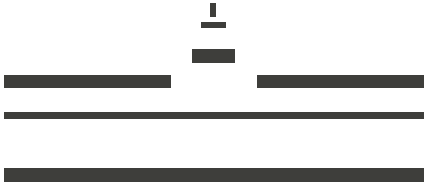


Westfälische Wilhelms-Universität Münster

Angewandte Mathematik Münster

Pythonkurs Wintersemester 2016/17

Einführung in die Programmierung zur Numerik mit Python

wissen.leben
WWU Münster

Stephan Rave

27.-31. März 2017

Organisation

- Anwesenheitsliste
- Teilnahmenachweis / Allgemeine Studien: Teilnahme an allen 5 Tagen
- Logische Grundlagen und Programmierung (1-Fach Bachelor, PO 2014): Teilnahme an allen 5 Tagen und erfolgreiches Bearbeiten einer Hausaufgabe
- Vorkenntnisse?
- Funktioniert der Login?

Python/NumPy Tutorials

- Programming for Computations – Python
- A Byte of Python
- Python-Tutorial
- Dive Into Python 3
- Numpy-Tutorial
- Numpy for Matlab Users

Hilfe!

- Google
- stackoverflow
- Python 3 Sprach-Referenz
- Python 3 Standardbibliothek-Referenz
- NumPy-Referenz

Übersicht

Tag 1: Python Grundlagen

Tag 2: Kontrollstrukturen, Exceptions und Funktionen

Tag 3: Numerik mit Python - das Modul NumPy

Tag 4: matplotlib, numerische Mathematik, sowie Klassen und Vererbung in Python

Tag 5: Verschiedenes

Programmierung in der Numerik

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 $A \in \mathbb{R}^{m \times n}$
  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?

Programmierung in der Numerik

- 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 Unbekannten als auch durch die Nichtexistenz einer analytischen(exakten) Lösung bedingt.

- Die Vorlesung „Numerische Lineare Algebra" beschäftigt sich u.a. mit der Theorie von Algorithmen zur Lösung großer linearer Gleichungssysteme. Für die Umsetzung dieser Algorithmen verwenden wir Programmiersprachen wie Python, Matlab, Julia, C, C++ oder Fortran.

Was ist Python?

- **Python** ist eine interpretierte, objektorientierte, dynamisch-typisierte höhere Programmiersprache mit automatischer Speicherverwaltung.
- **Python** wurde im Februar 1991 von Guido van Rossum (BDFL) am Centrum Wiskunde und Informatica in Amsterdam veröffentlich.
- **Python** ist in den Versionen 2.x und **3.x** verbreitet (nicht abwärtskompatibel!). Wir werden Python 3 verwenden.

Achtung!

Supportende für Python 2 in 2020.

Programmierung mit Python

Wie sage ich dem Computer was er zu tun hat?

- **Python-Programme** (auch: Python-Skripte) sind Textdateien bestehend aus nacheinander (oder mit Sprüngen) auszuführenden Anweisungen.
- Die **Python Programmiersprache** legt die Form der Anweisungen fest, die in der Datei stehen dürfen.
- **CPython** ist ein ausführbares Programm (Binary), der sogenannte **Python-Interpreter**, das diese Anweisungen so interpretiert, dass sie vom Computer verarbeitet werden können
- Ein Python-Programm **auszuführen** bedeutet also, CPython mit dem Python-Programm als Eingabe aufzurufen:

```
> python3 mein_programm.py
```

Besonderheiten von Python

- **Objektorientierung:** *Alles* ist ein Objekt.
- **Stark typisiert:** Jedes Objekt hat einen eindeutigen unveränderlichen[1] Typ.
- **Dynamisch typisiert:** Der Typ einer Variablen entscheidet sich erst zur Laufzeit und kann sich ändern.
- **Automatische Speicherverwaltung:** Kein manuelles (fehleranfälliges) anlegen und freigeben von Speicher.
- **Whitespace sensitiv:** Einrückung entscheidet über Gruppierung von Anweisungen in logischen Blöcken.

[1] fast!

Einstieg in Python

Wie erstellt man ein Pythonprogramm? - Basis-Variante

Schreiben des Programms:

- In einem beliebigen **Texteditor** das Programm schreiben und die Datei als **.py** Datei, zum Beispiel `my_program.py` speichern.

Ausführen des Programms im Linux-Terminal:

- Terminal öffnen (`Strg + Alt + T`) und in das Verzeichnis wechseln, in dem die Datei liegt

```
> cd ordner/unterordner/unterunterordner
```

- Den Python-Interpreter aufrufen um das Programm zu starten, in diesem Fall:

```
> python3 my_program.py
```

Hello world!

Ein einfaches Standardbeispiel zum Start zeigt die Verwendung der Funktion `print` zur Ausgabe von Strings:

hello_world.py

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

Es folgt die explizite Ausführung der Datei im Terminal:

```
> python3 hello_world.py
```

Mit der Ausgabe:

```
Hello world!
```

Hello world!

Aufgaben

(1) Reproduzieren Sie obiges Beispiel mit entsprechender Erstellung einer **.py** Datei in einem geeigneten Ordner mit einem beliebigen Editor.

(2) Fügen Sie Ihrem Programm eine weitere Zeile hinzu, in der Sie einen erneuten `print` Befehl mit einem selbst gewähltem String schreiben.

(3) Python as Taschenrechner: Geben sie mit einer weiteren Zeile den Wert von 99^{99} aus.[2]

[2] a^b wird in Python als `a**b` geschrieben.

Lernkontrolle

(1) Zur Ausführung eines Python-Programms benötigt man ein(en) ...

(a) Python-Interpret. (b) Python-Compiler.

(c) Python-Skript.

(2) Die logische Unterteilung eines Python-Codes erfolgt per ...

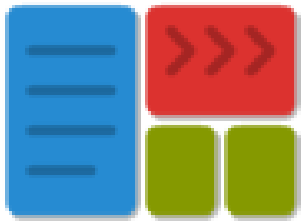
(a) Klammersetzung mit {}. (b) Auslagerung in Datei.

(c) Einrückung.

(3) Sie haben bereits die `print` Funktion kennengelernt. Welcher der folgenden Anweisungen erzeugt einen Fehler?

(a) print(3) (b) print(3+3) (c) print(3+"3")

Die Python-IDE Pyzo

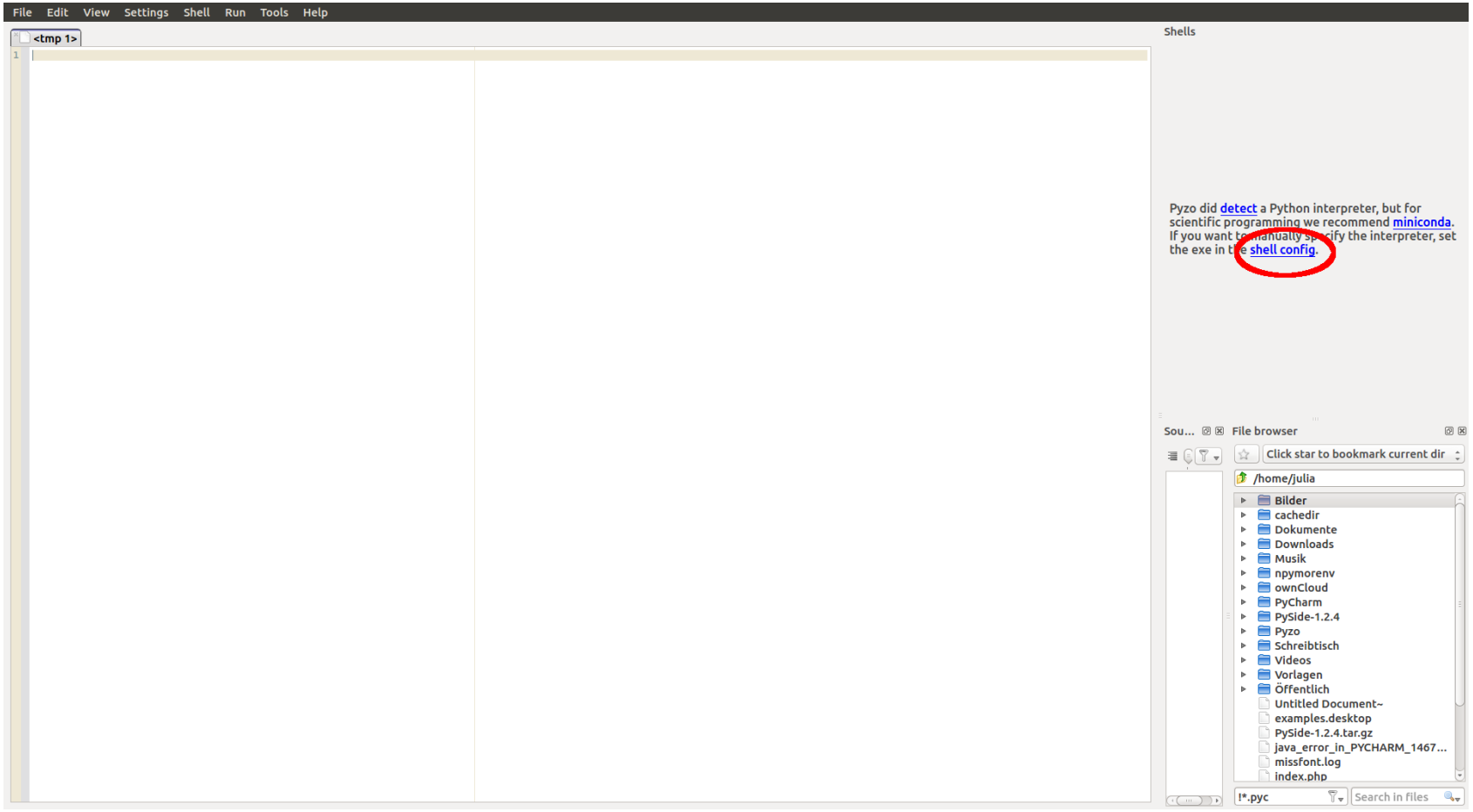


Pyzo ist eine integrierte Entwicklungsumgebung (IDE) für Python

- Installation (auf eigenen PCs): www.pyzo.org
- Gleichzeitig einfache Installation des Python-Interpreters und benötigter Pakete
- Funktionen:
  - Editor zum Schreiben von Programmen
  - Terminal zum Ausführen
  - **IPython**-Shell zum direkten Eingeben von Python-Befehlen
  - Debugger
  - ...

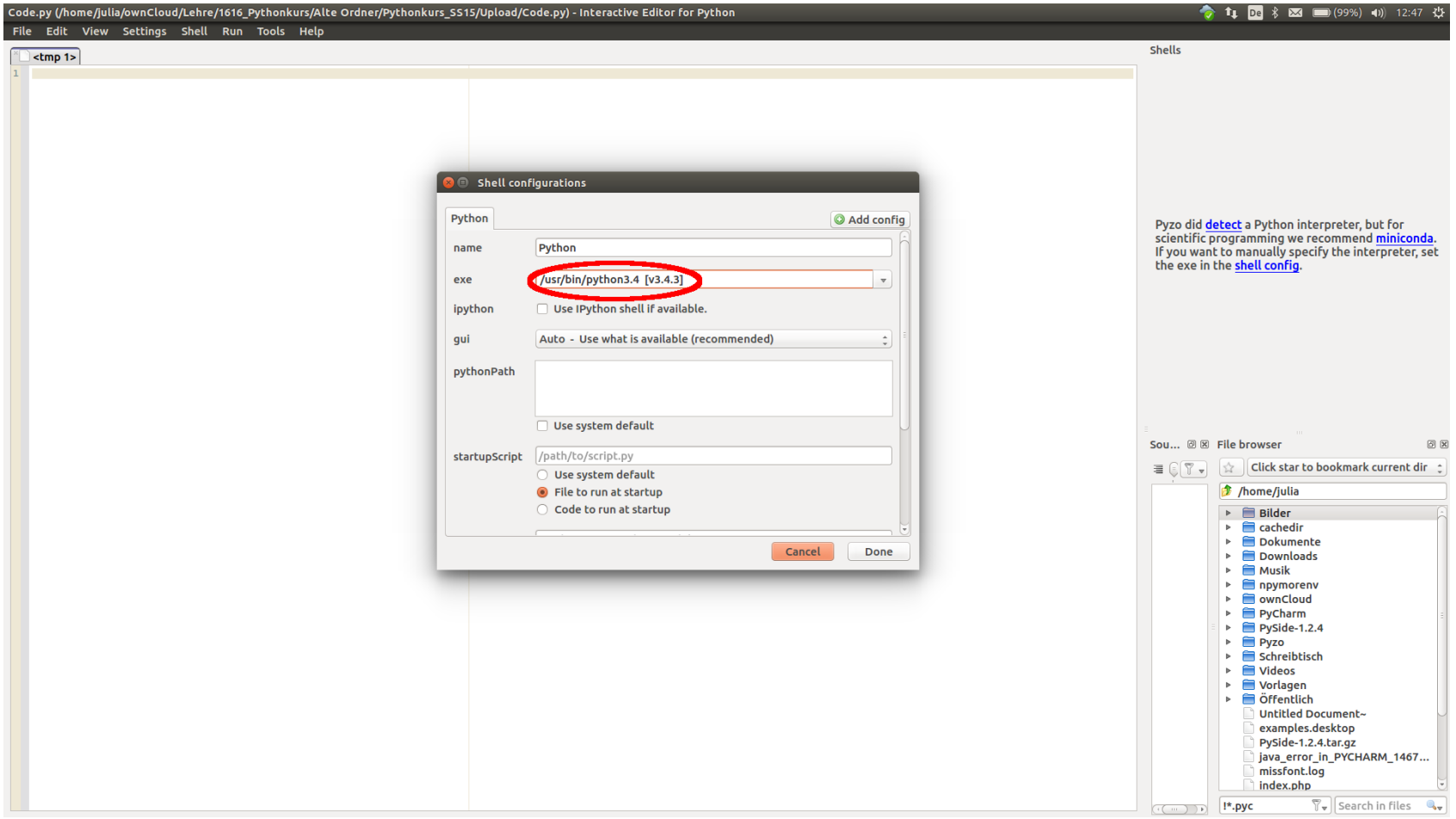
Pyzo einrichten

Wenn Shell nicht automatisch ausgewählt: `shell config`-Menü öffnen



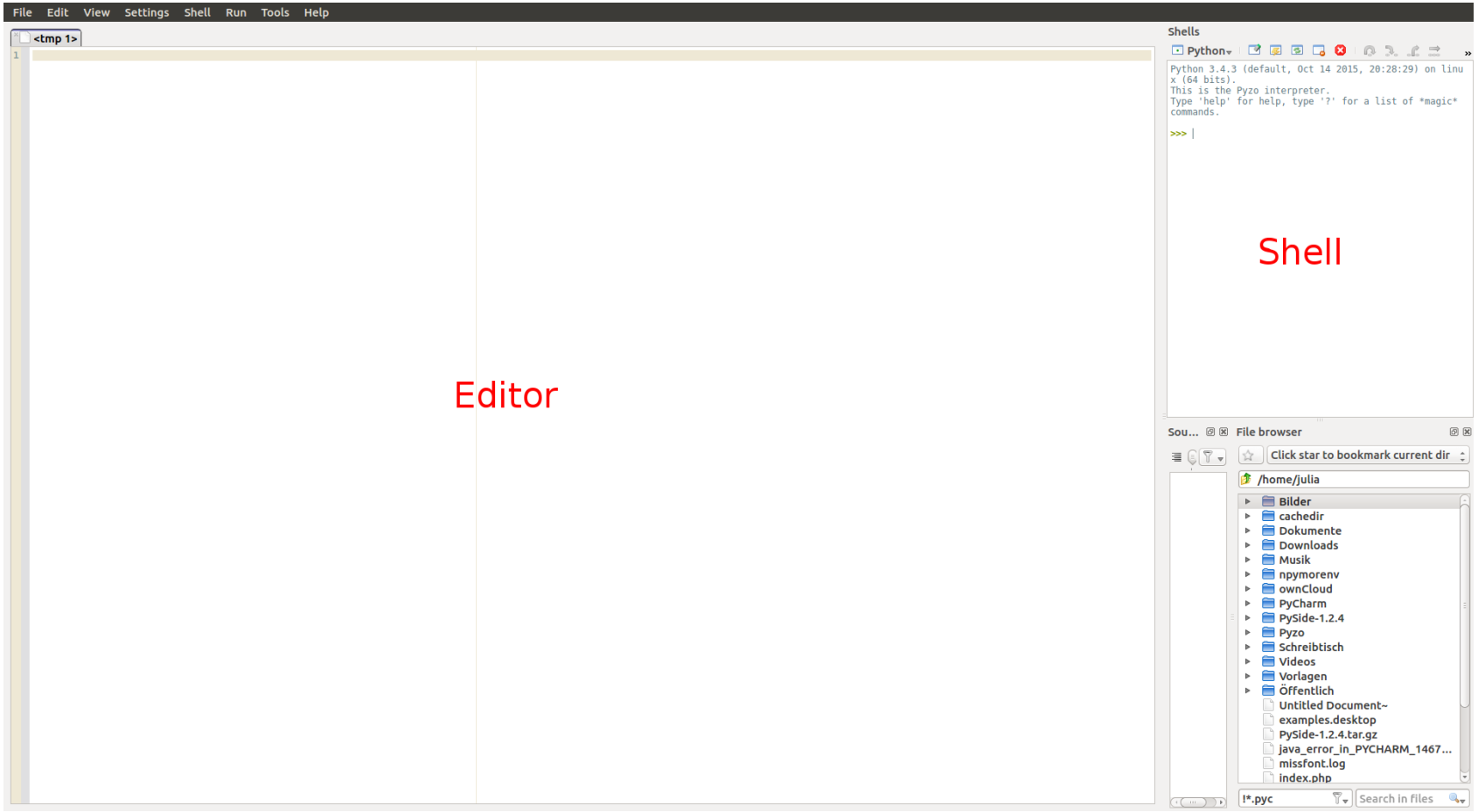
Pyzo einrichten

Python 3-Interpreter auswählen



Pyzo einrichten

Pyzo-Basis-Startbildschirm



Editor und Shell

Die **Python-Shell**:

- Führt Python-Befehle interaktiv aus.
- Befehle können einfach Zeile für Zeile eingetippt und ausgeführt werden.

$\Longrightarrow$ Nützlich für schnelles Ausprobieren von Code.

Der **Editor**:

- Programm wird im Editor geschrieben.
- Kompletter Code oder Teile können in der Shell ausgeführt werden.

Code ausführen in Pyzo

Es gibt verschiedene Arten, wie Code aus dem Editor in der Shell ausgeführt werden kann:

Interactive Mode

- **Execute file:** Kompletter Quellcode wird ausgeführt.
- **Execute selection:** Markierter Quellcode wird ausgeführt; ist nichts markiert, wird die aktuelle Zeile ausgeführt.

Interactive Mode entspricht praktisch dem Kopieren des Codes in die aktuelle Shell, die den Code dann ausführt

Code ausführen in Pyzo

Script Mode

- **Run file as script:** Neue Shell wird gestartet, Skript-Name und Arbeitsverzeichnis werden passend gesetzt, im Editor geschriebene Datei wird dort ausgeführt.

Script Mode entspricht praktisch dem Ausführen der Datei im Terminal. Die Datei muss dazu abgespeichert werden.

Achtung!

Der Code kann beim Entwickeln schnell und einfach im *Interactive Mode* ausgeführt werden, das fertige Programm **muss** am Ende aber auch im *Script Mode* funktionieren!

Pyzo

Aufgaben

Probieren Sie die Funktionen der Python-Shell und der Pyzo-Run-Optionen aus:

- Benutzen Sie die **print**-Funktion direkt in der Shell.

Öffnen Sie Ihre `hello_world.py`-Datei im Editor.

- Führen Sie den kompletten Code im *Interactive Mode* aus.
- Führen Sie nur eine Zeile aus.
- Führen Sie die Datei im *Script Mode* aus.

Namen (=Variablen) und Zuweisung

Die Zuweisung eines Objekts an einen Namen (Variable) erfolgt in Python über das Konstrukt:

<Name> = <Wert>

Im Gegensatz zu anderen Programmiersprachen wie z.B. C++ steht der Typ einer Variablen in Python erst zur Laufzeit fest, d.h. eine Variable muss nicht im Vorhinein korrekt deklariert werden.

Der Typ einer Variable legt dessen Zugehörigkeit zu einer Menge gleichartiger Datenstrukturen und damit dessen Verwendungsmöglichkeiten fest. Einfaches Beispiel sind **int** und **float** Typen in Python.

Variablen, Zuweisungen und Typen

Folgendes Programmbeispiel zeigt Definition und Verwendung einfacher Variablen:

vazutyp.py

```
x = 1              # x verweist aus das 1 Objekt des Typs int
y = 1.0            # y verweist auf das 1.0 Objekt des Typs float

print("x =", x, "has the Type", type(x))
print("y =", y, "has the Type", type(y))

x = x + 3
y = y * 2

print(x)
print(y)
```

Built-in types

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

- Numeric types: *integers* (**int**), *floating point numbers* (**float**) und *complex numbers* (**complex**)
- Boolean type (**bool**) 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**)

Operatoren

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`
(Echte) Division:	`z=x/y`	`x=x/y`	bzw. `x/=y`
Gerundete Division:	`z=x//y`	`x=x//y`	bzw. `x//=y`
Modulo:	`z=x%y`	`x=x%y`	bzw. `x%=y`
Potenzieren:	`z=x**2`	`x=x**2`	
Negation:	`z=-x`	`x=-x`	

Strings

Strings können auf verschiedene Weisen angelegt und verkettet werden:

strings.py

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

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

Sequence Types

sequence.py

```
# list ist mutable
l = list()
print(l)
l = [1, '2', list()]
print(l)
l[0] = 9
print(l)

# tuple ist immutable
t = tuple()
t = ('value', 1)
print(t)
t[0] = t[1] # error

# range ist immutable
r = range(0,4,1) # das selbe wie range(4)
print(r)
print(list(r))
```

Zuweisungen erstellen keine Kopien!

assignnocopy.py

```
l = [0, 4, 42]
k = l
k[1] = 7
print(k)
print(l)

# int, float, str sind immutable!
i = 42
j = i
j += 1  # erzeugt 43 Objekt und weist es j zu
print(j)
print(i)
```

Set type und dictionary type

setdict.py

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

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

Aufgaben

(1) Schreiben Sie ein Python-Programm, in dem 2 verschiedene Variablen x, y mit selbst gewählte Werte vom Typ **float** zugeweisen werden und dann deren Summe, Differenz und Produkt ausgegeben werden.

(2) Konstruieren sie ein **list**-Objekt, das x und y enthält, und erweitern Sie dieses schrittweise mit den Ergebnissen der Rechenoperationen aus (1). Geben Sie das Ergebnis aus.
Hinweis: Recherchieren Sie hierzu im Internet, wie man Zahlen und Objekte an Listen anhängt.

(3) Konstruieren Sie ein **dict**, dessen **keys** aus x, y sowie den Namen der Rechenoperationen aus (1) bestehen, und denen als **values** die passenden Werte zugeordnet sind. Geben Sie den Wert der Summe und der Differenz mithilfe dieser Datenstruktur aus.

Lernkontrolle

(1) Von welchem Typen ist das Ergebnis der Operation **1 + 2.0**?

(a) int (b) complex (c) float

(2) Im Gegensatz zum sequence type **list** ist ein **tuple** ...

(a) mutable. (b) immutable. (c) ein set type.

(3) Welche der folgenden Definitionen eines **dict** ist nicht zulässig?

(a) {3: 'drei'} (b) {(3,4): 'Drei und vier!' }

(c) {[3,4]: 'Drei und vier!' }

Boolean type und logische Verknüpfungen

Der Typ **bool** hat genau zwei Werte: **True** und **False**.

Allen **Python**-Objekten **o** ist mit **bool(o)** ein Wahrheitswert zugeordnet.

Python stellt logische Operatoren bereit um Wahrheitswerte zu vergleichen und einen neuen Wahrheitswert liefen:

- **not**, **or**, **and**

Operanden **o**, die nicht vom Typ **bool** sind, wird dabei implizit zu **bool(o)** umgewandelt. Der Rückgabewert von **o and/or o2** entspricht dem Operanden, der über den Wahrheitswert des Ausdrucks entschieden hat.

Python-Objekte können mit == und **is** auf **Gleichheit** bzw. **Identität** geprüft werden.

Boolean Typ und logische Verknüpfungen

logical.py

```
print(bool(7))
print(bool(0))
print(bool(0.))
print(bool([]))
print(bool([0]))
print(bool(''))
print(bool(None))
print(bool(()))

print(False or 3)
print(True and False)
print(4 or [4])
```

Boolean Typ und logische Verknüpfungen

logical2.py

```
a, b = [8], [8]
print(a == b)
print(a is b)

a = 9
b = 9
print(a == b, a is b)

a = 9999
b = 9999
print(a == b, a is b)
```

Achtung!

Der Operator **is** sollte in der Regel nur in der folgenden Form verwendet werden: <**Variable**> **is None**

Kontrollstrukturen

Python bietet folgende Kontrollstrukturen an:

- Bedingte Verzweigung:

```
if <Bedingung>:
    <Anweisungen>
elif <Bedingung>:
    <Anweisungen>
else:
    <Anweisungen>
```

- Bedingte Schleife:

```
while <Bedingung>:
    <Anweisungen>
```

Kontrollstrukturen

- Zählschleife:

for <Variable> **in** <Iterierbares Objekt>**:**
    <Anweisungen>

In einer **for** Schleife lässt sich z.B. über sequence types und dictionary types iterieren.

Die Anweisungen bzw. Anweisungsblöcke, die logisch zusammengehören, müssen in der selben Tiefe eingerückt sein. Hierzu empfiehlt sich die Benutzung der Tabulator-Taste.

Achtung!

Stellen Sie Ihren Editor so ein, dass die Tabulator-Taste eine feste Anzahl an Leerzeichen ausgibt!

Kontrollstrukturen

verzweigung.py

```
# Verzweigung mit if
condition = "" or (3 - 3)

if condition:
    print("Condition is true!")
elif 1 == 2:
    print("1 is equal to 2!")
else:
    print("Nothing 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)
```

Aufgaben

(1) Schreiben Sie ein Python-Programm, in dem zunächst zwei leere Listen angelegt werden und eine beliebige positive **Integer**-Zahl in einer Variablen **end** gespeichert wird.
Ihr Programm soll nun mithilfe einer **for** oder **while** Schleife unter Verwendung eines **if** - **else** Verzweigungsblock alle natürlichen Zahlen kleiner gleich **end** in jeweils gerade und ungerade Zahlen in die zuvor definierten Listen aufteilen und speichern. Geben Sie diese Listen aus.

Lernkontrolle

(1) Welchem Wert entspricht der folgende Ausdruck?
`False or ((True and False) or (True is False) or (True is True and (True or False)))`

(a) True (b) False (c) 2

(2) Welcher der folgenden Typen kann in einer for-Schleife nicht als Typ des iterierbaren Objekts verwendet werden?

(a) list (b) dict (c) int

(3) Welcher der folgenden Ausdrücke erzeugt für `a=5` und `b=[1,2]` eine Endlosschleife?

(a) for i in range(a): a+=1

(b) while a < 5: a+=1

(c) for i in b: b.append(i)

Guter Programmierstil

Was ist 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.
- PEP 8.

Zusammenfassung

Bisherige Themen

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

Kommende Themen

- Mehr zu Kontrollstrukturen
- Input/Output
- Exceptions
- Funktionen

Pyzo erweitern

Pyzo kann mit verschiedenen Tools erweitert werden. Diese können im Tools-Menü aktiviert werden.
Besonders nützlich sind:

- **File Browser**
- **Workspace**: Eine Übersicht aller aktuell in der Shell definierten Variablen
- **Interactive Help**: Hilfe-Funktion, mit der nach Infos über Module, Objekte, etc. gesucht werden kann. Wird beim Schreiben im Editor passend aktualisiert.

Alle einzelnen Pyzo-Komponenten können je nach Wunsch beliebig platziert werden.

Kontrollstrukturen

In Kontrollstrukturen können weitere, hilfreiche Schlüsselwörter innerhalb einer Schleife benutzt werden:

- **continue**: Springt sofort zum Anfang des nächsten Schleifendurchlaufs.
- **break**: Bricht Schleife ab.

Kontrollstrukturen

control.py

```
for i in range(5):
    if i == 2:
        continue    # Zur naechsten Iteration
    print(i)
    if i == 3:
        break       # Bricht Schleife ab

# enumerate zaehlt Elemente eines iterables auf
for i, value in enumerate(range(0,10,2)):
    print('Die', i, '-te Zahl ist', value)

# Schleife ueber key/value-Paare eines dicts
dic = { 'Paris': 2.5, 'Berlin': 3.7, 'Moskau': 11.5 }
for key, value in dic.items():
    print(key + ':', value, 'Mio Einwohner')
```

Strings - Fortsetzung

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

stringout.py

```
x, y = 50/7, 1/7
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 (builtin) 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.
- **AttributeError**, **KeyError**, **IOError** und viele mehr

Für eine vollständige Liste siehe:
`https://docs.python.org/3/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 <Fehler-Typ>:
        <Anweisungen im Fehlerfall>
except <Anderer Fehler> as e:
        <Anweisungen im Fehlerfall>
finally:
        <Anweisungen, die immer ausgeführt werden>
```

Exceptions kann man selber mithilfe des Schlüsselworts **raise** auslösen:

exception.py

```
def get_name():
    name = input('Name: ')  # Eingabe des Nutzers
    if len(name) == 0:
        raise ValueError('Keine Eingabe')
    return name

try:
    n = get_name()
except ValueError:
    print('Kein Name angegeben')
except Exception as e:
    print('Anderer Fehler: ', e)
finally:
    print('Dies wird immer ausgefuehrt!')

print('Hallo {}!'.format(n))
```

Aufgaben

(1) Schreiben Sie ein Python-Programm, das eine Nutzereingabe erwartet und testet, ob die Eingabe eine positive Integer-Zahl ist. Ist das nicht der Fall, so bricht das Programm mit einer selbst gewählten Fehlermeldung ab. Ist das der Fall, so werden alle natürlichen Zahlen von 0 bis zur Eingabezahl in einer Schleife addiert und das Ergebnis ausgegeben.
Hinweis: Verwenden Sie die Funktion `input()` und wandeln Sie die String-Eingabe mithilfe von `int()` in eine gültige Integerzahl um. Schlagen Sie im Zweifel in der Dokumentation von Python deren richtige Verwendung nach.

(2) Erweitern Sie Ihr Programm aus (1) wie folgt: Benutzen Sie **continue** um nur gerade Zahlen in der Schleife zu addieren und brechen Sie die Schleife ab, falls die bisher berechnete Summe den fünffachen Wert der Eingabezahl übersteigt.

Lernkontrolle

(1) Welche der folgenden Anweisungen erzeugt einen Fehler?

(a) 2*[1,3] (b) "das"*10 (c) "das"+10

(2) Sei d={'a': 1, 'b': 2} ein dict. Welcher Fehler wird bei dem Aufruf print(d['c']) erzeugt?

(a) KeyError (b) IOError (c) NameError

(3) Mit welcher Formatspezifikation können Sie die Zahl 11/26 als float Zahl mit 8 Stellen nach dem Komma rechtsbündig mithilfe von print ausgeben?

(a) {:>8f} (b) {:>9.8f} (c) {:>9.8l}

Lesen und Schreiben von Dateien

In **Python** lassen sich einfach 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","r+")

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

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

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

# Datei schliessen
d.close()
```

Kopieren von Container-Objekten

copy.py

```
l = [1,2]
k = l           # k und l verweisen auf dasselbe Objekt
k[0] = 99
print k, l


l = [1,2]
k = l.copy()    # k verweist auf eine Kopie des Objekts
                # auf das l verweist
k[0] = 99
print k, l
```

Genauso haben **dict** und **set** eine **copy**-Methode.

Funktionen

Funktionen dienen zur Auslagerung und Systematisierung von Anweisungen, die dann bequem über den Aufruf der Funktion beliebig oft ausgeführt werden können. Die Funktion kann Eingabeargumente erhalten und selbst Objekte zurückgeben.

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

```
def Funktionsname(a,b,...):
        <Anweisungen>
```

Funktionen

function.py

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

result = summe(1,2)
print(result)

# Funktionen sind Objekte wie alles andere in Python
f = summe
print(result - f(1,2))
print(f)
```

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

Bei der Arbeit mit Funktionen muss darauf geachtet werden, dass in Funktionen lokal definierte Variablen nicht global bekannt sind, wie folgendes Beispiel zeigt:

scope.py

```
def function():
    x = 1
    return x

if True:
    y = 2

# Kontrollstrukturen erzeugen keinen lokalen scope
print(y)

# Funktionsdefinitionen hingegen schon - Fehler
print(x)
```

Lambda-Funktionen

Lambda-Funktionen dienen zur Erstellung von anonymen Funktionen, d.h. Funktionen ohne Namen. Speziell bei Nutzung der **map**- oder **filter**-Funktion sind solche **Lambda**-Funktionen hilfreich. 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]
```

and und **or** sind short-circuit Operatoren

In <**Ausdruck1**> **and/or** <**Ausdruck2**> wird **Ausdruck2** nur dann ausgewertet, wenn **Ausdruck1** nicht bereits über den Wahrheitswert des Gesamtausdrucks entschieden hat.

short-circuit

```
def f1(x):
    print('in f1')
    return x

def f2(x):
    print('in f2')
    return x

f1(0) or f2(0)     # gibt 'in f1' und 'in f2' aus
f1(1) or f2(3)     # gibt 'in f1' aus
f1(0) and f2(3)    # gibt 'in f1' aus
f1(3) and f2(3)    # gibt 'in f1' und 'in f2' aus
```

Import von Modulen

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

Folgende Befehle stehen hierzu zur Verfügung:

- **import** <Modulname>
  Verwendung per <Modulname>.<Funktion/Klasse>
- **from** <Modulname> **import** <Funktion/Klasse> [as <Name>]
  Verwendung per <Funktion/Klasse>

Dabei ist <Modulname> der Dateiname einer Quelldatei (ohne **.py**) im gleichen Verzeichnis oder im global Modul-Suchpfad von Python.

Import von Quelldateien

tools.py

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

importSource.py

```
import tools # tools.py liegt im gleichen Verzeichnis
z = tools.summe(2,3)
print(z)

# Alternativen
from tools import summe # Einzelne Funktion
z = summe(2,3)
print(z)

from tools import summe as import_summe
z = import_summe(2,3)
print(z)
```

Aufgaben

(1) Schreiben Sie eine Python-Datei, die eine Funktion enthält, welche zwei gleich lange übergebene Listen elementweise addiert und die Ergebnisliste zurückgibt.
Hinweis: Entweder wird in Ihrer Funktion eine neue Liste erzeugt, oder sie verwenden eine der bereits gegebenen Listen und modifizieren diese.

(2) Schreiben Sie ein Python-Programm, das Ihre Datei aus (1) importiert. Definieren Sie zwei gleich lange Listen und verwenden Sie die importierte Funktion. Geben Sie die Ergebnisliste aus.

Lernkontrolle

(1) def incr(a): a+=2 sei eine Funktion. Angenommen sie definieren b=2 und rufen incr(b) auf. Welchen Wert hat b nun?

(a) 4 (b) 2 (c) Undefiniert

(2) def incr(a): a+=[2] sei eine Funktion. Angenommen sie definieren b=[2] und rufen incr(b) auf. Welchen Wert hat b nun?

(a) [4] (b) [2] (c) [2,2]

(3) Sie haben zufällig zwei Funktionen gleicher Signatur aus zwei verschiedenen Dateien importiert und können in ihrem Code beide nicht explizit unterscheiden. Was passiert bei Aufruf der Funktion?

(a) Ein Fehler wird erzeugt.

(b) Die zuletzt Importierte wird aufgerufen.

(c) Die zuerst Importierte wird aufgerufen.

Zusammenfassung

Bisherige Themen

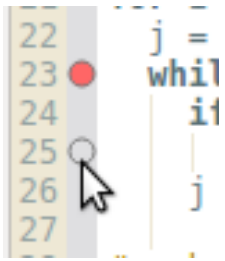
- Grundlagen der Programmiersprache **Python**: Funktionen, Strings, Exceptions
- Importieren von Quelldateien in **Python**

Kommende Themen

- Numerik mit **Python** - das **numpy**-Modul

Debuggen in Pyzo

Ein **Debugger** dient zum Auffinden und Analyiseren von Fehlern in Programmen. Insbesondere kann mit einem Debugger das Programm angehalten und schrittweise ausgeführt werden.



- **Breakpoint** setzen: An der passenden Stelle in die graue Spalte zwischen Zeilennummer und Code klicken
- Programm starten; stoppt automatisch beim Breakpoint
- Verschiedene Icons in der Shell zum „Weiterklicken“:
  - Weiter bis zur nächsten Zeile
  - Nächster Schritt (auch in aufgerufene Funktionen, etc. hinein)
  - Bis zum Ende des aktuellen Bereichs (z.B. `return`)
  - Weiter (bis zum nächsten Breakpoint)
  - Debuggen beenden

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**-Pakets.

Wesentliche Vorteile von NumPy:

- Prägnante Formulierung mathematischer Operationen wie z.B. Matrix-Multiplikation.
- Meist viel höhere Geschwindigkeit als bei Verwendung von Python-Listen und Schleifen.
- Große Sammlung grundlegender mathematischer Algorithmen, z.B. LGS lösen.

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

NumPy

Arrays

```
import numpy as np
# 1d array (= Vektor)
a = np.array([1,2,3,4])
a.shape == (4,)
a.dtype == np.int64

# 2d array (= Matrix)
a = np.array([[1.,2.], [3.,4.], [5.,6.]])
a.shape, a.dtype == (3,2), np.float64

# 3d array (= 3-Tensor)
a = np.array([[[1.,2.],[4,5]],[[1,2],[4,5]]])
a.shape, a.dtype == (2,2,2), np.float

# Achtung, alle Zeilen muessen gleich lang sein!
a = np.array([[1,2],[3],[4]])
a.shape, a.dtype == (3,), object
```

NumPy

Array-Erstellung und -Indizierung (Slicing)

```
import numpy as np

np.ones((4,4), dtype=complex)
np.zeros((3,3,3))
np.empty(5)
np.eye(3)

# leere, aber unterschiedliche Arrays!
np.zeros(0)
np.zeros((0,5))
np.zeros(())

# Indizierung und Slicing
a = np.arange(6).reshape((3,2))
print(a)
a[2,0]              # == 4
a[1,:]              # zweite Zeile
a[:,0]              # erste Spalte
a[1:3,0]            # == [3,5]
```

NumPy

Grundlegende Operationen

```
import numpy as np
a = np.array([1,2,3])
b = np.arange(3)

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

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

# Matrizen
A = np.array([[1,0],[0,1]])
B = np.array([[2,3],[1,4]])
A*B                          # elementweises Produkt
A.dot(B)                     # Matrixprodukt
```

NumPy

Weitere Operationen

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

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

c = np.random.rand(5)
print(c)
c.sort()
print(c)
```

NumPy

Stacking und Splitting von arrays

```
# stacking
a, b = np.arange(3), np.arange(3,6)
np.vstack((a,b))                # == np.array([[0,1,2],[3,4,5]])
np.hstack((a,b))                # == np.array([0,1,2,3,4,5])
np.column_stack((a,b))          # == np.array([[0,3],[1,4],[2,5]])

# splitting
c = np.hstack((a,b))
np.hsplit(c,3)                  # c in 3 Teile trennen
np.hsplit(c,(3,4))              # c nach 3. und 4. Spalte trennen
```

Achtung!

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

Aufgaben

(1) Schreiben Sie ein Python-Programm, in dem folgende Matrix-Vektor Multiplikation mit Matrix A und Vektor x mit Hilfe von NumPy-Arrays vereinfacht wird:

$$A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -2 & 1 \\ 0 & 2 & 5 \end{pmatrix}, \quad x = (4, 1, 2)^T$$

Extrahieren Sie hierzu möglichst geschickt die 2×2 Untermatrix bei Streichung der ersten Spalte und ersten Zeile von A und multiplizieren diese mit dem richtigen Teilvektor von x. Geben Sie das Ergebnis aus.

Lernkontrolle

(1) Sei a=np.array([1,2,3,4]) gegeben. Was ist das Ergebnis der Operation a/=2?

(a) np.array([0.5,1.0,1.5,2.0])

(b) np.array([0,1,1,2])

(c) TypeError

(2) Was ist der Unterschied zwischen range() und arange()?

(a) range() liefert Liste, arange() NumPy array.

(b) Es gibt keinen!

(c) range() liefert iterierbares Range-Objekt, arange() NumPy array.

NumPy

Achtung!

Bei der Arbeit mit **NumPy** kann es schnell zu Fehlern bezüglich der Zuweisung gleicher Daten kommen. Folgende, Fälle müssen unterschieden werden:

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

NumPy

Referenz

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

Kopie

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

NumPy

View

```
a = np.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/b werden veraendert
a                       # == array([0,1,2,100])
```

Achtung!

- Das Kopieren speicherintensiver Objekte (in der Praxis z.b. Gitter mit Millionen von Gitterpunkten) sollte vermieden werden.
- Bei Unsicherheiten lässt sich über **is** feststellen, ob Variablen auf das gleiche Objekt verweisen.

NumPy

Übersicht wichtiger **array** Befehle

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

Achtung!

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

NumPy

Universal functions (**ufuncs**)

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

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

NumPy

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

Matrix class

```
A = np.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 = np.matrix("1.0, 2.0; 3.0, 4.0")
A*B     # Matrixmultiplikation
y = np.matrix("3.0; 2.0")
np.linalg.solve(A,y) # loest lineares Gleichungssystem Ax = y nach x
```

Achtung!

Die Verwendung von **matrix** ist aufgrund der subtilen Unterschiede zu **array** generell nicht empfehlenswert.

NumPy

NumPy ermöglicht neben der indizierung mit Zahlen und slices auch die Indizierung mit Listen von Indices. **Dies erzeugt immer eine Kopie!**

Advanced Indexing

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

j = np.array([[1,2],[6,5]]) # 2-dim Index-Array
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                      # bool array
a[b]                          # a ohne Werte gleich 0
```

NumPy

View vs. Kopie beim Indizieren

```
a = np.eye(5)                   # 5x5 Einheitsmatrix
a[2,3].base is a                # False

a[3,:].base is a                # True
a[:,0].base is a                # True
a[2:4, 1:2].base is a           # True
a[3].base is a                  # True
a.T.base is a                   # True
a.ravel().base is a             # True

a[[3,4,2], :].base is a         # False
a[[0], :].base is a             # False
a[[0]].base is a                # False
```

Regel

view bei **slicing**, **Kopie** bei **Indizierung mit Listen**.

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.

Aufgaben

(1) Schreiben Sie ein Python-Programm, das für ein gegebenes **array** die Summe der Komponenten, den betragsmäßig größten und kleinsten Eintrag sowie deren Index mithilfe von builtin NumPy Funktionen bestimmt und ausgibt.

(2) Schreiben Sie ein Python-Programm, in dem der Code aus (1) in eine Funktion ausgelagert ist, die das entsprechende array als Argument erhält. Testen Sie Ihr Programm an den folgenden Daten:

np.array([0, 2, -4, 5, 3]), np.array([1.5, 3.0, -3.0, -1.5])

Lernkontrolle

(1) Sie möchten zeitweise mit einem Form-veränderten NumPy-Array arbeiten, ohne die Form des Arrays global zu verändern. Womit arbeiten Sie?

(a) Referenz (b) Kopie (c) View

(2) Mit welchem NumPy-Befehl können Sie eine Diagonalmatrix der Dimension n mit Einträgen konstant 2 erzeugen?

(a) np.diag([2],n) (b) 2*np.ones((n,n))

(c) np.diag(np.full(n, 2))

(3) Sie möchten alle Einträge eines Arrays v ungleich null als NumPy array erhalten - wie?

(a) v[v!=0] (b) v[v==0] (c) v!=0

Zusammenfassung

Bisherige Themen

- Grundlagen des Moduls **NumPy**: Arrays, Matrizen, Indizierung, Referenzen, Views und hilfreiche Module

Kommende Themen

- **matplotlib**
- Numerische Mathematik
- Klassen und Vererbung

matplotlib

matplotlib ist eine 2D-Plotting-Bibliothek für Diagramme und wissenschaftliche Visualisierungen.

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

Weitere Informationen und Dokumentation sind zu finden unter `www.matplotlib.org`

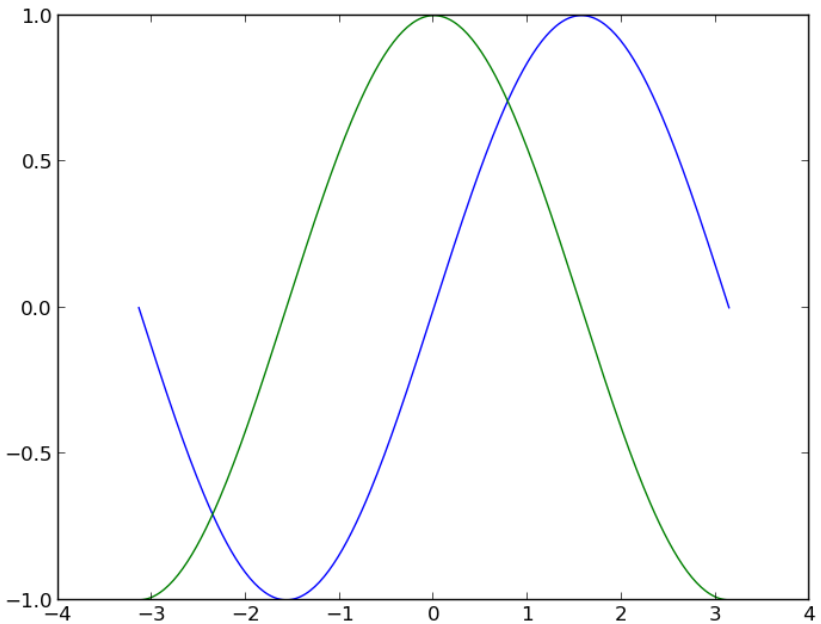
matplotlib

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)
S,C = np.sin(x), np.cos(x)

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

# Erzeuge Ausgabe
plt.show()
```

matplotlib

Ausgabe:



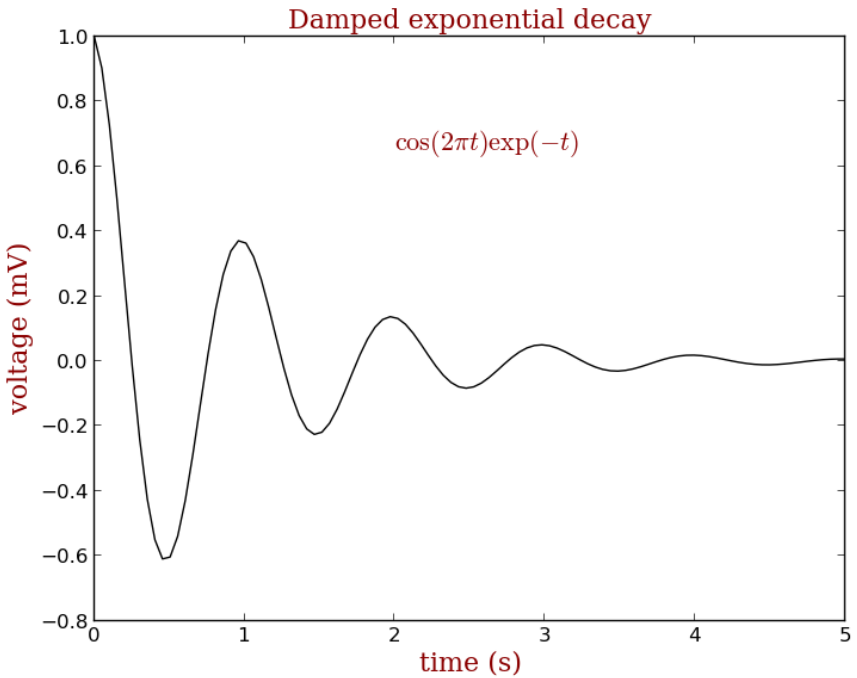
matplotlib

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 **SciPy**-Ökosystem (`www.scipy.org`) bietet neben **numpy** und **matplotlib** noch zahlreiche weitere Bibliotheken und sonstige Ressourcen zu wissenschaftlichem Rechnen mit **Python**.

Weiter wichtige **SciPy**-Bibliotheken:

- **scipy:** Große Sammlung grundlegender numerischer Algorithmen die über das in **numpy** angebotene weit hinausgehen.
- **jupyter:** Interaktive Web-Notebooks für wissenschaftliches Arbeiten mit **Python** und weiteren Sprachen.
- **sympy:** Computer-Algebra mit **Python**.
- **pandas:** Datenanalyse-Paket mit zu **R** vergleichbaren Features.

Aufgaben

(1) Schreiben Sie ein Python-Programm, in dem Sie selbst ein Polynom beliebigen Grades definieren und dieses in einem gewünschten Intervall mithilfe von **matplotlib** darstellen. Beschriften Sie die gegebenen Achsen und den Graphen des Polynoms entsprechend. *Hinweis: Sie können dazu den Graphen mit dem plot-Befehl labeln und mithilfe einer Legende eine Bezeichnung einblenden. Suchen Sie hierzu eine Anleitung im Internet.*

Numerische Mathematik

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 approximativen Lösen von Problemen der Analysis, insbesondere von Differentialgleichungen.

Numerische Mathematik

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.
- Solche Änderungen der Eingabedaten können dabei durch die Problemstellung aber auch die Maschinengenauigkeit bedingt werden.

Numerische Mathematik

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

Numerische Mathematik

Beispiel: Numerische Integration

- **Gegeben:** Riemann integrierbare Funktion $f : [a, b] \to \mathbb{R}$ 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

Numerische Mathematik

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:

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

Numerische Mathematik

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.

Numerische Mathematik

Numerische Analysis

Die Themen der numerischen Analysis umfassen u.a.:

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

Klassen

Eine **Klasse** ist eine Vorlage für gleichartige Objekte. Sie legt fest welche Daten-Attribute und **Methoden** (=Funktionen) **Instanzen** (=Objekte) dieser Klasse 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 ein Grundkonzept der **objektorientierten Programmierung** dar.

Klassen

Syntax zur Definition einer Klasse:

```
class MyClass:
        # Konstruktor
        def _ _init_ _(self, ...):
                <Anweisungen>

        def eine_methode(self, ...):
                <Anweisungen>
         ...
```

Klassen

class1.py

```
# Klassendefinition
class MyClass:
    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!")
print(obj1.type)
obj1.some_function()
obj2 = MyClass("Ich verkaufe diese feinen Lederjacken")
obj2.some_function()
```

Klassen

class2.py

```
class car:
    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()
```

Aufgaben

(1) Schreiben Sie ein Python-Programm, in dem Sie eine Klasse **Hund** implementieren. Diese soll Attribute **alter** und **name** enthalten, die beim Aufruf des Konstruktors initialisiert werden. Schreiben Sie zwei Methoden die jeweils das Alter und den Namen des Hundes zurückgeben. Testen Sie Ihre Klasse an selbst gewählten Beispielen.

(2) Erweitern Sie Ihre Klasse aus (1) um die **bool** Variable **hunger**, die sie bei Instanziierung eines Hund-Objekts automatisch auf **True** setzen. Fügen Sie eine Methode zur Abfrage des Hungers und eine Methode **fuettern** zum Setzen der Variable Hunger auf **False** hinzu.

Klassen

Special (Magic) Members

Mithilfe besonderer Methoden und Attribute - sogenannter **special 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

Special Members

Beispiele für special members sind:

- **_ _ init _ _**: Wird bei Erzeugung einer neuen Klasseninstanz aufgerufen.
- **_ _ str_ _**: Gibt an was **str(obj)** zurückgibt, nützlich für **print(obj)**.
- **_ _ call_ _**: Instanzen einer Klasse wie eine Funktion aufrufen.
- Für Vergleichsoperationen: **_ _ eq_ _**, **_ _ lt_ _**, **_ _ le_ _**, **_ _ gt_ _**, **_ _ ge_ _** usw.
- Für binäre Operationen: **_ _ add_ _**, **_ _ sub_ _**, **_ _ mul_ _**, **_ _ truediv_ _**, **_ _floordiv_ _** usw.

Klassen

magic.py

```
class Gummibaeren:
    def __init__(self, menge):
        self.menge = menge
    def __add__(self, other):
        return Gummibaeren(self.menge + other.menge)
    def __iadd__(self, other):
        self.menge += other.menge
        return self
    def __eq__(self, other):
        return self.menge == other.menge
    def __str__(self):
        if self.menge > 0:
            return '{} Gummibaeren, hmmm!'.format(self.menge)
        else:
            return 'Keine Gummibaeren :((('

meine, deine = Gummibaeren(100), Gummibaeren(79)
unsere = meine+deine
print(unsere, meine != deine)
meine += deine                    # meine = meine.__iadd__(deine)
```

Vererbung

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

- Ist die Klasse **B** abgeleitet von Klasse **A**, so erbt **B** alle Methoden von **A** - sofern **B** diese nicht überschreibt.
- Bei Klassen ähnlicher Struktur und Anwendung kann so viel Programmieraufwand gespart werden. Dabei kann allgemeines Verhalten der Basisklasse in der abgeleiteten Klasse spezialisiert werden.
- Beispiel: **Quadrat** könnte von **Viereck** ableiten und die Methode **area()** durch eine effizientere Implementierung überschreiben.

Vererbung

inherit1.py

```
class BaseClass:
    def __init__(self, msg):
        self._msg = msg

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

class DerivedClass(BaseClass):
    def __init__(self, msg):
        self._msg = msg.upper()

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

base.report()
deriv.report()
```

Vererbung

inherit2.py

```
class BaseClass:
    pass

class DerivedClass(BaseClass):
    pass

issubclass(DerivedClass, BaseClass)   # True

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

Aufgaben

(1) Erweitern Sie Ihr vorheriges Python-Programm um eine Klasse Welpe, die von der Klasse Hund erbt. Fügen Sie der neu definierten Klasse eine Methode eigener Wahl, die charakteristisch für Welpen ist, hinzu und testen Sie anhand eines Welpen selbst gewählten Namens und Alters.

Zusammenfassung

Bisherige Themen

- Grundlegender Umgang mit **matplotlib**
- Numerische Mathematik
- Klassen und Vererbung

Kommende Themen

- Ein paar fortgeschrittene Themen.

Iteratoren

Ein **Iterator** bezeichnet einen Zeiger, mit dem man die Elemente eines **iterables** (z.B. **list**, **dict**, **set**) durchlaufen kann. Iteratoren können als argument einer **for**-Schleife verwendet werden.

Mithilfe der Funktion **iter** erhält man zu einem iterierbaren Objekts einen Iterator.

iter

```
it = iter([1,4,9])
print(next(it))
for i in it:
    print('in for loop', i)
next(it)  # StopIteration Exception
```

Generatoren

Mithilfe von **Generatorfunktionen** lassen sich leicht neue `iterables` definieren:

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

Generatorausdrücke

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

absolute_values == list(absolute_values_gen)
```

List Comprehensions

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

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)
```

Aufgaben

(1) Schreiben Sie ein Python-Programm, in dem Sie zwei gleich lange Listen mit Hilfe einer list comprehension elementweise multiplizieren.

(2) Erweitern Sie Ihr Programm aus (1), sodass alle negativen Einträge der jeweiligen Listen vor der Multiplikation durch eine 0 ersetzt werden.

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 oder dem Aufruf der Funktion **help( )** 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

In **Python** lassen sich sogenannte **Dekoratoren** 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 twice(obj):
    def wrapper(x):
        return obj(obj(x))
    return wrapper

@twice
def function(x):
    return x**2

function(4)
```

Dekoratoren

deco2.py

```
class call_counter:
    def __init__(self, func):
        self.count = 0
        self.func = func
    def __call__(self, *args, **kwargs):
        self.count += 1
        print('Funktionsaufruf-Nummer:', self.count)
        return self.func(*args, **kwargs)

@call_counter
def f(a, b):
    print(a + b, a * b)

f(2, 3)
f(1, 4)
```

Monkey Patching

- **Python** bietet die Möglichkeit zur Laufzeit Funktionen in Modulen oder Klassen zu ersetzen.
- Sollte sehr sparsam eingesetzt werden!

monkey.py

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

foo = Foo()
foo.run()

def run_bar(self):
    print('bar')
Foo.run = run_bar

foo.run()
```

ENDE