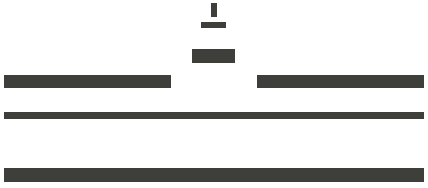


Westfälische Wilhelms-Universität Münster

Angewandte Mathematik Münster

Pythonkurs Sommersemester 2014

Einführung in die Programmierung zur Numerik mit Python

wissen.leben
WWU Münster

F. Meyer, M. Schaefer

19. September 2014

Organisation

- Anwesenheitsliste
- Leistungspunkte: Unbenoteter Schein für 2 ECTS Punkte bei Teilnahme an allen 5 Tagen
- Vorkenntnisse: Umgang mit Linux oder Programmiererfahrung?
- Funktioniert der Login?
- Dank an René

Literatur

- https://en.wikibooks.org/wiki/Non-Programmer's_Tutorial_for_Python
- http://docs.python.org/2/reference/
- A Primer on Scientific Programming with Python, Hans Peter Langtangen, Springer 2011
- http://docs.scipy.org/doc/numpy/reference/index.html
- http://wiki.scipy.org/Tentative_NumPy_Tutorial
- http://community.linuxmint.com/tutorial/view/244

Übersicht

Tag 1: Linux-Konsole und Python Grundlagen

Tag 2: Funktionen, Klassen und Vererbung

Tag 3: Numerik mit Python - das Modul NumPy

Tag 4: Einführung in die Numerik, die Matplotlib und Debuggen in Python

Tag 5: Generatoren, Lambdas, Comprehensions und die Python Standardbibliothek

Warum Programmierung?

Beispiel: Matrixinverse berechnen

Das Invertieren einer Matrix mit Millionen Elementen ist per Hand zu aufwändig. Man bringe also dem Computer bei: für $A \in GL_n(\mathbb{R})$ finde A^{-1} sodass $AA^{-1} = I$

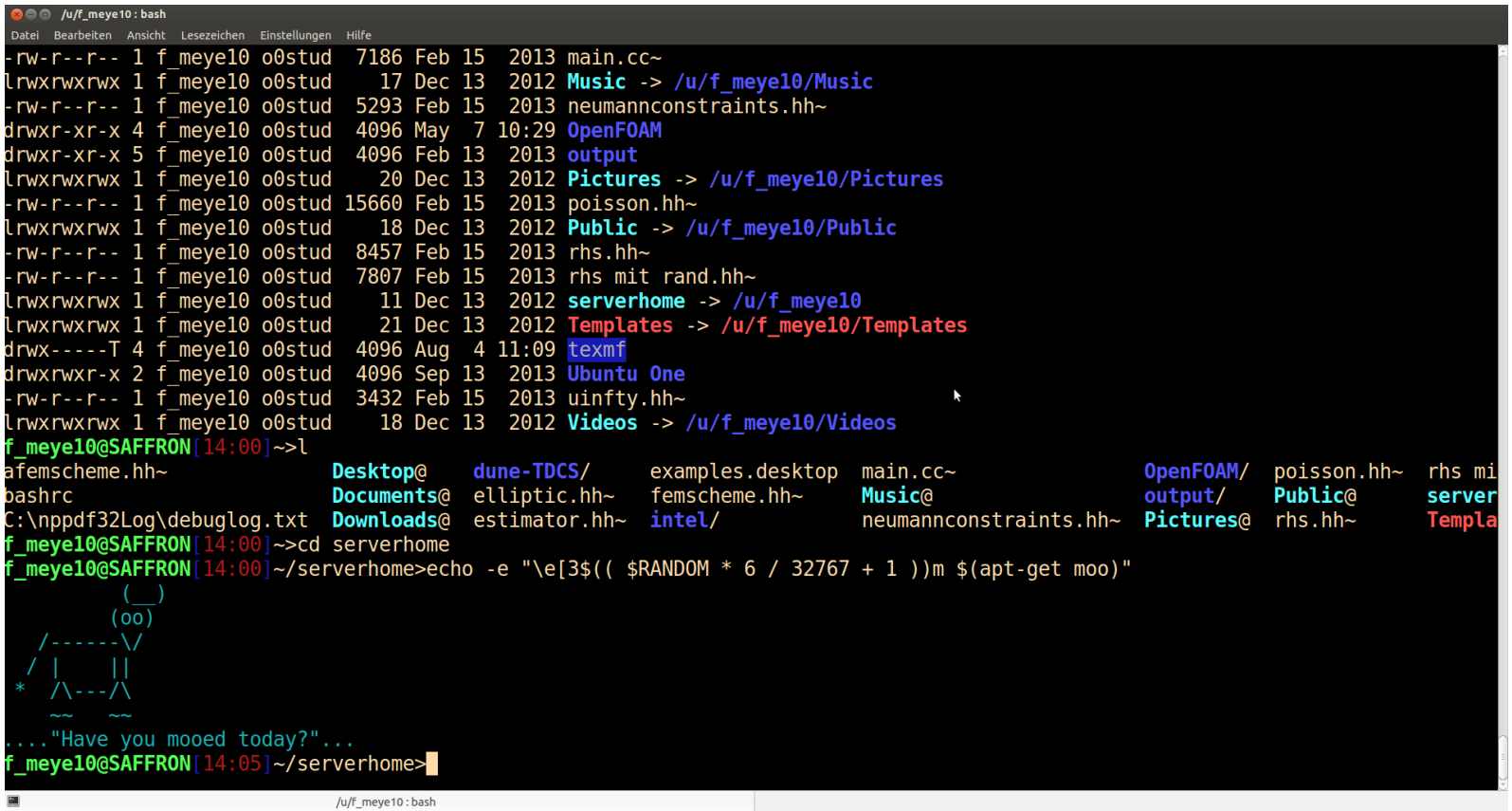
- Eingabe: Matrix $A \in \mathbb{R}^{mxn}$
  Keinerlei Forderung an m, n. A vielleicht gar nicht invertierbar. Welche (Daten-)Struktur hat A?
- Überprüfung der Eingabe: erfüllt A notwendige Bedingungen an Invertierbarkeit? Ist die Datenstruktur wie erwartet?
- A^{-1} berechnen, etwa mit Gauss-Algorithmus.
- Ausgabe: Matrix A^{-1}, falls A invertierbar, Fehlermeldung sonst.
- Probe: $AA^{-1} = I$? Was ist mit numerischen Fehlern?

Warum Programmierung?

- Um mathematische Problemstellungen, insbesondere aus der linearen Algebra und Analysis, zu lösen, ist die Anwendung von Rechnersystemen unerlässlich. Dies ist einerseits durch eine in der Praxis enorm hohe Anzahl von Variablen und entsprechender Dimensionen als auch durch die Nichtexistenz einer analytischen(exakten) Lösung bedingt.
- Die Numerische Lineare Algebra beschäftigt sich u.a. mit der Theorie und Anwendung von Algorithmen zur Lösung großer linearer Gleichungssystem und folglich der günstigen Berechnung sowie Darstellung entsprechend großer Matrizen. Für die Umsetzung dieser Algorithmen verwenden wir Programmiersprachen wie Python, C++ und Java oder Programme wie Matlab.

Umgang mit der Linux Konsole

Eine typische Ubuntu-Linux Konsole (Aufruf mit `Strg+Alt+T`):



Umgang mit der Linux Konsole

Kurze Befehlsübersicht

- **ls** (list), **ls -l**, **cd** (change directory), **mv** (move), **cp** (copy)
- **mkdir** (make directory), **rm** (remove)
- **chmod** (change mode)
- **vim** (vi Improved) als konsolenbasierter Texteditor
- **‹Befehl› - -help** zeigt die Hilfeseite zum Befehl an

Was ist Python?

- **Python** ist eine interpretierte, höhere Programmiersprache.
- Wir werden **Python** als objektorientiere Programmiersprache nutzen.
- **Python** wurde im Februar 1991 von Guido van Rossum am Centrum Wiskunde und Informatica in Amsterdam veröffentlich.
- **Python** ist in den Versionen 3.4.1 und **2.7.8** verbreitet, wir werden Letztere verwenden.

Programmierung mit Python

Wie sage ich dem Computer was er zu tun hat?

- **Python**-Programme sind Textdateien bestehend aus nacheinander aufgeführten Anweisungen.
- Ausführen eines Programms heißt: Diese Dateien werden einem Programm übergeben, der die Anweisungen so interpretiert, dass sie vom Betriebssystem verarbeitet werden können.
- Die **Python** Programmiersprache legt fest, wie diese Anweisungen in einer Datei stehen dürfen
- **CPython** ist ein ausführbares Programm (Binary), der sogenannte **Python-Interpreter**, das diese Anweisungen in einen ausführbaren Binärcode umwandelt.

Besonderheiten von Python

- **Alles** ist ein Objekt
- Vielfältig erlaubte Programmierparadigmen: objekt-orientiert, funktional, reflektiv
- Whitespace sensitiv: Einrückung entscheidet über Gruppierung von Anweisungen in logischen Blöcken
- Dynamisch typisiert: Jedes Objekt hat einen eindeutigen Typ, der aber erst zur Laufzeit feststeht

Hello world!

hello_world.py

```
print("Hello world!") # This is a comment
```

Es folgt die explizite Übersetzung der Datei in der Konsole:

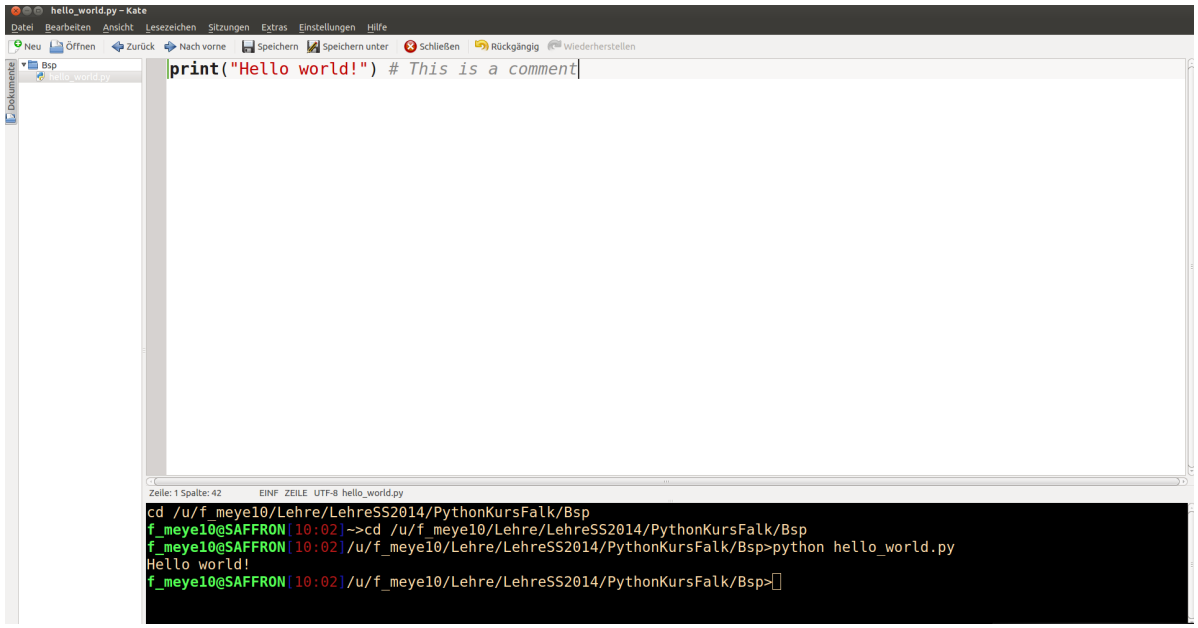
```
› python hello_world.py
```

Mit der Ausgabe:

```
Hello world!
```

Alternativ empfiehlt sich die Verwendung einer **IDE**, z.b. **PyCharm** als spezielle **Python IDE**. Wir werden den Texteditor **Kate** verwenden.

Kate



- Kate starten und konfigurieren

Built-in types

Built-in types in Python sind u.a.:

- 3 numerische Typen: **integers** (int), **floating point numbers** (float) und **complex numbers**
- Boolean type mit Werten **True** und **False**
- Text sequence type **string** (str)
- Sequence types: **list**, **tuple** und **range**
- Set types: **set** und **frozenset**
- Mapping type **dictionary** (dict)

Variablen, Zuweisungen und Typen

vazutyp.py

```
x = 1                  # Variable x ist Objekt des Typs int
y = 1.0                # Variable y ist Objekt des Typs float

print("x = {} has the Type {}".format(x,type(x)))
print("y = {} has the Type {}".format(y,type(y)))

x += 3                 # Das Gleiche wie x = x + 3
y *= 2                 # Das Gleiche wie y = y * 2

print(x)
print(y)
```

Grundlegende Operationen

Wie rechnet man in Python?

Seien **x**, **y** und **z** Variablen, dann sind in **Python** u.a. folgende binäre und unäre Rechenoperationen möglich:

Addition:	**z=x+y**	**x=x+y** bzw. **x+=y**
Subtraktion:	**z=x-y**	**x=x-y** bzw. **x-=y**
Multiplikation:	**z=x*y**	**x=x*y** bzw. **x*=y**
Division:	**z=x/y**	**x=x/y** bzw. **x/=y**
Modulo:	**z=x%y**	**x=x%y** bzw. **x%=y**
Potenzieren:		**x=x*x** bzw. **x**2** usw.

Variablen, Zuweisungen und Typen

strings.py

```
# Strings koennen auf verschiedenen
# Wegen definiert werden
name            = 'Bond'
prename         = "James"
salutation      = """My name is"""

# Ausgabe
agent = salutation+" "+name+", "+prename+" "+name+"."
print(agent)
```

Boolean type und logische Verknüpfungen

Python stellt verschiedene binäre, logische Operatoren bereit um Wahrheitswerte miteinander zu vergleichen:

- **or**, **and**, **is** und **==**

Außerdem existieren unäre Operatoren wie **not**.

Das Ergebnis ist in jedem Fall wieder ein Wahrheitswert.

Boolean Typ und logische Verknüpfungen

logical.py

```
# Wahrheitswerte
x, y = True, False
# Andere
v, w = None, 0

# Ausgabe der Auswertung unter
# logischen Operatoren
print(  x or y  )
print(  x and y )
print(  v is w  )
print(  w == y  )
print(  w is y  )
```

Sequence types

sequence.py

```
# list ist mutable
l = list()
print(l)
l = [1, '2', list()]
print(l)
l[0] = 9
print(l)
# range()
r = range(0,4,1) # dasselbe wie range(4)
print(r)
r = range(2,-6, -2)
print(r)
# tuple ist immutable
t = tuple()
t = ('value', 1)
print(t)
t[0] = t[1]      # error
```

Set type und dictionary type

setdict.py

```
# set
s = set()
s = set([1,2,3])
print(s)
print(s == set([1,2,2,1,3])) # getreu Mengendefinition

# dictionary
d = dict()
d = {'key':'value'}
d = {'Paris':2.5, 'Berlin':3.4}
print("Einwohner Paris:  {} Mio".format(d['Paris']))
print("Einwohner Berlin: {} Mio".format(d['Berlin']))
```

Kontrollstrukturen

Python bietet folgende Kontrollstrukturen mit Schlüsselwörtern an:

- Bedingte Verzweigung mit **if** - **elif** - **else**
- Bedingte Schleife mit **while**
- Zählschleife mit **for**

Achtung!

Zu beachten ist die Whitespace Sensitivität von **Python**. Es empfiehlt sich mit Tabulator-Einrückungen zu arbeiten.

Kontrollstrukturen

verzweigung.py

```
# Verzweigung mit if
condition = True or False

if condition:
        print("Condition's true!")
elif 1 == 2:
        print("1 is 2!")
else:
        print("Nothings true here :(")
```

Kontrollstrukturen

schleife.py

```
# while-Schleife
a = 0
while a < 5:
        a+=1
        print(a)

# for-Schleife
for i in range(0,4,1):
        print(i)
```

Guter Programmierstil

Was ist im Allgemeinen zu empfehlen?

- Programmteile übersichtlich gruppieren und besonders bei Verzweigungen sowie Schleifen mit Einrückungen arbeiten (**Python** erzwingt dies automatisch!)
- Genügend Kommentare einfügen um potentielle Leser die Funktionsweise des Codes nahezubringen
- Genügend Kommentare einfügen um stets selbst zu verstehen was man programmiert hat - wichtig für Fehlerbehandlung!
- Variablennamen sinnvoll wählen
- Große Quelldateien in diverse, übersichtlichere Module aufteilen

Zusammenfassung

Was haben wir heute kennengelernt?

- Grundlegender Umgang mit der **Linuxkonsole**
- Erstellen und Übersetzen einer **.py** Datei
- Grundlegender Umgang mit dem Editor **Kate**
- Grundlagen der Programmiersprache **Python**: Zuweisungen, Variablen, Kontrollstrukturen und Typen

Was fehlt noch?

- Funktionen, Klassen und Vererbung in **Python**
- Numerik mit **Python** - das Modul **NumPy** und die **Matplotlib**
- Generatoren, Lambdas, Comprehensions

Funktionen

Funktionen in **Python** werden mit dem Schlüsselwort **def**, dem Funktionsnamen und der übergebenen Parameterliste wie folgt definiert:

```
def Funktionsname(a,b,...):
    ‹Anweisungen›
```

Achtung!

Den Doppelpunkt ":" und das Einrücken im Anweisungsblock bei der Definition der Funktion nicht vergessen!

Funktionen

function.py

```
# Definiere Summenfunktion
def summe(a, b):
        return a+b

result = summe(1,2)
print(result)
# Funktionen sind auch immer Objekte
diff = summe
print(result-diff(1,2))
```

Funktionen

functionobj.py

```
# Produktfunktion
def product(a,b):
        return a*b
# Funktion: wende a,b auf op an
def execute(a,b,op):
        return op(a,b)

# Funktionen lassen sich wie Objekte
# als Parameter uebergeben
print(execute(3,5,product))
```

Geltungsbereich (Scope) von Variablen

scope.py

```
def function():
        x = 1

if True:
        y = 2

# Kontrollstrukturen erzeugen keinen lokalen scope
print(y)
# Funktionskoerper hingegen schon - Error
print(x)
```

Python Dateien als Script ausführen

Eine **‹Name›.py** Datei lässt sich auch als Skript im Terminal ausführen:

```
› ./ <Name> .py
```

wenn folgende Struktur eingehalten wird:

script.py

```
#! /usr/bin/env python

def main():
        # ----
        # Code
        # ----
        print("Hallo!")

if __name__ == '__main__':
        main()
```

Kontrollstrukturen - Fortsetzung

In Kontrollstrukturen können weitere, hilfreiche Schlüsselwörter benutzt werden:

- **continue**: Springe sofort zum Anfang des nächsten Schleifendurchlaufs.
- **break**: Brich Schleife ab.

In einer **for**-Schleife kann zudem durch Einträge eines **sequence types** oder eines **dictionary types** iteriert werden.

Kontrollstrukturen - Fortsetzung

control.py

```
for i in range(5):          # Das Gleiche wie range(0,5,1)
        if i == 2:
                continue# Zur naechsten Iteration
        print(i)
        if i == 3:
                break    # Bricht Schleife ab
# enumerate listet Zahlen auf
for i, value in enumerate(range(5)):
        print("Die {}-te Zahl ist {}".format(i,value))
# Schleife durch dictionary
dic = { "Paris": 2.5, "Berlin": 3.7, "Moskau": 11.5 }
for key, value in dic.items():
        print("{}: {} Mio Einwohner".format(key,value))
```

Strings - Fortsetzung

Die Ausgabe von Strings kann mithilfe von Angaben in { } in der formatierten Ausgabe spezifiziert werden:

stringout.py

```
x, y = 50/7.0, 1/7.0
print("{}".format(x))                  # 7.14285714286
print("{1}, {0}".format(x,y))          # 0.1428..., 7.1428...
print("{0:f} {1:f}".format(x,y))       # 6 Nachkommastellen
print("{:e}".format(x))                # Exponentialformat
print("{:4.3f}".format(x))             # Ausgabe 4 Stellen
                                       # 3 Nachkommastellen
print("{0:4.2e} {1:4.1f}".format(x,y))
print("{0:>5} ist {1:<10.1e}".format("x",x))
print("{0:5} ist {1:10.1e}".format("y",y))
print("{0:2d} {0:4b} {0:2o} {0:2x}".format(42))
```

Exceptions

In **Python** werden unter anderem folgende (Built-in) Fehlertypen unterschieden:

- **SyntaxError**: Ungültige Syntax verwendet
- **ZeroDivisonError**: Division mit Null
- **NameError**: Gefundener Name nicht definiert
- **TypeError**: Funktion oder Operation auf Objekt angewandt, welches diese nicht unterstützt
- **StandardError**, **ArithmeticError**, **BufferError** uvm.

Für eine vollständige Liste siehe:
`https://docs.python.org/2/library/exceptions.html`

Exceptions

Schreibt man selbst Programme, so lassen sich Fehler mit einem **try** - **except** Block abfangen. Die Struktur ergibt sich wie folgt:

```
try:
        ‹Anweisungen›
except ‹Built-in error type›:
        ‹Anweisungen im Fehlerfall›
except ‹Anderer Fehler› as e:
        ‹Anweisungen im Fehlerfall›
finally:
        ‹Anweisungen, die immer ausgeführt werden›
```

Built-in errors können außerdem mithilfe des Schlüsselworts **raise** erzeugt werden:

exception.py

```
def absolut(value):
        if value < 0:
                # Built-in error erzeugen
                raise ValueError()

a = int(raw_input("Zahl: ")) # Eingabe des Nutzers

try:
        absolut(a)
except ValueError:
        print("Eingabe zu klein!")
except Exception as e:
        print("Anderer Fehler: "+str(e))
finally:
        print("Dies wird immer ausgefuehrt!")
```

Klassen

Eine **Klasse** ist eine Vorlage für gleichartige Objekte. Sie legt fest welche Datentypen und Funktionen Objekte dieser Klasse (**Instanzen**) besitzen. Als Beispiel lässt sich ein Pendant aus dem Alltag heranziehen: Die Klasse Auto gibt technische Eigenschaften eines Automobils vor (4 Räder, Chassis, Motor etc.). Einzelne Autos sind Instanzen dieser Klasse, die eine ähnliche Funktionalität bieten, aber unterscheidbar sind.

Das Konzept der **Klasse** stellt die Grundlage der **objekt-orientierten Programmierung** dar.

Klassen

Syntax zur Definition einer Klasse:

```
class Myclass(object):
        # Konstruktor
        def _ _init_ _(self,...):
                ‹Anweisungen›

        def function(...):
                ‹Anweisungen›
        ...
```

Klassen

class1.py

```
# Klassendefinition
class MyClass(object):
        def __init__(self,msg):
                self._member = msg

        def some_function(self):
                print(self._member)

# Objekte der Klasse instanziieren
obj1 = MyClass("Hinter dir! Ein dreikoepfiger Affe!")
obj1.some_function()
obj2 = MyClass("Ich verkaufe diese feinen Lederjacken")
obj2.some_function()
```

Klassen

class2.py

```
class car(object):
        def __init__(self, color):
                self.color = color
                self.speed = 0

        def accelerate(self):
                self.speed += 10

        def info(self):
                print("This car is "+str(self.color) \
                        + " and its speed is "+str(self.speed) \
                        +" km/h.")

myCar = car("blue")
myCar.info()
myCar.accelerate()
myCar.info()
```

Klassen

class3.py

```
class outer_class(object):
        def __init__(self):
                self._member_variable = 1        # _ Variable "privat"

        class inner_class(object):
                class_level_member = 2

        def report(self):
                print(self._member_variable)

outer = outer_class()
inner = outer_class.inner_class()
inner.class_level_member = 4              # Aendert Instanz member
new_inner = outer_class.inner_class()
print(new_inner.class_level_member)       # = 2
outer_class.inner_class.class_level_member = 4   # Aendert type member
new_inner = outer_class.inner_class()
print(new_inner.class_level_member)       # = 4
```

Klassen

Es ist ebenso möglich statische Methoden und Klassenmethoden einer Klasse als **decorator** ohne Instanziierung eines Objekts aufzurufen.

Statische Methoden sind Funktionen in einer Klasse, die keine Membervariable verändern, d.h. die Klasse bzw. das Objekt einer Klasse nicht modifizieren. Der Parameter **self** wird dabei nicht mitübergeben.

Klassenmethoden sind Funktionen, die die Eigenschaften (d.h. Variablen) einer Klasse verändern. Der Parameter **self** wird dabei mitübergeben.

Klassen

classDeco.py

```
class MyClass(object):

        class_var = 1

        @staticmethod
        def say(some_parameter):
                return some_parameter

        @classmethod
        def hear(self):
                return self.class_var

print(MyClass.say("Hello!"))
print(MyClass.hear())
```

Klassen

Magic Members

Mithilfe besonderer Methoden und Attribute - sogenannter **magic members** - lassen sich einer Klasse spezielle Funktionalitäten geben.

Die Namen dieser Methoden beginnen und enden mit “_ _” . Sie werden meist nicht mit ihrem Namen benutzt, sondern implizit verwendet.

Klassen

Magic Members

Beispiele für ein Objekt **obj** sind:

- _ _ **init** _ _: Wird bei Erzeugung einer neuen Klasseninstanz aufgerufen.
- _ _ **str**_ _: Gibt an was **str(obj)** zurückgibt, nützlich für **print(obj)**.
- _ _ **dict**_ _: Speichert Member des Objekts in einem dictionary.
- _ _ **call**_ _: Instanzen einer Klasse wie eine Funktion aufrufen.
- Für Vergleichsoperationen: _ _ **eq**_ _, _ _ **lt**_ _, _ _ **le**_ _, _ _ **gt**_ _, _ _ **ge**_ _ usw.
- Für binäre Operationen: _ _ **add**_ _, _ _ **sub**_ _, _ _ **mul**_ _, _ _ **div**_ _ usw.

Vererbung

Ein Vorteil des Klassenkonzepts ist die Möglichkeit der **Vererbung**:

- Klasse **A** sei abgeleitet von Klasse **B**.
- Klasse **A** erbt alle Membervariablen und Membermethoden der Klasse **B** - sofern sie diese nicht überschreibt.
- Bei Klassen ähnlicher Struktur und Anwendung wird so viel Programmieraufwand gespart und eine gewisse Klassenhierarchie etabliert.
- Außerdem können durch Vererbung Funktionalitäten in abgeleiteten Klassen erzwungen werden.
- Klassisches Beispiel: Geometrische Objekte der Ebene, z.b. Kreis, Dreieck, Rechteck teilen gemeinsame Eigenschaften wie den Flächeninhalt, welcher dann jeweils passend berechnet werden muss.

Vererbung

inherit1.py

```
class BaseClass(object):

        def __init__(self,msg):
                self._msg = msg

        def report(self):
                print(self._msg)

class DerivedClass(BaseClass):

        def __init__(self,msg):
                self._msg = msg

basis = BaseClass("Hier Basis!")
deriv = DerivedClass("Ich bin abgeleitet!")

basis.report()
deriv.report()
```

Vererbung

inherit2.py

```
class BaseClass(object):
        pass

class DerivedClass(BaseClass):
        pass

issubclass(DerivedClass, BaseClass)      # True

basis   = BaseClass()
derived = DerivedClass()
isinstance(DerivedClass, BaseClass)      # False
isinstance(derived, BaseClass)           # True
isinstance(basis, BaseClass)             # True
```

Vererbung

In **Python** ist **mehrfache** Vererbung möglich:

inheritMult.py

```
class LinkeBasis(object):
        def shout(self): return "links"

class RechteBasis(object):
        def shout(self): return "rechts"

class LinksRechts(LinkeBasis, RechteBasis):
        pass
class RechtsLinks(RechteBasis, LinkeBasis):
        pass

lr = LinksRechts()
rl = RechtsLinks()
print(lr.shout())          # -> links
print(rl.shout())          # -> rechts
```

Zusammenfassung

Was haben wir bisher kennengelernt?

- Grundlagen der Programmiersprache **Python**: Funktionen, Strings, Exceptions, Klassen und Vererbung

Was fehlt noch?

- Numerik mit **Python** - das Modul **NumPy** und die **Matplotlib**
- Generatoren, Lambdas, Comprehensions
- Die **Python** Standardbibliothek

Lesen und Schreiben von Dateien

In **Python** lassen sich einfache Dateien öffnen, schreiben und lesen. Folgende Funktionen stellt **Python** dazu bereit:

- **open()**: Öffnen einer Datei
- **write()**: Einzelne strings in Datei schreiben
- **writelines()**: Liste von strings in Datei schreiben
- **readlines()**: Liest Zeilen einer Datei in Liste von strings
- **readline()**: Liest einzelne Zeile einer Datei in einen string
- **read()**: Liest alle Zeilen einer Datei in einen string

Lesen und Schreiben von Dateien

Außerdem lässt sich dem Befehl **open( )** ein zusätzliches Attribut übergeben, dass die Art des Zugriffs auf eine Datei regelt:

- **r**: Öffnen zum Lesen (Standard)
- **w**: Öffnen zum Schreiben - impliziert Überschreiben
- **a**: Öffnen zum Schreiben am Ende der Datei
- **r+**: Öffnen zum Lesen und Schreiben am Anfang der Datei
- **w+**: Öffnen zum Lesen und Schreiben, Dateiinhalt zuvor gelöscht
- **a+**: Öffnen zum Lesen und Schreiben am Ende der Datei

Lesen und Schreiben von Dateien

lesen.py

```
# Datei oeffnen
d = open("Beispiel.txt","a+")

# Lesen
contents = d.read()
if contents != "":
        print(contents)
else:
        print("Datei ist leer!\n\n")

# Nutzereingabe
text = raw_input("Schreibe an das Ende der angezeigten Zeilen: ")

# Schreiben mit Zeilenumbruch am Ende
d.write(text+"\n")

# Datei schliessen
d.close()
```

Import von Quelldateien

Bei größeren Programmierprojekten bietet es sich zur besseren Übersicht an, verschiedene Programmteile in verschiedenen Quelldateien auszulagern. Entsprechende Dateien können dann per **import** Funktion in **Python** in andere Quelldateien importiert werden. Alle importierten Klassen und Funktionen sind dann verwendbar.

Folgende Befehle stehen hierzu zur Verfügung:

- **import** ‹Pfad/Dateiname›
  # Verwendung per ‹Dateiname›.‹Funktion/Klasse›
- **from** ‹Pfad/Dateiname› **import** ‹Funktion/Klasse› [as ‹Name›]
  # Verwendung per ‹Funktion/Klasse›

Import von Quelldateien

importHead.py

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

importSource.py

```
import importHead # importHead.py liegt im gleichen Verzeichnis
z = importHead.summe(2,3)
print(z)
# Alternative
from importHead import summe # Einzelne Funktion
z = summe(2,3)
print(z)
#Alternative
from importHead import summe as importSumme
z = importSumme(2,3)
print(z)
```

IPython

IPython ist eine Kommandokonsole für das interaktive Verarbeiten von Befehlen in diversen Programmiersprachen, insbesondere **Python**. Es bietet einen flexiblen und leicht zu bedienenden **Python**-Interpreter.

Folgender Befehl in der Linux Konsole ruft IPython auf:

```
›ipython
```

Auf den folgenden Seiten werden wir in Beispielen vermehrt auf die Eingabe unter IPython zurückgreifen - sofern kein Dateiname in der Überschrift angegeben ist.

Das Modul NumPy

NumPy ist ein externes **Python** Modul für wissenschaftliches Rechnen. Es liefert mächtige **array** Objekte mit deren Hilfe effektive Berechnungen im Sinne der numerischen linearen Algebra möglich sind. Dies ist allerdings nur ein Verwendungszweck des **NumPy** Packets.

Weitere Informationen und ein ausführliches Tutorial sind zu finden unter:
`www.numpy.org`

Achtung!

Die Indexierung bei **Python** startet bei **0** und nicht bei 1!

NumPy

Arrays

```
import numpy
# 1-dim array
a = numpy.array([1,2,3,4])
a.shape == (4,)
a.dtype == numpy.int64
# 3-dim array
a = numpy.array([[1.,2.],[3.,4.],[5.,6.]])
a.shape == (3,2)
a.dtype == numpy.float64

a[1,:] # Zweite Zeile
a[:,0] *= 2 # Erste Spalte elementweise *2
print(a)
a[:,0]-a[:,1]
a[1:3,0] # == [3,5]
a*a
a.dot(a) # Fehler
a.dot(a.transpose())
```

NumPy

Arrays

```
import numpy as np
a = np.array([[1,2],[3],[4]])
a.shape == (3,)
a.dtype == object

a = np.array([[[1.,2.],[4,5]],[[1,2],[4,5]]])
a.shape == (2,2,2)
a.dtype == np.float

np.ones((4,4),dtype=complex)
np.zeros((3,3,3))
```

NumPy

Basic operations

```
from numpy import *
a = array([1,2,3])
b = arange(3)

c = a-b  # = array([1,1,1])
b**2     # = array([0,1,4])
b+= a    # = array([1,3,7])

sin(a)
a < 3   # = array([True,True, False], dtype = bool)

# Matrix
A = array([[1,0],[0,1]])
B = array([[2,3],[1,4]])
A*B # Elementweises Produkt
dot(A,B)# Matrixprodukt
```

NumPy

Unary operations

```
a = arange(5)
a.sum() # = 10
a.min() # = 0
a.max() # = 4

b = arange(6).reshape(2,3)
b.sum(axis=0) # = array([3,5,7])
b.max(axis=1) # = array([2,5])
b.cumsum(axis=0) # = array([[0,1,2],[3,5,7]])
```

NumPy

Stacking and Splitting of arrays

```
# stacking
a = arange(3)
b = arange(3,6)
vstack((a,b)) # = array([[0,1,2],[3,4,5]])
hstack((a,b)) # = array([0,1,2,3,4,5])
column_stack((a,b)) # = array([[0,3],[1,4],[2,5]])
# splitting
c = hstack((a,b))
hsplit(c,3)   # c in 3 Teile trennen
hsplit(c,(3,4)) # c nach dritter und vierter Spalte trennen
```

Achtung!

Funktionen wie **hstack()** erwarten **einen** Parameter, d.h. hstack(a,b) statt hstack((a,b)) erzeugt einen Fehler.

NumPy

Achtung!

Bei der Arbeit mit **NumPy** kann es schnell zu Fehlern bezüglich der Zuweisung gleicher Daten kommen. Folgende, sich ausschließende, Objekte können erzeugt werden:

- **Referenz**
- **View**
- **Kopie**

NumPy

Referenz

```
a = arange(4)
b = a # Kein neues Objekt! Referenz auf a
b is a  # True
b.shape = (2,2) # Aendert Form von a
a.shape # = (2,2)
```

View

```
a = arange(4)
b = a.view()
b is a  # False
b.base is a  # True - b ist View auf Daten von a
b.shape = (2,2)
a.shape    # = (4,) - Form von a wurde nicht veraendert
b[1,1] = 100 # Daten von a werden veraendert
a  # = array([0,1,2,100])
```

NumPy

Kopie

```
a = arange(4)
b = a.copy()     # Neues array mit neuen Daten
b is a          # False
b.base is a     # False
b[1] = 5
b    # = array([0,5,2,3])
a    # = array([0,1,2,3])
```

Achtung!

- Das Kopieren speicherintensiver Objekte (in der Praxis z.b. Gitter mit Millionen von Gitterpunkten) sollte unbedingt vermieden werden.
- Bei Unsicherheiten lässt sich über die **id()** Funktion feststellen, ob Variablen voneinander referenziert sind.

NumPy

Übersicht wichtiger **array** Befehle

- **Erstellung**: array(), ones(), zeros(), eye(), empty(), arange(), linspace()
- **Manipulation**: transpose(), inv(),reshape(), ravel()
- **Information**: shape, ndim, dtype, itemsize, size, print, sum(), min(), max()
- **Operationen**: dot(), trace(), column_stack(), row_stack(), vstack(), hstack(), hsplit(), vsplit()

Achtung!

Befehl **array([1,2,3,4])** korrekt, array(1,2,3,4) erzeugt Fehler!

NumPy

Universal functions

NumPy bietet die Nutzung verschiedener mathematischer Funktionen wie zum Beispiel:

- sin, cos, exp, sqrt und add

Diese agieren jeweils elementweise auf eingegebene Arrays.

Universal functions

```
from numpy import *
a = arange(4)
exp(a)    # = array([1., 2.718, ...])
sqrt(a)   # = array([0.,1.,...])
add(a,a)  # = array([0,2,4,6])
```

NumPy

Eine weitere Möglichkeit mit Matrizen zu arbeiten bietet die **matrix class** in **Numpy**:

Matrix class

```
A = matrix("1.0, 0.0; 0.0, 1.0")
type(A) # = <class 'numpy.matrixlib.defmatrix.matrix'>
A.T     # transponierte Matrix
A.I     # inverse Matrix
B = matrix("1.0, 2.0; 3.0, 4.0")
A*B     # Matrixmultiplikation
y = matrix("3.0; 2.0")
linalg.solve(A,y) # loest lineares Gleichungssystem Ax = y nach x
```

NumPy

NumPy eröffnet dem geübten Programmierer trickreichere Möglichkeiten zum Indizieren von Arrays:

Indexing

```
a = arange(10)**2       # Erste zehn Quadratzahlen
i = array([2,3,3,7,8])  # Ein Indexarray
a[i] # = array([4,9,9,49,64])

j = array([[1,2],[6,5]]) # 2-dim Indexarray
a[j] # = array([[1,4],[36,25]])

a[i] = 0
a # = array([0,1,0,0,16,25,36,0,0,81])

b = a!=0 # Boolean array
a[b] # = a ohne Werte gleich 0
```

NumPy

Wichtige Module in **NumPy**

NumPy bietet weitere Untermodule, die zusätzliche Funktionalitäten bereitstellen, unter anderem:

- **linalg**: Lineare Algebra Modul zur Lösung linearer Gleichungssysteme, Bestimmung von Eigenvektoren etc.
- **fft**: Modul für die diskrete Fourier-Transformation
- **random**: Modul für Generierung von Zufallszahlen, Permutationen, Distributionen etc.

Für weitere Informationen siehe die **NumPy** Dokumentation.

Zusammenfassung

Was haben wir heute kennengelernt?

- Importieren von Quelldateien in **Python**
- Grundlegender Umgang mit der **IPython** Konsole
- Grundlagen des Moduls **NumPy**: Arrays, Matrizen, Indizierung, Referenzen, Views und hilfreiche Module

Was fehlt noch?

- Numerik mit **Python** - die **Matplotlib**
- Generatoren, Lambdas, Comprehensions
- Die **Python** Standardbibliothek

Einführung in die Numerik

Die **angewandte Mathematik** befasst sich mit der Übertragung mathematischer Konzepte auf reale Anwendungen.

Die **Numerik** beschäftigt sich mit der konkreten Umsetzung und Herleitung entsprechender Lösungsverfahren sowie -algorithmen und deren Analyse hinsichtlich Robustheit und Effizienz.

Die **lineare Algebra** gibt Problemstellungen vor, deren effiziente Lösung Aufgabengebiet der **numerischen linearen Algebra** ist. Ein Musterbeispiel ist das Lösen eines linearen Gleichungssystems.
Die **numerische Analysis** hingegen befasst sich mit dem approximierten Lösen analytischer Probleme, insbesondere von Differentialgleichungen.

Einführung in die Numerik

Beispiel: Lösen eines linearen Gleichungssystems

- **Gegeben:** $Ax = b$ für $A \in GL(n)$, $x \in \mathbb{R}^n$ und $b \in \mathbb{R}^n$
- **Gesucht:** Die Lösung x des linearen Gleichungssystems
- **Lösungsvorschläge:**
  - Da A regulär, löse $x = A^{-1}b$
    - Unbrauchbar, denn die Bestimmung von A^{-1} ist ineffizient (entspricht der Lösung von n Gleichungssystemen $Ax = e_i$) und ist numerisch instabil;
    Anwendung der Cramerschen Regel ebenso zu aufwändig (in $O(2^n)$)
  - Einfachste Variante: **Gauß-Algorithmus** oder Dreieckszerlegung

Einführung in die Numerik

Beispiel: Lösen eines linearen Gleichungssystems

- **Gauß-Algorithmus:**

$\underline{j = 1, \ldots, n-1:}$

$r_{jk} := a_{jk}^{(j-1)}$ für $k = j, \ldots, n$

$c_j := b_j^{(j-1)}$

$\underline{i = j+1, \ldots, n:}$

$e_{ij} := a_{ij}^{(j-1)} / r_{jj}$ falls $r_{jj} \neq 0$

$b_i^{(j)} := b_i^{(j-1)} - e_{ij} c_j$

$\underline{k = j+1, \ldots, n:}$

$a_{ik}^{(j)} = a_{ik}^{(j-1)} - e_{ij} r_{jk}$

$\underline{i = n, n-1, \ldots, 1:}$

$x_i = (c_i - \sum_{j=i+1} r_{ij} x_j) / r_{ii}$ falls $r_{ii} \neq 0$

Einführung in die Numerik

Beispiel: Lösen eines linearen Gleichungssystems

- Der **Gauß-Algorithmus** hat den Aufwand $O(n^3)$
- In der **numerischen linearen Algebra** werden weitere Verfahren vorgestellt, wie z.b. die Cholesky Zerlegung und Verfahren mit orthogonalen Transformationen
- Wichtig dabei sind **Effizienz** und **Stabilität**
- Die **Kondition** des linearen Gleichungssystems zur Untersuchung der Empfindlichkeit der Lösung gegenüber Änderungen der Eingabedaten spielt eine große Rolle
- Änderungen der Eingabedaten können dabei durch die Problemstellung und die Maschinengenauigkeit bedingt werden

Einführung in die Numerik

Numerische Lineare Algebra

Die Themen der numerischen linearen Algebra umfassen u.a.:

- Fehlerrechnung
- Direkte Verfahren zur Lösung linearer Gleichungssysteme
- Iterative Lösung von Gleichungssystemen mit Fixpunktiteration
- Krylovraumverfahren zur Lösung linearer Gleichungen (u.a. CG Verfahren)
- Berechnung von Eigenwerten

Einführung in die Numerik

Beispiel: Numerische Integration

- **Gegeben:** Riemann integrierbare Funktion $f : [a, b] \mapsto \mathbb{R}$ für $a, b \in \mathbb{R}$
- **Gesucht:** $\int_a^b f(x)\mathrm{d}x$
- **Lösungsvorschläge:**
  - Finde Stammfunktion **F** von **f** und berechne Integral exakt - Unbrauchbar, da sich in praktischen Anwendungen faktisch nie eine Stammfunktion von **f** berechnen lässt, einfaches Beispiel: $f(x) = \sin(x)/x$
  - Idee: Approximiere Integral mithilfe einer **Quadraturformel** Q: $\int_a^b f(x)\mathrm{d}x = Q(f) + E(f)$ mit möglichst minimalem Fehlerterm E

Einführung in die Numerik

Beispiel: Numerische Integration

- Ein einfaches Beispiel für eine **Quadraturformel** ist die **Trapezregel:**

$$Q(f) = (b-a)\frac{f(a)+f(b)}{2}$$

- Ist f wenigstens zweimal stetig differenzierbar, so gilt für den Fehler E bei Benutzung der Trapezregel:

$$|E(f)| \leq \frac{(b-a)^3}{12} \max_{a \leq x \leq b} |f''(x)|$$

Einführung in die Numerik

Beispiel: Numerische Integration

- Die Trapezregel ist ein Spezialfall der **Newton-Cotes-Formeln**, deren Idee es ist die zu integrierende Funktion durch Polynome zu interpolieren und diese dann zu integrieren.
- Eine andere Möglichkeit Integrale zu approximieren bietet die **Gauß**-Quadratur .

Einführung in die Numerik

Numerische Analysis

Die Themen der numerischen Analysis umfassen u.a.:

- Interpolation (Polynom-, Funktions-)
- Numerische Integration
- Numerik Gewöhnlicher Differentialgleichungen
- Numerik Partieller Differentialgleichungen

matplotlib

Die **matplotlib** ist eine 2D plotting Bibliothek für Diagramme und wissenschaftliche Visualisierungen. Sie ist u.a. in **Python** und **IPython** verwendbar.

Die Visualisierungen lassen sich in vielen Aspekten manipulieren, z.b. in Größe, Auflösung, Linienbreite, Farbe, Stil, Gittereigenschaften, Schriftarten und vieles mehr.

Weitere Informationen und ein Tutorial sind zu finden unter
`matplotlib.org`
`www.loria.fr/~rougier/teaching/matplotlib`

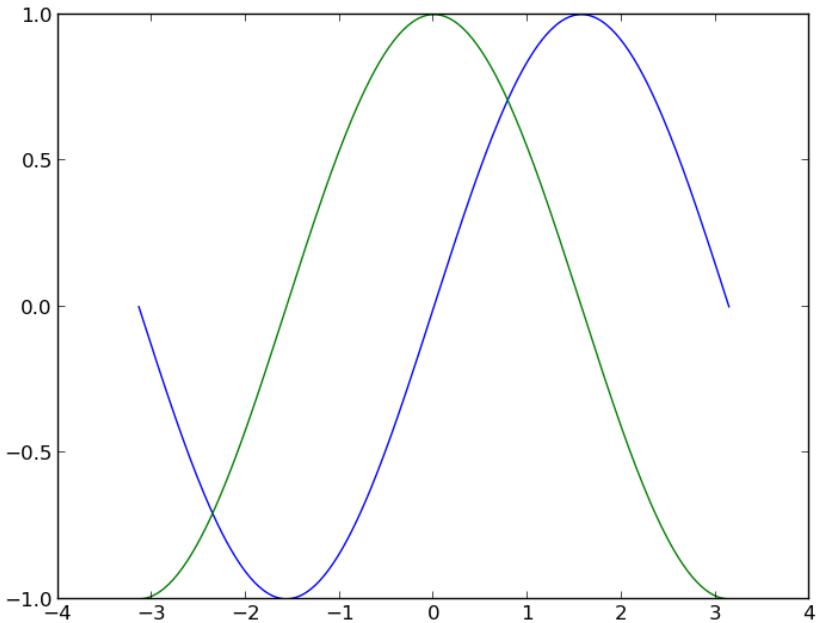
matplotlib

simple.py

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

# visualisiere sin und cos auf 256er Gitter
x = np.linspace(-np.pi,np.pi, 256, endpoint=True)
S,C = np.sin(x), np.cos(x)

# plot
plt.plot(x,S)
plt.plot(x,C)

# Erzeuge Ausgabe
plt.show()
```

matplotlib

Ausgabe:



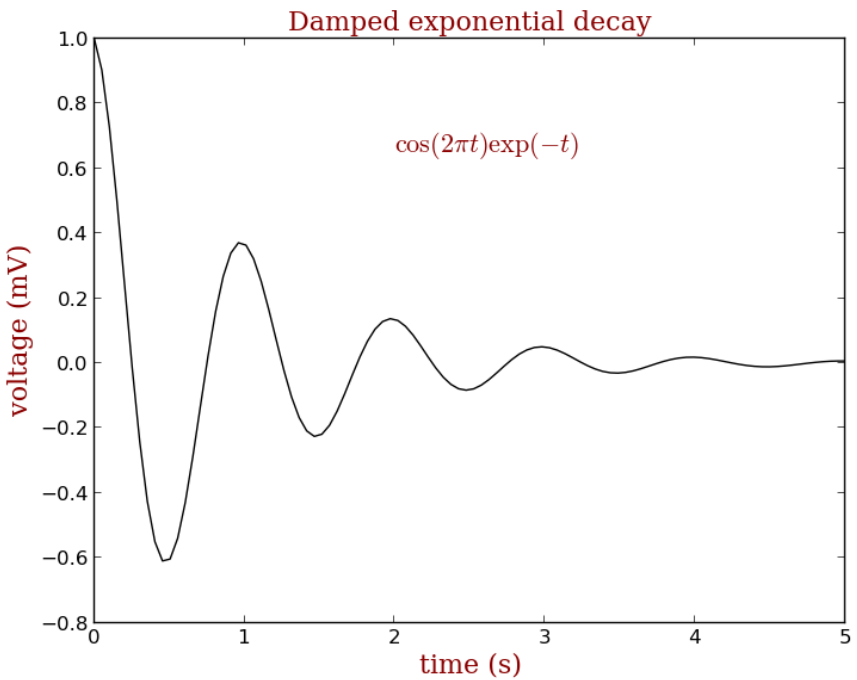
matplotlib

advanced.py

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

# dictionary for fontstyle
font = {'family' : 'serif', 'color'  : 'darkred',
        'weight' : 'normal','size'   : 16}

# numpy routines
x = np.linspace(0.0, 5.0, 100)
y = np.cos(2 * np.pi * x) * np.exp(-x)
# matplotlib routines
plt.plot(x, y,'k')
plt.title('Damped exponential decay', fontdict=font)
plt.text(2, 0.65, r'$\cos(2 \pi t) \exp(-t)$', fontdict=font)
plt.xlabel('time (s)', fontdict=font)
plt.ylabel('voltage (mV)', fontdict=font)

plt.show()
```

matplotlib

Ausgabe:



SciPy

Das **Python** basierte **SciPy** ist eine open-source Software für wissenschaftliche Anwendungen. Die bereits vorgestellten Packete **NumPy** und die **matplotlib** sowie die **IPython** Konsole sind Kernpakete von **SciPy**, machen aber nur einen Teil der Software aus.

Für weitergehende Informationen siehe die **SciPy** Webseite:
`www.scipy.org`

Debugging in Python

Ein **Debugger** dient zum Auffinden und Analyiseren von Fehlern in Soft- und Hardware. In **Python** benutzen wir ihn um Fehler in unserem Quellcode zu finden oder eine nicht beabsichtigte Funktionsweise eines Programms zu untersuchen.

Pdb ist ein interaktiver Quellcode **Python-Debugger**, der sich als Modul in den Code einbinden lässt oder über eine Konsole wie **IPython** nach Import aufrufen lässt. Eine ausführliche Erläuterung seiner Funktionen sind zu finden unter:
`docs.python.org/2/library/pdb.html`

debug1.py

```
import pdb
pdb.set_trace()
a = 2
b = 5 / 2
print(b)
```

Ausführung

```
> python debug1.py
-> a = 2
(Pdb) n
-> b = 5 / 2
(Pdb) p a
2
(Pdb) n
-> print(b)
(Pdb) n
2
-- Return --
```

debug2.py

```
print("Hier kein Fehler!")
print(x) # Fehler: x unbekannt
print("la")
print("li")
print("lu")
```

Ausführung

```
>ipython
>>> import pdb
>>> run debug2.py
...
NameError: name 'x' is not defined
>>> pdb.pm()
-> print(x) # Fehler: x unbekannt
(Pdb)
```

Zusammenfassung

Was haben wir heute kennengelernt?

- Einführung in die Numerik
- Grundlegender Umgang mit der **matplotlib**
- **Python**-Debugging mit **pdb**

Was fehlt noch?

- Generatoren, Lambdas, Comprehensions
- Die **Python** Standardbibliothek

Iteratoren

iter

```
it = iter([1,4,9])
1 == next(it)
4 == next(it)
9 == next(it)
it.next() # StopIteration Exception
```

Iteratorfunktionen

iter.py

```
def is_positive(value):
    return value > 0

def add(current_value, next_value):
    return current_value + next_value

values = [ -1, 4, -9]

absolute_values = map(abs, values)
print(absolute_values)

positive_values = filter(is_positive, values)
print(positive_values)

# optionaler 3. parameter -> startwert
summe = reduce(add, values)
print(summe)
```

List Comprehensions

Eine **List comprehension** ermöglichen dem Nutzer Listen auf folgende kurze, prägnante Weise zu erstellen:

[‹Ausdruck› **for** ‹Ausdruck› **in** ‹it.Objekt› **if** ‹Bedingung›**]**

List Comprehensions

comprehen1.py

```
# List
squaresLong = []
for x in range(5):
        squaresLong.append(x**2)

print(squaresLong)
# List comprehension
squaresShort = [x**2 for x in range(5)]
print(squaresShort)

print(squaresLong == squaresShort)
```

List Comprehensions

comprehen2.py

```
values = [ -1, 4, -9]

# aequiv. zu map(abs, values)
absolute_values = [abs(i) for i in values]
# aequiv. zu filter(is_positive, values)
positive_values = [i for i in values if i > 0]

ersteListe = values
zweiteListe = range(2)
zusammen = [ wert1 + wert2 for wert1 in ersteListe \
                           for wert2 in zweiteListe]
zusammen == [-1, 0, 4, 5, -9, -8]
# entspricht
zusammen = list()
for wert1 in ersteListe:
    for wert2 in zweiteListe:
        zusammen.append(wert1 + wert2)
```

Generatoren

Mithilfe von **Generatorfunktionen** lassen sich Funktionen definieren, die sich wie Iteratoren verhalten, also z.b. in einer Schleife verwendbar sind.

generate1.py

```
def generator_function(end):
    i = 1
    while i <= end:
        yield i # Schluesselwort yield
        i *= i+2

generator_object = generator_function(3)
next(generator_object) # 1
generator_object.next() # 3
next(generator_object) # StopIteration Exception
```

Generatoren

Besonders sinnvoll sind **Generatoren** um Speicherplatz zu sparen, wie nachfolgendes Beispiel zeigt:

generate2.py

```
def To_n_List(n): # Erstellt Liste
        num, numList = 0, []
        while num < n:
                numList.append(num)
                num += 1
        return numList

def To_n_Gen(n): # Generator
        num = 0
        while num < n:
                yield num
                num += 1

sum_of_first_n_List = sum(To_n_List(100)) # Sehr speicherintensiv
sum_of_first_n_Gen = sum(To_n_Gen(100)) # Sehr viel sparsamer
```

Generatoren

Mit der **send** Methode lässt sich einem **Generator** ein **yield** Argument von außen vorgeben:

generate3.py

```
def coroutine(start):
        end = 2 * start
        i = start
        while i < end:
                print("end {}| i {}| start {}".format(end, i, start))
                end = (yield i) or end
                i += 1

coroutine_object = coroutine(1)
coroutine_object.next()
coroutine_object.send(4)
coroutine_object.next()
```

Generatoren

Mit **Generatorausdrücken** lassen sich Generatoren herstellen, die ähnlich wie list-Comprehensions funktionieren:

Generatorausdrücke

```
# list
absolute_values = [abs(i) for i in xrange(-100,100)]
# vs. generator
absolute_values_gen = (abs(i) for i in xrange(-100,100))

absolute_values == list(absolute_values_gen)
```

Lambda Funktionen

Lambda Funktionen dienen zur Erstellung von anonymen Funktionen, d.h. Funktionen ohne Namen. Speziell bei Nutzung der **map**, **filter-** oder **reduce** Funktion sind solche **Lambda** Funktionen sehr praktisch. Sie werden wie folgt definiert:

lambda ‹Argumente›**:** ‹Ausdruck›

Lambda Funktionen

Lambda

```
# Einfache Verwendung
f = lambda x,y: x+y
f(2,3) == 5

# Verwendung auf map, filter
values = [-1,2,-3]
map(lambda x: x > 0, values) == [False,True,False]
filter(lambda x: x > 0, values) == [2]

# Verwendung auf reduce
reduce(lambda x,y: x*y, [1,2,3,4]) == 24
```

Dokumentation

Wir haben bereits kennengelernt wie einfache Kommentare im **Python** Code mit **#** integriert werden können. Um die Dokumentation eines Codes zu vereinfachen und auch extern Beschreibungen über Module, Klassen oder Funktionen zu erhalten, lassen sich **Docstrings** verwenden.

Docstrings stehen immer am Anfang eines Klassen- oder Funktionskörpers und werden mit drei doppelten oder einfachen Hochkommata eingerahmt.

Mithilfe des Attributs _ _**doc**_ _ einer Klasse oder Funktion lassen sich diese Beschreibungen dann ausgeben.

Dokumentation

docstrings.py

```
class some_class(object):
        """
        This is the docstring of this class containing information
        about its contents: it does nothing!
        """
        def __init__(self):
                pass

def some_function():
        """
        This function does nothing
        """
        pass

print(some_class.__doc__)
print(some_function.__doc__)
```

Dekoratoren

Wie bereits an Tag 2 erwähnt lassen sich in **Python** sogenannte **decorators** verwenden. Eine Funktion die eine Methode oder eine Funktion modifizieren soll und mit einem **@** vor die entsprechende Definition geschrieben wird, heißt **decorator** Funktion. Diese wirkt wie **function=decorator(function)**, lässt sich aber wie folgt schreiben:

```
@decorator
def function():
        ‹Anweisungen›
```

Ein **decorator** kann entweder als Funktion selbst oder als Klasse mit dem implementierten _ _ **call** _ _ Attribut definiert werden.

Dekoratoren

deco1.py

```
def deco(obj):
        print(obj.__doc__)
        return obj

@deco
def function():
        """
        This function does nothing
        """
        pass

a = function
```

Dekoratoren

deco2.py

```
class decorator(object):
        def __init__(self):
                self.count=0
                self.func = None
        def countfunc(self,a):
                self.count += 1
                print("Funktionsaufruf-Nummer: "+str(self.count))
                return self.func(a)
        def __call__(self,func):
                self.func = func
                return self.countfunc
@decorator()
def function(a):
        print(a)

a = function
a("Hallo!")
a("Guten Tag!")
```

Magic

Monkeypatching

- **Python** bietet die Möglichkeit zur Laufzeit Funktionalitäten zu ersetzen
- Funktionsreferenzen in Klassen und Modulen können ersetzt werden.
- Sollte sehr sparsam eingesetzt werden!

Magic

monkey.py

```
class Foo(object):
    def run(self): print('fooooo')

foo = Foo()
foo.run()

def run_bar(self): print('bar')

Foo.run = run_bar

bar = Foo()
bar.run()
```

Die Python Standardbibliothek

Python bietet unter folgender Webseite seine Standardbibliothek an:
`https://docs.python.org/2/library/`

Diese beinhaltet Dokumentationen und Beispiele der wichtigsten **Python** Funktionen, Typen und Modulen. Sie bietet insbesondere eine Suchfunktion um nach potentiell schon vorhandenen Funktionalitäten zu suchen, die man selbst in seinem Code benötigt.

Im folgenden werden einzelne nützliche Module anhand ihrer Docstrings kurz vorgestellt.

Die Python Standardbibliothek

tempfile

```
Temporary files.

This module provides generic, low- and high-level interfaces for
creating temporary files and directories.  The interfaces listed
as "safe" just below can be used without fear of race conditions.
Those listed as "unsafe" cannot, and are provided for backward
compatibility only.

This module also provides some data items to the user:

  TMP_MAX  - maximum number of names that will be tried before
             giving up.
  template - the default prefix for all temporary names.
```

Die Python Standardbibliothek

fnmatch

```
Filename matching with shell patterns.

fnmatch(FILENAME, PATTERN) matches according to the local convention.
fnmatchcase(FILENAME, PATTERN) always takes case in account.

The functions operate by translating the pattern into a regular
expression.  They cache the compiled regular expressions for speed.

The function translate(PATTERN) returns a regular expression
corresponding to PATTERN.  (It does not compile it.)
```

Die Python Standardbibliothek

shutil

```
Utility functions for copying and archiving files and directory trees.

XXX The functions here don't copy the resource fork or other metadata
on Mac.
```

Die Python Standardbibliothek

pickle

```
Create portable serialized representations of Python objects.

See module cPickle for a (much) faster implementation.
See module copy_reg for a mechanism for registering custom picklers.
See module pickletools source for extensive comments.

Classes:

    Pickler
    Unpickler
```

Die Python Standardbibliothek

argparse

```
Command-line parsing library

This module is an optparse-inspired command-line parsing library that:

    - handles both optional and positional arguments
    - produces highly informative usage messages
    - supports parsers that dispatch to sub-parsers

The following is a simple usage example that sums integers from the
command-line and writes the result to a file::

    parser = argparse.ArgumentParser(
        description='sum the integers at the command line')
```

Die Python Standardbibliothek

subprocess

```
subprocess - Subprocesses with accessible I/O streams

This module allows you to spawn processes, connect to their
input/output/error pipes, and obtain their return codes.  This module
intends to replace several other, older modules and functions, like:

os.system
os.spawn*
os.popen*
popen2.*
commands.*

Information about how the subprocess module can be used to replace these
```

Die Python Standardbibliothek

CSV

```
CSV parsing and writing.

This module provides classes that assist in the reading and writing
of Comma Separated Value (CSV) files, and implements the interface
described by PEP 305.  Although many CSV files are simple to parse,
the format is not formally defined by a stable specification and
is subtle enough that parsing lines of a CSV file with something
like line.split(",") is bound to fail.  The module supports three
basic APIs: reading, writing, and registration of dialects.


DIALECT REGISTRATION:
```

Die Python Standardbibliothek

sys

```
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules

displayhook -- called to show results in an interactive session
excepthook -- called to handle any uncaught exception other than SystemExit
  To customize printing in an interactive session or to install a custom
  top-level exception handler, assign other functions to replace these.
```

Die Python Standardbibliothek

copy

```
Generic (shallow and deep) copying operations.

Interface summary:

        import copy

        x = copy.copy(y)        # make a shallow copy of y
        x = copy.deepcopy(y)    # make a deep copy of y

For module specific errors, copy.Error is raised.

The difference between shallow and deep copying is only relevant for
compound objects (objects that contain other objects, like lists or
```

Die Python Standardbibliothek

pprint

```
Support to pretty-print lists, tuples, & dictionaries recursively.

Very simple, but useful, especially in debugging data structures.

Classes
-------

PrettyPrinter()
    Handle pretty-printing operations onto a stream using a configured
    set of formatting parameters.

Functions
---------
```

Die Python Standardbibliothek

StringIO

```
File-like objects that read from or write to a string buffer.

This implements (nearly) all stdio methods.

f = StringIO()      # ready for writing
f = StringIO(buf)   # ready for reading
f.close()           # explicitly release resources held
flag = f.isatty()   # always false
pos = f.tell()      # get current position
f.seek(pos)         # set current position
f.seek(pos, mode)   # mode 0: absolute; 1: relative; 2: relative to EOF
buf = f.read()      # read until EOF
buf = f.read(n)     # read up to n bytes
```

Die Python Standardbibliothek

re

```
Support for regular expressions (RE).

This module provides regular expression matching operations similar to
those found in Perl.  It supports both 8-bit and Unicode strings; both
the pattern and the strings being processed can contain null bytes and
characters outside the US ASCII range.

Regular expressions can contain both special and ordinary characters.
Most ordinary characters, like "A", "a", or "0", are the simplest
regular expressions; they simply match themselves.  You can
concatenate ordinary characters, so last matches the string 'last'.

The special characters are:
```

Zusammenfassung

Was haben wir heute kennengelernt?

- Iteratoren, Comprehensions, Generatoren
- Lambda Funktionen und Dekoratoren
- Monkeypatching und Docstrings
- Die **Python** Standardbibliothek