(x,y,s),\
64                 full_output=True)
65 print()
66 print('optparam', optparam)
67 print('mesg', mesg)
68 print("-----")
69 pl.plot(x,fitfunc(x,*optparam),'y-', linewidth=2)
70
71 print("-----")
72 # Die Funktion fmin optimiert global auf Null, sprich
73 # das Callable errfunc muss hier nur genau einen Wert
74 # zurückgeben.
75 errfunc = lambda pr,x,y,s: sum( abs((y-fitfunc(x,*pr))/s) )
76 optparam,covar,infodict,mesg,ier =\
77     opt.fmin(errfunc, [1,1,1], args=(x,y,s),\
78              full_output=True)
79 print()
80 print('optparam', optparam)
81 print('mesg', mesg)

```

```

82 print("-----")
83 pl.plot(x, fitfunc(x,*optparam), 'b-', linewidth=2)
84
85
86 print("-----")
87 # ODR funktioniert ähnlich wie curve_fit, jedoch
88 # werden hier auch Daten mit Fehlern in der abhängigen Variablen
89 # (die die Variable x) möglich.
90 #
91 model = odr.Model(lambda Aphiw,x: fitfunc(x,*Aphiw)) #expand tuple
92 data = odr.RealData(x,y,sx=z, sy=s) # mit x-fehler
93 #data = odr.RealData(x,y,sy=s,sx=0.0001) # (fast) ohne x-fehler
94 #data = odr.RealData(x,y) ganz ohne Fehler
95 modr = odr.ODR(data,model,beta0=[1,1,1])
96 outr = modr.run()
97 print("beta", outr.beta)
98 print("fehler in beta", outr.sd_beta)
99 #
100 # Anmerkung: der "Fehler in Beta" (also hier outr.sd_beta)
101 # sind die Quadratwurzeln aus den Diagonaleinträgen
102 # der Kovarianzmatrix: sd_beta[i] = sqrt(cov_veta[i,i])
103 #
104 # Wie findet man das heraus?
105 # 1) Quellcode von scipy ansehen,
106 # 2) ODR Manual lesen:
107 #   http://docs.scipy.org/doc/external/odrpac_guide.pdf
108 #
109 print("covariance")
110 print(outr.cov_beta)
111 print("-----")
112
113 pl.plot(x, fitfunc(x,*outr.beta), 'c-')
114
115 if len(sys.argv) == 2:
116     pl.savefig(sys.argv[1]+".pdf")
117     pl.show()
118     sys.exit(0)
119 else:
120     pl.show()

```

(Dieses Beispiel ist auch als Datei fitexample.py auf GRIPS.)

11 3D-Graphik mit Matplotlib und Axes3d

Das Paket **matplotlib** (**pylab** ist ein Teil davon) besitzt Möglichkeiten, einfachere 3D-Graphiken zu erzeugen. Der nachfolgende Codeblock kann das Grundgerüst darstellen:

```
1 import pylab as pl
2 from mpl_toolkits.mplot3d import Axes3D # 3D plots
3 import matplotlib.tri as mtri          # Triangulation
4 import mpl_toolkits.mplot3d as a3      # 3D shortcut
5 # ...
6 fig = pl.figure()
7 ax = fig.add_subplot(1, 1, 1, projection='3d')
8 # ...
9 pl.show()
```

Dabei wird jedoch das 3D-rendering in Software vorgenommen, sprich, die 3D-Darstellung wird berechnet und dann in einer zweidimensionalen Zeichenebene dargestellt. Im Allgemeinen wird auf eine 3D-Beschleunigung oder fortgeschrittene Algorithmen zur Darstellung von Flächen (OpenGL oder ähnliches) verzichtet. Dadurch kann es in Einzelfällen zu Artefakten kommen, sprich, es kann sein dass Linien, die eigentlich durch andere Flächen verdeckt sein sollten, trotzdem angezeigt werden.

11.1 Flächen

11.1.1 Funktionsplots

Die Funktion **ax.plot_surface(X,Y,Z)** dient dazu, Funktionen zweier Variablen (also Funktionen der Form $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$) darzustellen. Dazu müssen die Parameter **X**, **Y** und **Z** eine zweidimensionale Arraystruktur aufweisen, am besten sind diese dann vom Typ **ndarray**.

Dabei wird dann jedem Punkt (**X[i,j]**, **Y[i,j]**) in der xy-Ebene ein Funktionswert **Z[i,j]** zugeordnet, und alle diese Werte werden dann durch eine geeignete Fläche verbunden. (Die Variablen **i** und **j** laufen dann durch alle möglichen Werte. Somit müssen also **X**, **Y** und **Z** dieselbe Dimension haben.

Damit man übliche Funktionen auf regulären Gittern schon zeichnen kann, gibt es Funktionen, die geeignete **X**, **Y** erstellen lassen:

```
1  pl.mgrid[0:2:3j, -1:1:3j]
2  #array([[ 0.,  0.,  0.],
3  #       [ 1.,  1.,  1.],
4  #       [ 2.,  2.,  2.]])
5  #
6  #       [[-1.,  0.,  1.],
7  #       [-1.,  0.,  1.],
8  #       [-1.,  0.,  1.]])
```

```
1  x = pl.linspace(0,2,3)
2  y = pl.linspace(-1,1,3)
3  pl.meshgrid(x,y)
4  #[array([[ 0.,  1.,  2.],
5  #       [ 0.,  1.,  2.],
6  #       [ 0.,  1.,  2.]])
7  #
8  # array([[ -1., -1., -1.],
9  #       [ 0.,  0.,  0.],
10 #       [ 1.,  1.,  1.]])]
```

Das Ergebnis sind immer **ndarrays**, in denen bei einem die Einträge über Spalten hinweg konstant sind, und beim anderen die Einträge über Zeilen hinweg konstant sind. Damit erhält man dann alle möglichen Kombinationen in einem regulären Gitter.

Die üblichen mathematischen Funktionen von **numpy** (und damit auch von **pylab**) sind wieder für diese Art von Arrays überladen, so dass man mit diesen Arrays ganz normal rechnen kann.

So kann man mit:

```
ax.plot_surface(X,Y,pl.cos(pl.sqrt(X*X + Y*Y)))
```

Die Funktion $\cos(x^2 + y^2)$ in der Ebene darstellen.

11.1.2 Polygone

Für beliebige Freiflächen kann man auch eine Anzahl an Polygonen mit individuell festgelegten Flächen- und Kantenfarben zeichnen:

```
1 ax.add_collection3d(a3.art3d.Poly3DCollection(\
2     vertices,
3     facecolors=fcol,
4     edgecolors=ecol))
```

Dabei ist **vertices** eine Liste der Polygone, wobei ein Polygon eine Liste der Ecken ist, wobei eine Ecke ein Triplet **[x,y,z]** der Koordinaten ist.

Die Parameter **fcol** und **ecol** sind die Flächen- und Kantenfarben. Dies sind Arrays mit einem Eintrag pro Polygon. Die Farbe selbst kann dabei entweder ein Python-String sein (z.B. **"red"**) oder ein Triplet **(r,g,b)** mit den Rot-, Grün- und Blauwerten der Farbe, oder ein Quadruplet **(r,g,b,a)** mit den Rot-, Grün, Blau- und Alphawerten der Farbe. Alphawert bedeutet dabei die Transparenz. Alpha=0 bedeutet völlig Transparent, Alpha=1 bedeutet völlig Opak.

Nachfolgendes Beispiel zeichnet eine Pyramide. (Code auch auf GRIPS ohne Zeilennummern verfügbar.)

```
1 import pylab as pl
2 import numpy as np
3 from mpl_toolkits.mplot3d import Axes3D
4 import matplotlib.tri as mtri
5 import random
6 import mpl_toolkits.mplot3d as a3
7
8
9 vertices = [
10     [ [0,0,0], [1,0,0], [0,1,0] ],
11     [ [0,0,0], [1,0,0], [0,0,1] ],
12     [ [0,0,0], [0,0,1], [0,1,0] ],
13     [ [0,0,1], [0,1,0], [1,0,0] ] ]
14
15 fcol = [ 'blue', 'red', 'green', 'yellow' ]
16 ecol = [ 'black', 'white', 'black', 'white' ]
17
18 fig = pl.figure()
19 ax = fig.add_subplot(1, 1, 1, projection='3d')
20
21 ax.set_aspect('equal')
22 ax.add_collection3d(a3.art3d.Poly3DCollection(\
23     vertices,
24     facecolors=fcol,
25     edgecolors=ecol))
26
27 pl.show()
```

11.1.3 Triangulationen

Hier zuerst ein Beispiel, dass einen Zylinder zeichnet:

```
1 import numpy as np
2 from mpl_toolkits.mplot3d import Axes3D
3 import matplotlib.tri as mtri
4
5 u_ = np.linspace(0,2.0*np.pi,20) # u-Koordinate
6 v_ = np.linspace(0,4,20)         # v-Koordinate
7
8 U,V = np.meshgrid(u_, v_)        # 2D-Grid erstellen
9 u = U.flatten()                  # in 1d-Arrays
10 v = V.flatten()                  # "ausrollen"
11
12 x = np.cos(u)                   # Berechnung der
13 y = np.sin(u)                   # Fläche im Raum.
14 z = v                           # Hier: ein Zylinder
15
16 tr = mtri.Triangulation(u, v)    # Triangulation
17 # auf den ursprünglichen(!) Koordinaten berechnen lassen
18
19 fig = plt.figure()              # zeichnen...
20 ax = fig.add_subplot(111, projection='3d')
21 ax.plot_trisurf(x,y,z, triangles=tr.triangles )
```

Dabei sind **u_** und **v_** die Koordinaten auf dem Zylindermantel. **u_** stellt als den Winkel entlang des Zylinderumfanges dar, und **v_** die Höhe dar.

Auf dem Zylindermantel wird dann mit **mtri.Triangulation(...)** eine Zerlegung in Dreiecke durchgeführt. Dazu müssen aber die 2D-Arrays **U** und **V** in 1D-Arrays umgewandelt werden, da **mtri.Triangulation(...)** nur eindimensionale Arrays als Input akzeptiert. (Mehr dazu gleich, das die Triangulation eigentlich macht.)

Die Funktion **plot_trisurf** dann dann mit dem Output von **mtri.Triangulation(...)** umgehen, und diesen automatisch in Polygone umrechnen. Das geschieht nach diesem Schema:

Die 1D-Inputarrays sind:

$$\begin{aligned}x &= (x_1, x_2, x_3, \dots) \\y &= (y_1, y_2, y_3, \dots) \\z &= (z_1, z_2, z_3, \dots)\end{aligned}$$

Die Triangulation wird dargestellt durch ein Array von Tripeln, wobei jeder Eintrag eine Ganzzahl ist: $\text{triangles} = ((t_{a1}, t_{b1}, t_{c1}), (t_{a2}, t_{b2}, t_{c2}), (t_{a3}, t_{b3}, t_{c3}), \dots)$

Diese Ganzzahlen dienen dann als Index in die 1D-Inputarrays, und so werden die Polygone dann angelegt:

Polygon 1: $\left((x_{t_{a1}}, y_{t_{a1}}, z_{t_{a1}}), (x_{t_{b1}}, y_{t_{b1}}, z_{t_{b1}}), (x_{t_{c1}}, y_{t_{c1}}, z_{t_{c1}}) \right),$

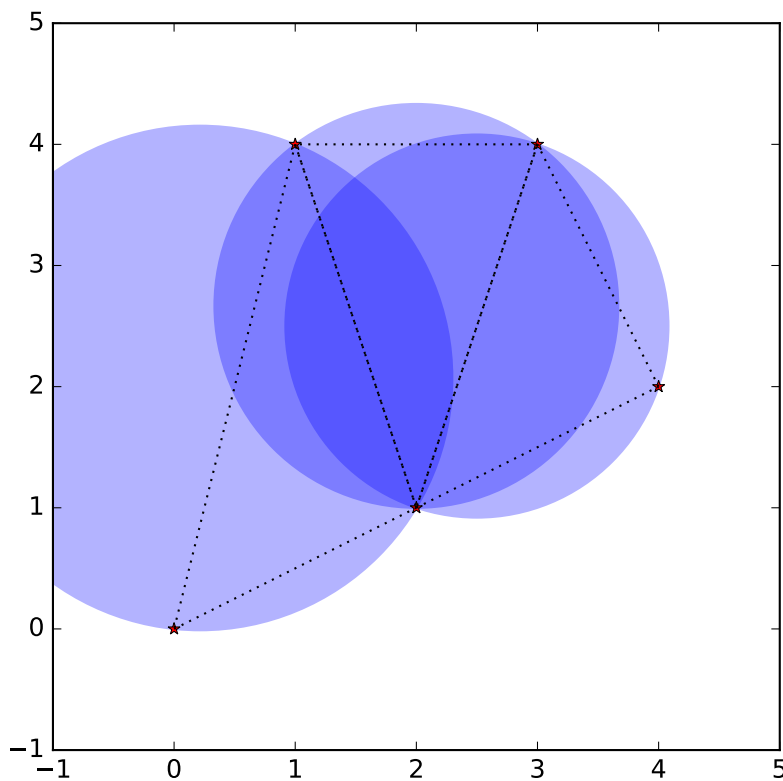
Polygon 2: $\left((x_{t_{a2}}, y_{t_{a2}}, z_{t_{a2}}), (x_{t_{b2}}, y_{t_{b2}}, z_{t_{b2}}), (x_{t_{c2}}, y_{t_{c2}}, z_{t_{c2}}) \right),$

Dabei darf der Umkreis eines Dreiecks keinen weiteren Punkt enthalten. Eine solche Triangulierung ist immer möglich, aber nicht notwendigerweise eindeutig.

Siehe auch:

<https://de.wikipedia.org/wiki/Delaunay-Triangulation>

Der nachfolgende Plot zeigt die Triangulierung für fünf Punkte, so dass sich drei Dreiecke ergeben. (Beispielcode für diesen Plot in der Datei `triang-demo.py`)



12 3D-Graphik mit Mayavi

12.1 Plots mit Mayavi

Eine recht gut strukturierte Anleitung zu **mayavi** gibt es hier:

http://code.enthought.com/projects/mayavi/docs/development/latex/mayavi/mayavi_user_guide.pdf

Im Kern ist wichtig: ähnlich wie bei 3D-Graphik mit matplotlib verlangen auch die meisten Funktionen in mayavi ndarrays oder die 2-dimensionalen Arrays, die man mit **meshgrid** oder **mgrid** erzeugt, wie in Abschnitt 11.1.1 „Funktionsplots“ beschrieben.

Nützliche Funktionen sind:

- **plot3d(X,Y,Z,T,...)**
Hier sind **X,Y,Z,T** eindimensionale Arrays und es wird eine Kurve im Raum gezeichnet. Der vierte Parameter (**T**) wird als Input für die Farbgebung benutzt.
- **points3d(X,Y,Z, T,)**
Hier sind **X,Y,Z,T** eindimensionale Arrays. Es werden einzelne Punkte im Raum dargestellt. Der vierte Parameter dient zur Farbgebung oder steuert die Größe der Punkte.
- **surf(X,Y,Z,colormap='flag')**
Hier sind **X,Y,Z** zweidimensionale Arrays. Es wird eine Funktionsfläche im Raum gezeichnet, so dass also $f(X,Y) \mapsto Z$ dargestellt wird.

*Leider ist **mayavi** bislang nur für python2 verfügbar!* Die Installation per Hand ist ebenfalls sehr kompliziert, da das Paket zahlreiche nichttriviale Abhängigkeiten hat, insbesondere zur VTK-Bibliothek¹¹. Der einfachste Weg ist über Anaconda&Python2, oder über Macports (für OSX).

12.2 Animationen mit Mayavi

Animationen in Mayavi erreicht man, indem man eine Funktion definiert, die

- mit dem Dekorator **mayavi.mlab.animate** dekoriert ist (Der Parameter **delay=ms** des Dekorators bestimmt die Zeit in Millisekunden zwischen den Frames).
- und die nach jedem neuen Frame mittels **yield** die Kontrolle zurück an das aufrufende Programm gibt.

¹¹ <http://www.vtk.org/>

```

1  import mayavi.mlab as ml
2  import numpy as n
3
4  t = n.linspace(0,2*n.pi,200)
5  x = n.sin(t)
6  y = n.cos(t)
7  z = n.ones_like(t)
8
9  src = ml.pipeline.line_source(x,y,z,t)
10 src2 = ml.pipeline.stripper(src)
11 src3 = ml.pipeline.tube(src2,tube_radius=0.2,
12     tube_sides=50)
13 src4 = ml.pipeline.surface(src3,colormap='flag')
14
15 @ml.animate(delay=30)
16 def animiere():
17     az = 0
18     el = 0
19     es = 1
20     ax = 1
21     while True:
22         ml.view(azimuth=az, elevation=el, distance=10)
23         az += ax
24         el += es
25         if az > 350: az = -1
26         if ax < 0: ax = 1
27         if el > 90: es = -1
28         if el <= 0: es = 1
29         yield
30
31 ani = animiere()
32 ml.show()

```

In obigem Beispiel wird also in Zeile 15 die Funktion dekoriert. Innerhalb dieser Funktion wird nach jedem neuen Frame in Zeile 29 die Kontrolle zurück an das aufrufende Programm gegeben, und schließlich wird in Zeile 31 die Funktion einmal aufgerufen.

Die Funktion ändert hierbei nicht die Daten, sondern variiert lediglich den Sichtpunkt.

```

1  import mayavi.mlab as ml
2  import numpy as n
3
4  t = n.linspace(0,2*n.pi,200)
5  x = n.sin(t)
6  y = n.cos(t)
7  z = n.ones_like(t)
8
9  src = ml.pipeline.line_source(x,y,z,t)
10 src2 = ml.pipeline.stripper(src)
11 src3 = ml.pipeline.tube(src2,tube_radius=0.2,
12     tube_sides=50)
13 src4 = ml.pipeline.surface(src3,colormap='flag')
14
15 # Animation definieren.
16 # Der Dekorator "ml.animate" ist bereits definiert.
17 @ml.animate(delay=1000)
18 def animiere(quelle):
19     aenderung = 0.1
20     grenze = 0.1
21     while True:
22         t = n.linspace(0, grenze,200)
23         x = n.sin(t)
24         y = n.cos(t)
25         z = n.ones_like(t)
26         grenze = grenze + aenderung
27         if grenze > 2*n.pi:
28             grenze = 0.1
29         quelle.set(x=x,y=y,z=z,t=t)
30         yield
31
32 # die Animation ausfuehren
33 ani = animiere(src.mlab_source)
34 ml.show()

```

In diesem Beispiel werden die Daten animiert. Dazu wird zuerst in Zeile 9 ein Datenobjekt geschaffen. Die Funktion, die die Animation ausführt wird wieder dekoriert (Zeile 17) und gibt nach jedem Animationsschritt die Kontrolle mittels yield (Zeile 30) zurück an das ausführende Programm. Um die Daten zu animieren, werden bei dem bereits existierenden Datenobjekt andere Daten eingefügt (Zeile 29). Zuletzt wird die Animationsfunktion einmal aufgerufen (Zeile 33).

13 Einschub: Dekoratoren

Die sogenannten *Dekoratoren* bieten eine hübsche Syntax um Funktionen zu „verziern“, sprich, eine Funktionsdefinition zu erweitern. Die Schreibweise ist:

```
1 @meindeko
2 def wasanderes():
3     # normale Funktionsdefinition
```

Dies stellt aber lediglich eine abkürzende Schreibweise dafür da:

```
1 def irgendwas():
2     #normale Funktionsdefinition
3
4 irgendwas = meindeko(irgendwas)
```

Hieran sieht man, dass ein Dekorator eine gewöhnliche Funktion sein kann, welche eine Funktion als Parameter nimmt und wieder eine Funktion zurückgibt.

```
1 # Function decorator, no arguments
2 def fundecNoArgs(fun):
3     def closure(*args, **kwargs):
4         print("START fundec, no args")
5         fun(*args,**kwargs)
6         print("END fundec, no args")
7     return closure
8
9 @fundecNoArgs
10 def plain(a,b,c):
11     print("plain",a,b,c)
12
13 plain(1,2,3)
```

```
1 # Function decorator, no arguments
2 def fundecNoArgs(fun):
3     def closure(*args, **kwargs):
4         print("START fundec, no args")
5         fun(*args,**kwargs)
6         print("END fundec, no args")
7     return closure
8
9 def plain(a,b,c):
10     print("plain",a,b,c)
11
12 plain=fundecNoArgs(plain)
13
14 plain(1,2,3)
```

```
1 # Class decorator, no arguments
2 class classdecNoArgs(object):
3     def __init__(self,fun):
4         self.fun = fun
5     def __call__(self,*args,**kwargs):
6         print("START classdec, no args")
7         self.fun(*args,**kwargs)
8         print("END classdec, no args")
9
10 @classdecNoArgs
11 def plain(a,b,c):
12     print("plain",a,b,c)
13
14 plain(1,2,3)
```

```
1 # Class decorator, no arguments
2 class classdecNoArgs(object):
3     def __init__(self,fun):
4         self.fun = fun
5     def __call__(self,*args,**kwargs):
6         print("START classdec, no args")
7         self.fun(*args,**kwargs)
8         print("END classdec, no args")
9
10 def plain(a,b,c):
11     print("plain",a,b,c)
12
13 plain=classdecNoArgs(plain)
14
15 plain(1,2,3)
```

Etwas komplizierter ist es bei Dekoratoren mit Argumenten. Dann ist das Dekorieren:

```
1 @meindeko(a,b,c)
2 def irgendwas():
3     #code
```

äquivalent dazu:

```
1 irgendwas=meindeko(a,b,c)(irgendwas)
```

Dekoratoren mit und ohne Parameter lassen sich entweder als Funktionen oder als Klassen (mit geeigneter Methode `__call__`) realisieren.

```
1 # Function decorator, with arguments
2 def fundecWithArgs(x,u):
3     def intermediate(fun):
4         def anon(*args,**kwargs):
5             print("START fundec, with args",
6                   x,u)
7             fun(*args,**kwargs)
8             print("END fundec, with args",
9                   x,u)
10            return anon
11        return intermediate
12
13 @fundecWithArgs('foo','bar')
14 def plain(a,b,c):
15     print("plain",a,b,c)
16
17 plain(1,2,3)
```

```
1 # Function decorator, with arguments
2 def fundecWithArgs(x,u):
3     def intermediate(fun):
4         def anon(*args,**kwargs):
5             print("START fundec with args",
6                   x,u)
7             fun(*args,**kwargs)
8             print("END fundec with args",
9                   x,u)
10            return anon
11        return intermediate
12
13 def plain(a,b,c):
14     print("plain",a,b,c)
15
16 plain=fundecWithArgs('foo','bar')(plain)
17
18 plain(1,2,3)
```

```
1 # Class decorator, with arguments
2 class classdecWithArgs(object):
3     def __init__(self,x,u):
4         self.x = x
5         self.u = u
6     def __call__(self,fun):
7         def anon(*args,**kwargs):
8             print("START classdec with args",
9                   self.x,self.u)
10            fun(*args,**kwargs)
11            print("END
12            classdec with args",
13                  self.x,self.u)
14            return anon
15
16 @classdecWithArgs('foo','bar')
17 def plain(a,b,c):
18     print("plain",a,b,c)
19
20 plain(1,2,3)
```

```
1 # Class decorator, with arguments
2 class classdecWithArgs(object):
3     def __init__(self,x,u):
4         self.x = x
5         self.u = u
6     def __call__(self,fun):
7         def anon(*args,**kwargs):
8             print("START classdec with args",
9                   self.x,self.u)
10            fun(*args,**kwargs)
11            print("END
12            classdec with args",
13                  self.x,self.u)
14            return anon
15
16 def plain(a,b,c):
17     print("plain",a,b,c)
18
19 plain=classdecWithArgs('foo','bar')(plain)
20
21 plain(1,2,3)
```


14 Einschub: yield

Das Schlüsselwort **yield** innerhalb von Funktionen wirkt ähnlich wie **return** in dem Sinne, als dass die Funktion einen Wert zurückgibt.

Es ist quasi so, dass ich **yield** „merkt“, wo in der Funktion das letzte mal **yield** stand, und macht gewissermaßen das nächste mal beim vorangehenden **yield** weiter.

Genauer: eine Funktion mit **yield** gibt keinen normalen Funktionswert zurück, sondern ein Objekt welches dem *Iteratorprotokoll* folgt.

```
1  def itfunc(von,bis):
2      for x in range(von,bis):
3          yield x
4
5  for i in itfunc(1,7):
6      print(i)
7
8  #oder auch:
9  iter = itfunc(1,7)
10 for i in iter:
11     print(i)
12
13 #oder auch:
14 iter = itfunc(1,7)
15 print(iter.__next__())
16 print(iter.__next__())
17 print(iter.__next__())
18 print(iter.__next__())
19 print(iter.__next__())
20 print(iter.__next__())
21 print(iter.__next__())
```