



Westfälische
Wilhelms-Universität
Münster

Einführung in die Programmierung zur Numerik mit Python

Tobias Leibner, Stephan Rave



Kursinhalte

Montag:

- ▶ Organisatorisches
- ▶ Warum Python?
- ▶ Arbeiten mit der Shell
- ▶ Erste Python-Programme

Dienstag:

- ▶ Anweisungen und Ausdrücke
- ▶ Objekte und Methoden
- ▶ Grundlegende Datentypen
- ▶ Container-Datentypen
- ▶ Iteration über Container mit der `for`-Schleife



Kursinhalte (fortges.)

Mittwoch:

- ▶ Funktionen
- ▶ Module

Donnerstag:

- ▶ Effiziente numerische Algorithmen mit NumPy

Freitag:

- ▶ Kurze Einführung in `matplotlib`
- ▶ Ausblick auf weitere Sprach-Features
- ▶ Nützliche Python-Pakete



Tag 1



Organisatorisches

- ▶ Teilnamebescheinigung:
Anwesenheit an allen Kurstagen.
- ▶ Logische Grundlagen und Programmierung (1-Fach Bachelor, PO 2014):
Anwesenheit an allen Kurstagen & erfolgreiches Bearbeiten einer Hausaufgabe
- ▶ Pro Nachmittag ein Übungszettel
- ▶ Arbeit in Zweiergruppen
- ▶ Funktioniert der Login?
- ▶ Mittagspause

Programmieren in der Numerik - Ein Beispiel

Problem

Zu lösen ist auf einem beschränkten Gebiet $\Omega \in \mathbb{R}^2$ mit polygonalem Rand die Poisson-Gleichung für ein gegebenes $f \in L^2(\Omega)$:

$$-\Delta u = f, \quad u|_{\partial\Omega} \equiv 0$$

Schwache Formulierung

Wir multiplizieren die Gleichung mit einer Testfunktion $\varphi \in C_0^\infty(\Omega)$ und integrieren:

$$\int_{\Omega} -\Delta u \cdot \varphi \, d\mu = \int_{\Omega} f \cdot \varphi \, d\mu.$$

Angenommen u ist hinreichend regulär, dann liefert partielle Integration

$$b(u, \varphi) := \int_{\Omega} \nabla u \cdot \nabla \varphi \, d\mu = \int_{\Omega} f \cdot \varphi \, d\mu =: l(\varphi).$$

Aufgrund der **Poincaré-Ungleichung** ist die Bilinearform b auf dem Hilbert-Raum $H_0^1(\Omega)$ uniform positiv definit, das lineare Funktional l aufgrund der Beschränktheit von f stetig.

Programmieren in der Numerik - Ein Beispiel

Schwache Formulierung (fortges.)

Mit dem **Rieszschen-Darstellungssatz** existiert daher eine **schwache Lösung** $u \in H_0^1(\Omega)$ mit

$$b(u, \varphi) = l(\varphi) \quad \forall \varphi \in H_0^1(\Omega).$$

Finite-Elemente-Diskretisierung

1. Wähle ein geeignetes Dreiecksgitter \mathcal{T}_h , welches Ω überdeckt.
2. Wähle endlichdimensionalen Finite-Elemente-Raum $S_{0,h}^1 \subset H_0^1(\Omega)$ aus stückweise linearen Funktionen mit Nullrandwerten.

Da b auch auf $S_{0,h}^1$ ein Skalarprodukt definiert, existiert die eindeutige **Finite-Elemente-Lösung** $u_h \in S_{0,h}^1$ mit

$$b(u_h, \varphi_h) = l(\varphi_h) \quad \forall \varphi_h \in S_{0,h}^1.$$

Laut dem **Céa-Lemma** ist u_h eine Quasi-Bestapproximation von u in $S_{0,h}^1$.

Programmieren in der Numerik - Ein Beispiel

Wie rechnet man das aus??

1. Wir müssen ein beliebig feines Gitter \mathcal{T}_h mit sehr vielen Elementen und geeigneten Geometrie-Eigenschaften konstruieren.
2. Für eine Basis $\Phi := \{\varphi_1, \dots, \varphi_N\}$ von $S_{0,h}^1$, sodass $u_h = \sum_{i=1}^N u_i \varphi_i$, suchen wir die Unbekannten $u_i \in \mathbb{R}$ mit

$$\sum_{i=1}^N u_i b(\varphi_i, \varphi_j) = l(\varphi_j) \quad \forall 1 \leq j \leq N.$$

Wir müssen also die Matrix der Bilinearform b und den Vektor des Funktionals l bezüglich Φ berechnen. Dafür sind sehr viele Integrale zu bestimmen.

3. Wir müssen das resultierende lineare Gleichungssystem mit sehr vielen Unbekannten lösen (**Numerische Lineare Algebra**).

Warum Python?

Python ist

- ▶ eine frei verfügbare Programmiersprache für alle gängigen Betriebssysteme.
- ▶ eine moderne, ausdrucksstarke Sprache mit klaren Designprinzipien.
- ▶ leicht zu erlernen.
- ▶ Sprache der Wahl für zahlreiche Projekte in den Naturwissenschaften, Data Sciences und Machine Learning.
- ▶ universell einsetzbar mit über 100.000 verfügbaren Erweiterungspaketen.

Achtung: Python ist in den inkompatiblen Versionen 2 und 3 verbreitet. Wir lernen Python 3.

Die Shell

- ▶ Die Shell ist ein textbasiertes, Programm mit dem der Benutzer Dateioperationen ausführen und Programme starten kann.
- ▶ Unter grafischen Oberflächen wird zur Interaktion mit der Shell ein **Terminal**-Programm verwendet, welches sich um die Ein- und Ausgabe kümmert.
- ▶ Die gebräuchlichste Shell unter Linux/macOS ist die **bash**. Weitere Shells sind z.B. **zsh** und **fish**.
Unter Windows stellt die **Eingabeaufforderung** (`command.com`) eine rudimentäre Shell zur Verfügung. Verbreitet ist zudem die **Powershell**.

Die Shell: Verzeichnisse

- ▶ Jede Datei hat unter Linux/macOS/Android/iOS einen eindeutigen **Pfad** der Form

`/Verzeichnis1/Verzeichnis2/.../VerzeichnisN/DateiName`

dabei entsprechen `Verzeichnis1, ..., VerzeichnisN` bei einem grafischen Dateimanager ineinandergeschachtelten **Ordern**.

- ▶ Groß- und Kleinschreibung ist relevant. `/foo` und `/Foo` sind verschiedene Pfade.

- ▶ Das Verzeichnis

`/`

wird als das Wurzelverzeichnis oder **Rootverzeichnis** bezeichnet.

- ▶ Der Befehl

`ls <Pfad>`

zeigt alle Dateien an, die im durch `Pfad` benannten Verzeichnis enthalten sind.

Die Shell: Arbeitsverzeichnis

- ▶ Es ist stets ein aktuelles **Arbeitsverzeichnis** gesetzt. Der Befehl

```
pwd
```

gibt dieses aus.

- ▶ Der Befehl

```
ls
```

zeigt die Dateien im Arbeitsverzeichnis an.

- ▶ Der Befehl

```
cd <Pfad>
```

wechselt das Arbeitsverzeichnis zu Pfad.

- ▶ Der Befehl

```
cd
```

setzt das Arbeitsverzeichnis auf das **Homeverzeichnis** des Nutzers. Auf den Rechnern des Fachbereichs lautet dieses `/home/<ZIV_Nutzerkennung>`.

Die Shell: Relative Pfade

- ▶ Pfade der Form

`Verzeichnis1/Verzeichnis2/.../VerzeichnisN/DateiName`

sind relativ zum aktuellen Arbeitsverzeichnis. D.h., ist das Arbeitsverzeichnis

`/a/b/c`

so ist der Pfad

`d/e/f`

äquivalent zu

`/a/b/c/d/e/f`

- ▶ An allen Stellen, an denen ein Pfad erwartet wird, kann auch ein relativer Pfad verwendet werden. Insbesondere gilt dies für Shell-Befehle wie `cd` oder `ls`.

Die Shell: Die wichtigsten Befehle

```
cd <Pfad>                # Setze Arbeitsverzeichnis auf <Pfad>.

ls <Pfad>                # Zeige Dateien in <Pfad> an.
ls -a <Pfad>             # Zeige auch mit . beginnende Dateien an.
ls -l <Pfad>             # Zeige zusätzliche Informationen an.

mkdir <Pfad>              # Erstelle Verzeichnis <Pfad>.
                          # (Übergeordnete Verzeichnisse müssen existieren.)

cp <PfadA> <PfadB>        # Erzeuge Kopie von Datei <PfadA> in <PfadB>.
                          # Falls <PfadB> Verzeichnis, erzeuge Kopie gleichen Namens
                          # in <PfadB>.

mv <PfadA> <PfadB>        # Wie cp, aber verschiebe die Datei.

rm <Pfad>                 # Lösche Datei <Pfad>.
rmdir <Pfad>              # Lösche leeres Verzeichnis <Pfad>.
rm -r <Pfad>              # Lösche rekursiv Verzeichnis <Pfad> mit allen Dateien und
                          # Unterverzeichnissen.

cat <Pfad>                # Gebe Inhalt der Datei <Pfad> aus.
less <Pfad>               # Zeige Inhalt der Datei <Pfad> interaktiv an.
                          # Beenden mit der Taste "q".
```

Die Shell: Programme ausführen

- ▶ Alle Linux-Programme liegen ebenfalls als Dateien im Dateisystem. Sie können mit der Shell direkt ausgeführt werden, z.B.

```
/usr/bin/kate <Pfad>
```

öffnet die Datei <Pfad> mit dem Editor `kate`.

- ▶ Die Shell sucht automatisch in einigen Verzeichnissen, z.B. `/usr/bin`, nach Programmen. Daher können wir auch einfach

```
kate <Pfad>
```

schreiben.

- ▶ Durch Anhängen des `&`-Zeichens kann ein Programm als **Hintergrundjob** ausgeführt werden, während mit der Shell weitergearbeitet werden kann.

```
kate <Pfad> &
```



Die Shell: Vorteile

- ▶ Shell-Befehle sind einfacher zu beschreiben als die Nutzung grafischer Anwendungen.
- ▶ Standardisiert (POSIX-Standard).
- ▶ Programmierbar.
- ▶ Komplexe Aufgaben leichter umsetzbar als mit grafischen Dateimanagern.
- ▶ Terminal-basierte Programme sind leichter zu programmieren als Anwendungen mit grafischer Benutzeroberfläche.

Die Shell: Setup

1. Legen Sie mit der Shell die folgende Verzeichnishierarchie an:

```
/u/<ZIV_Nutzerkennung>/PythonKurs/Tag1
```

- ▶ Nutzerkennung heraus finden: `whoami`
- ▶ `cd /u/<ZIV_Nutzerkennung>`
- ▶ `mkdir PythonKurs`
- ▶ `cd PythonKurs`
- ▶ `mkdir Tag1`

2. Richten Sie `kate` so ein, dass als Einrückungsmodus `Python` mit 4 Leerzeichen verwendet wird.
3. Fügen Sie der Datei `/home/<ZIV_Nutzerkennung>/.bashrc` am Ende die Zeile

```
export PATH=$HOME/.local/bin:$PATH
```

mit `kate` hinzu.
4. Starten Sie das Terminal neu.

Hallo Welt

1. Legen Sie mit kate die folgende Datei in
/u/<ZIV_Nutzerkennung>/PythonKurs/Tag1 an:

hello.py

```
1 name = input('Ihr Name? ')
2 laenge = len(name)
3 if laenge == 0:
4     name = 'Namenloser'
5
6 print('Guten Tag, ' + name + '!')
7 print('Ihr Name ist ' + str(laenge) + ' Zeichen lang.')
```

2. Führen Sie das Programm mit den Befehlen

```
cd /u/<ZIV_Nutzerkennung>/PythonKurs/Tag1
python3 hello.py
```

im Terminal aus.

3. Was passiert, wenn Sie `laenge` statt `str(laenge)` in Zeile 7 schreiben?



Hello Welt im visuellen Debugger

1. Installieren Sie den visuellen Debugger pvdb:

```
pip3 install pvdb
```

2. Führen Sie das Programm mit pvdb aus:

```
pvdb hello.py
```

Ein weiteres Beispiel

1. Führen Sie das folgende Programm aus (auch mit `pdb`):

summe.py

```
1 n = int(input('n = '))
2 i = 1
3 summe = 0
4 while i < n + 1:
5     summe = summe + i
6     i = i + 1
7 print('Die Summe der Zahlen von 0 bis', n, 'ist', summe)
```

2. Berechnen Sie die gleiche Summe, indem Sie die Zählvariable `i` mit `i = n` initialisieren und dann rückwärts bis `1` zählen.
3. Berechnen Sie die Summe zusätzlich mit der Gaußformel

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}.$$

4. Berechnen Sie die Summe der ersten n Quadratzahlen.

FizzBuzz

Es soll für die Zahlen $1 \leq i \leq 100$ folgende Bildschirmausgabe gemacht werden:

- ▶ *'Fizz'*, falls i durch 3, jedoch nicht durch 5 teilbar ist,
- ▶ *'Buzz'*, falls i durch 5, jedoch nicht durch 3 teilbar ist,
- ▶ *'FizzBuzz'*, falls i durch 3 und 5 teilbar ist,
- ▶ i sonst.

1. Führen Sie das folgende Programm aus:

fizzbuzz.py

```
1 i = 1
2 while i < 100 + 1:
3     if i % (3 * 5) == 0:    # '%' ist der 'modulo' Operator
4         print('FizzBuzz')
5     elif i % 3 == 0:
6         print('Fizz')
7     elif i % 5 == 0:
8         print('Buzz')
9     else:
10        print(i)
11    i = i + 1
```

FizzBuzz (fortges.)

2. Vervollständigen Sie folgende alternative Implementierung:

fizzbuzz2.py

```
1 i = 1
2 while i < 100 + 1:
3     treffer = False
4     if i % 3 == 0:
5         print('Fizz', end='') # keine neue Zeile nach 'Fizz'
6         treffer = True
7     if i % 5 == 0:
8         ...
9     if not treffer:
10        ...
11    else:
12        print() # neue Zeile
13    i = i + 1
```

3. Erweitern Sie das Programm, sodass zusätzlich *'Peng'* ausgegeben wird, falls *i* durch 7 teilbar ist.



Tag 2

Warmup

Das Heron-Iterationsverfahren zur Berechnung der Quadratwurzel von a ist gegeben durch

$$x_1 = 1 \quad \text{und} \quad x_{n+1} = \frac{1}{2} \cdot \left(x_n + \frac{a}{x_n}\right).$$

1. Speichern Sie das folgende Programm in `/u/<ZIV_Nutzerkennung>/PythonKurs/Tag2` und führen Sie es aus:

heron.py

```
1 x = 1
2 i = 0
3 while i < 3:
4     x = 0.5 * (x + 2/x)
5     i = i + 1
6 print(x, x*x - 2)
```

```
cd /u/<ZIV_Nutzerkennung>/PythonKurs/Tag2
python3 heron.py
```

2. Geben Sie die aktuelle Approximation und das Residuum in jedem Iterationsschritt aus.
3. Nehmen sie die Zahl a und die Anzahl der Iterationen mit `input` vom Benutzer entgegen.

Anatomie eines Python-Programms

Ein Python-Programm ist eine Aneinanderreihung von **Anweisungen**, die nacheinander (von oben nach unten) abgearbeitet werden.

Anweisungen enthalten **Ausdrücke**, die zu **Objekten** ausgewertet werden, **Namen** die auf Objekte verweisen, sowie **Schlüsselwörter**, die Anweisungen und Ausdrücke strukturieren.

Beispiele:

- ▶ `if`, `else`, `elif`, `while` sind Schlüsselwörter.
- ▶ `1 + 1`, `print('foo')`, `a % 3 == 0` sind Ausdrücke.
- ▶ Beispiele für Anweisungen sind

```
a = a + 1
```

oder

```
if a == 0:  
    print('null')  
else:  
    print('nicht null')
```

Python-Anatomie: Zuweisung und Auswertung

Auswertungs- und Zuweisungs-Anweisung

```
<Name> = <Ausdruck>
```

Auswertungs-Anweisung

```
<Ausdruck>
```

Beispiele:

```
a = 1 + 1           # Auswertung und Zuweisung an a
a + 1              # Auswertung (kein Effekt)
print('Foo')      # Auswertung ('Foo' wird ausgegeben)
a = print('Foo')   # Auswertung und Zuweisung
print(a)           # Auswertung ('None' wird ausgegeben)
```

Achtung: Anders als in der Mathematik kann die Auswertung von Ausdrücken *Nebeneffekte* haben (z.B. Ausgabe der Zeichenkette Foo).

Python-Anatomie: if-Anweisung

if-Anweisung

```
1 if <Ausdruck1>:  
2     <Block1>  
3 elif <Ausdruck2>:  
4     <Block2>  
5     ...  
6 elif <AusdruckN>:  
7     <BlockN>  
8 else:  
9     <Block0>
```

Ausführung:

1. Werte `Ausdruck1` aus. Wenn wahr, führe `Block1` aus. Ende der Anweisung.
 2. Ansonsten, werte `Ausdruck2` aus. Wenn wahr, führe `Block2` aus. Ende der Anweisung.
 3. usw.
 4. Falls kein Ausdruck als wahr ausgewertet wird, führe `Block0` aus.
- ▶ Die `elif`- und `else`-Teile der Anweisung sind optional.

Python-Anatomie: if-Anweisung

- ▶ Ein *Block* ist eine Folge eingerückter Programmzeilen. Der Block endet bei der ersten nicht-leeren Programmzeile mit geringerer Einrückung.

bloecke.py

```
1 x = 3 #
2 if x == 0: #
3     x = x + 1 # | Block 1
4     y = x # |
5     if y == 2: # |
6         print('Foo') # | | Block 2
7         if y > x: # | |
8             x = y # | | | Block 3
9         print('Bar') # | |
10    x = 42 # |
11 else: #
12 print('Nicht null') # | Block 4
```

Python-Anatomie: if-Anweisung

- ▶ Die folgenden Programme sind äquivalent:

elif.py

```
1 a = 42
2 if a == 1:
3     print('Eins')
4 elif a == 42:
5     print('Zweiundvierzig')
6 else:
7     print('Sonstwas')
```

noelif.py

```
1 a = 42
2 if a == 1:
3     print('Eins')
4 else:
5     if a == 42:
6         print('Zweiundvierzig')
7     else:
8         print('Sonstwas')
```

Python-Anatomie: while-Anweisung

while-Anweisung

```
1 while <Ausdruck>:  
2     <Block>
```

Ausführung:

1. Werte `Ausdruck` aus. Wenn falsch, Ende der Anweisung.
2. Führe `Block` aus.
3. Springe zu 1.

fac.py

```
1 fac = 1  
2 i = 1  
3 while True:  
4     fac = fac * i  
5     print(fac)  
6     i = i + 1
```

- ▶ Ein Python-Programm kann mit der Tastenkombination `Strg+C` abgebrochen werden.

Python-Anatomie: Ausdrücke

Wir haben bisher folgende Ausdrücke kennengelernt:

► Literale

```
'<Zeichenkette>'  
"<Zeichenkette>"  
<Ziffernfolge>
```

Zeichenketten-Literale werden zu `str`-Objekten ausgewertet, die die entsprechende Zeichenkette enthalten.

Ziffernfolgen werden zu `int`-Objekten ausgewertet, die die entsprechende Zahl enthalten.

► Namen

```
<Name>
```

Namen werden zu den Objekten ausgewertet, auf die die jeweiligen Namen verweisen.

Eine Liste aller **vordefinierten Namen** kann wie folgt ausgegeben werden:

```
print(dir(__builtins__))
```

Python-Anatomie: Ausdrücke (fortges.)

► Arithmetische Operatoren

```
<Ausdruck1> + <Ausdruck2>    # Addition  
<Ausdruck1> - <Ausdruck2>    # Subtraktion  
<Ausdruck1> * <Ausdruck2>    # Multiplikation  
<Ausdruck1> / <Ausdruck2>    # Division  
<Ausdruck1> // <Ausdruck2>   # Ganzzahl-Division mit Abrunden  
<Ausdruck1> % <Ausdruck2>   # Modulo
```

Es werden Ausdruck1 und Ausdruck2 ausgewertet. Die Ergebnisse werden mit dem jeweiligen Operator verknüpft.

► Vergleichs-Operatoren

```
<Ausdruck1> == <Ausdruck2>   # Gleichheit  
<Ausdruck1> != <Ausdruck2>   # Ungleichheit  
<Ausdruck1> < <Ausdruck2>    # Kleiner  
<Ausdruck1> <= <Ausdruck2>  # Kleiner oder gleich  
<Ausdruck1> > <Ausdruck2>   # Größer  
<Ausdruck1> >= <Ausdruck2>  # Größer oder gleich
```

Es werden Ausdruck1 und Ausdruck2 ausgewertet. Die Ergebnisse werden mit dem jeweiligen Operator verglichen. Ergebnis der Auswertung ist **True** oder **False**.

Python-Anatomie: Ausdrücke (fortges.)

► Negation

- <Ausdruck>

Es wird `Ausdruck` ausgewertet. Ergebnis ist der negative Wert dieser Auswertung.

► Funktionsaufruf

<Name>(<Ausdruck1>, <Ausdruck2>, ..., <AusdruckN>)

Es werden `Ausdruck1` bis `AusdruckN` ausgewertet. Dann wird die von `Name` bezeichnete Funktion mit den Ergebnissen der Auswertungen als Argumente ausgeführt. Ergebnis des Funktionsaufrufs ist der Rückgabewert der Funktion.

- Viele Funktionen haben keinen sinnvollen Rückgabewert. Diese liefern den Wert `None` zurück.
- Funktionen können wie alle anderen Objekten neuen Namen zugewiesen werden:

```
xprint.py
```

```
1 x = print  
2 x('Hallo!')
```

Python-Anatomie: Erlaubte Namen

Für Namen sind in Python insbesondere folgende Regeln zu beachten:

- ▶ Namen dürfen nicht mit einer Zahl beginnen.
- ▶ Schlüsselwörter dürfen nicht als Namen verwendet werden.
- ▶ Die Zeichen - und . dürfen nicht verwendet werden.
- ▶ Unterschiedliche Groß-/Kleinschreibung führt zu verschiedenen Namen.

Beispiele:

```
foo, Foo, f0o           # OK, alle verschieden
foo3, fo3o             # OK
3foo, lambda, global   # Verboten!
foo_bar, _foo_bar      # OK
foo-bar, foo-bar, foo.bar # Verboten!
```

Python Schlüsselwörter

False, None, True, and, as, assert, break, class, continue, def, del,
elif, else, except, finally, for, from, global, if, import, in, is,
lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield

Der kleine Unterschied

1. Führen Sie das folgende Programm aus:

fac2.py

```
1 fac = 1
2 i = 1
3 while i < 200:
4     fac = fac * i
5     print(fac)
6     i = i + 1
7 print(type(fac))
```

2. Was passiert, wenn Sie `fac = 1` durch `fac = 1.` ersetzen?
3. Was passiert, wenn Sie stattdessen `i = 1` durch `i = 1.` ersetzen?

Python-Anatomie: Datentypen

▶ `bool` (**Wahrheitswert, immutable**)

Es gibt genau zwei Objekte vom Typ `bool`,

`True`, `False`

Mit `bool(x)` kann jedem Objekt `x` ein Wahrheitswert zugeordnet werden.

▶ `NoneType` (**Kein Wert, immutable**)

Es gibt genau ein Objekt vom Typ `NoneType`, nämlich `None`.

▶ `int` (**Integer, immutable**)

Speichert ganze Zahlen beliebiger Größe. Konstruktion über Integer-Literale, z.B.

`1`, `2`, `-1`, `0x0A3`, `0b10110`

oder über den `int()`-Konstruktor, z.B.

```
int('1')           # 1
int(1.3)           # 1 - es wird immer abgerundet
int(False)         # 0
int(True)          # 1
```

Python-Anatomie: Datentypen (fortges.)

► float (Fließkommazahl, immutable)

Speichert reelle Zahlen in wissenschaftlicher Notation $a \cdot 10^b$. Konstruktion über Float-Literale, z.B.

```
1., 1.03, .03, 1e10, 0.4e-3
```

oder über den `float()`-Konstruktor, z.B.

```
float(1)           # 1.  
float('1')        # 1.  
float('1.3e2')     # 130.
```

Die Genauigkeit von `float` hängt vom Computer ab. I.d.R. werden ca. 17 *Nachkommastellen der Mantisse und Exponenten bis 308* gespeichert.

Python-Anatomie: Datentypen (fortges.)

► **str (String, immutable, indizierbar, iterierbar)**

Speichert Zeichenketten beliebiger Länge. Konstruktion über String-Literale, z.B.

```
'Ein String', "Ein String"  
'Ein "String"', 'Ein "String"'  
'Ein\nString' # \n steht für das "Neue-Zeile"-Zeichen  
'Ein\'String\'' # \' steht für das "'-Zeichen  
'Ein\\String' # \\ steht für das "\"-Zeichen
```

Mittels `str(x)` kann aus jedem Objekt eine String-Darstellung erzeugt werden.

Listen

1. Führen Sie das folgende Programm aus:

fibolist.py

```
1 fibs = [1, 1]
2 i = 2
3 while i < 10:
4     fibs.append(fibs[i-2] + fibs[i-1])
5     i = i + 1
6 print(fibs)
```

2. Listen können auch mit negativen Zahlen indiziert werden. Dabei entspricht `l[-1]` dem letzten Element der Liste `l[-2]` dem vorletzten Element, usw. Modifizieren Sie das Programm, sodass die Zählvariable `i` nicht mehr in der Indizierung von `fibs` auftritt.
3. Legen Sie zusätzlich eine Liste der Quotienten der benachbarten Fibonaccizahlen an.

Listen und for-Schleifen

1. Führen Sie das folgende Programm aus:

```
prod.py
```

```
1 l = [4, 6, 3, 2]
2 prod = 1
3 for x in l:
4     prod = prod * x
5 print(prod)
```

2. Berechnen Sie zusätzlich die Summe aller Elemente von `l`.
3. Geben Sie auch das kleinste und das größte Element von `l` aus.

Listen und for-Schleifen (fortges.)

1. Führen Sie das folgende Programm aus:

bubble.py

```
1 l = [5, 2, 4, 3, 2]
2 print(l)
3 for i in range(len(l)):
4     for j in range(len(l) - 1):
5         if l[j] > l[j+1]:
6             x = l[j]
7             l[j] = l[j+1]
8             l[j+1] = x
9     print(l)
```

2. Sortieren Sie die Liste in absteigender Reihenfolge.
3. Optimieren Sie den Algorithmus, indem Sie anstelle der äußeren `for`-Schleife ein schärferes Abbruchkriterium wählen.



Python-Anatomie: Objekte

*Ein Python-Objekt enthält **Daten** einer bestimmten Struktur, die vom **Typ** (auch **Klasse**) des Objekts vorgegeben werden. Der Typ des Objekts bestimmt die verfügbaren **Methoden** des Objekts und das Verhalten unter Verknüpfung mit Operatoren.*

*Methoden operieren auf den Daten des Objekts und können diese verändern. Manche Python-Objekte sind unveränderlich (**immutable**).*

Python-Anatomie: Datentypen (fortges.)

► list (Liste, indizierbar, iterierbar)

Speichert geordnete Listen von **Objekt-Referenzen** beliebiger Länge. Konstruktion über Listen-Literale, z.B.

```
[] # Leere Liste
[1, 2, 3] # Liste mit drei Elementen
[1, len('asdf'), 'a', [1, 2]] # Listen-Literale dürfen beliebige Ausdrücke
# als Elemente enthalten
```

Mittels `list(x)` kann aus den Elementen eines iterierbaren Objekts `x` eine Liste erzeugt werden, z.B.

```
list('foo') # -> ['f', 'o', 'o']
```

Die Länge einer Liste `l` erhalten wir mit `len(l)`. Listen können indiziert und zerschnitten werden ('slicing'):

```
l = [1,9,7,2,6]
l[0] # -> 1 (erstes Element)
l[-1] # -> 6 (letztes Element)
l[-2] # -> 2 (vorletztes Element)
l[:3] # -> [1,9,7] (erste 3 Elemente)
l[2:] # -> [7,2,6] (erste 2 Elem. auslassen)
```

Python-Anatomie: Datentypen (fortges.)

► list (fortges.)

Die nützlichsten Methoden von list:

(Ergebnisse jeweils für `l = [1,9,7,2,6]`.)

```
l.append(42)           # -> l == [1,9,7,2,6,42]
                       #   Anhängen eines Elements

l.extend([2,3,5])     # -> l == [1,9,7,2,6,2,3,5]
                       #   Anhängen einer anderen Liste

l.pop()               # -> 6, l == [1,9,7,2]
                       #   entferne letztes Element und gebe es zurück

l.insert(0, 'x')      # -> l == ['x',1,9,7,2,6]
                       #   füge Element an gegebenen Index ein

l.sort()              # -> l == [1,2,6,7,9]
                       #   sortiere die Liste

l.copy()              #   erzeuge Kopie der Liste
```

Python-Anatomie: Datentypen (fortges.)

► str (String, immutable, indizierbar, iterierbar)

str hat viele nützliche Methoden, z.B.

```
'hallo'.upper()           # -> 'HALLO'  
' hallo '.strip()        # -> 'hallo'  
'Eins zwei drei'.split() # -> ['Eins', 'zwei', 'drei']  
'Eins,zwei,drei'.split(',') # -> ['Eins', 'zwei', 'drei']  
'\n'.join(['Zeile1', 'Zeile2', 'Zeile3']) # -> 'Zeile1\nZeile2\nZeile3'
```

► Beispiel:

cases.py

```
1 word = 'gRosSschReiBung'  
2 print(word.lower()[0].upper() + word.lower()[1:])
```

Python-Anatomie: for-Anweisung

for-Anweisung

```
for <Name> in <Ausdruck>:  
    <Block>
```

Ausführung:

1. Werte `Ausdruck` zu Objekt `o` aus.
2. Falls `o` keine Elemente enthält: Ende der Anweisung.
3. Weise `Name` das erste Element von `o` zu.
4. Führe `Block` aus.
5. Falls `o` kein weiteres Element enthält: Ende der Anweisung.
6. Weise `Name` das nächste Element von `o` zu.
7. Springe zu 4.

Achtung: `o` muss ein iterierbares Objekt sein. Ansonsten Fehler.

Identität und Gleichheit

1. Führen Sie das folgende Programm aus:

listcopy.py

```
1 a = 1
2 a2 = a
3 a = a + 1
4 print(a, a2)
5 print(a == a2, a is a2)
6
7 l = [0, 1, 2] * 3 + ['a', 3.5]
8 l2 = l
9 print(l, l2)
10 print(l == l2, l is l2)
11
12 l[3] = 99
13 print(l, l2)
14 print(l == l2, l is l2)
```

2. Ersetzen Sie Zeile 8 durch `l2 = l.copy()`. Welches Verhalten erwarten Sie?

Identität und Gleichheit (fortges.)

1. Führen Sie das folgende Programm aus:

```
bignum.py
```

```
1 x = 1.  
2 y = x + 1.  
3 x = x + 1.  
4 print(type(x))  
5 print(x == y, x is y)
```

2. Was passiert, wenn Sie `1` anstelle von `1.` verwenden?
3. Was passiert, wenn Sie Zeile 1 durch `x = 999` ersetzen?

Python-Anatomie: Objekte

Ein Python-Objekt enthält **Daten** einer bestimmten Struktur, die vom **Typ** (auch **Klasse**) des Objekts vorgegeben werden. Der Typ des Objekts bestimmt die verfügbaren **Methoden** des Objekts und das Verhalten unter Verknüpfung mit Operatoren.

Methoden operieren auf den Daten des Objekts und können diese verändern. Manche Python-Objekte sind unveränderlich (**immutable**).

Zwei Objekte vom selben Typ, die die selben Daten enthalten, sind **gleich** jedoch nicht **identisch**.

- ▶ `a == b` prüft auf Gleichheit.
- ▶ `a is b` prüft auf Identität.
- ▶ `print(type(a))` gibt den Typ von `a` aus.

Faustregel: Verwende `is` / `is not` in der Regel nur in der Form `'x is None'` bzw. `'x is not None'` (nicht zum Vergleich von Zahlen!).

Python-Anatomie: Zuweisung an Container-Elemente

Auswertungs-und-Element-Zuweisungs-Anweisung

```
<Name>[<Ausdruck1>] = <Ausdruck2>
```

Ausführung:

1. Werte `Ausdruck1` zu Objekt `o1` aus.
2. Werte `Ausdruck2` zu Objekt `o2` aus.
3. Weise dem Index `o1` des mit `Name` bezeichneten Objekts das Objekt `o2` zu.

Beispiel:

```
l = [1, 2, 3]
l[1 + 1] = 2 * 2    # -> l == [1, 2, 4]
```

Achtung: Das Objekt `<Name>` muss indizierbar und veränderlich sein, sonst Fehler. Bisher ist `list` der einzige uns bekannte Typ, der dies erfüllt. (`str` ist indizierbar aber `immutable`.)

Python-Anatomie: Löschen von Container-Elementen

Element-Löschungs-Anweisung

```
del <Name> [<Ausdruck>]
```

Ausführung:

1. Werte Ausdruck zu Objekt o aus.
2. Entferne das zum Index o gehörige Element aus dem mit Name bezeichneten Objekt.

Beispiel:

```
l = [1, 2, 3]
del l[1 - 1]      # -> l == [2, 3]
```

Achtung: Das Objekt <Name> muss indizierbar und veränderlich sein, sonst Fehler. Bisher ist list der einzige uns bekannte Typ, der dies erfüllt. (str ist indizierbar aber immutable.)

Python-Anatomie: Ausdrücke (fortges.)

▶ Indizierung

```
<Ausdruck1>[<Ausdruck2>]
```

Es werden `Ausdruck1` und `Ausdruck2` zu Objekten `o1`, `o2` ausgewertet. Ergebnis ist das Element von `o1` mit Index `o2` (falls `o1` indizierbar und ein Element mit Index `o2` existiert, sonst Fehler).

▶ Identität

```
<Ausdruck1> is <Ausdruck2>
```

Es werden `Ausdruck1` und `Ausdruck2` zu Objekten `o1`, `o2` ausgewertet. Ergebnis ist `True`, falls `o1` und `o2` *dasselbe* Objekt sind.

▶ Nicht-Identität-Relation

```
<Ausdruck1> is not <Ausdruck2>
```

Es werden `Ausdruck1` und `Ausdruck2` zu Objekten `o1`, `o2` ausgewertet. Ergebnis ist `True`, falls `o1` und `o2` nicht dasselbe Objekt sind.



Tag 3

Warmup

1. Führen Sie das folgende Programm aus:

vecadd.py

```
1 v = [ 1., -3., 0.4, 5.]
2 w = [ 2., 2., 3., 3.]
3 z = []
4
5 assert len(v) == len(w), 'Laengen von v und w muessen uebereinstimmen'
6 for i in range(len(v)):
7     z.append(v[i] + w[i])
8 print(z)
```

2. Was passiert, wenn Sie in der Definition von `w` ein Element weglassen?
3. Berechnen Sie zusätzlich das Skalarprodukt der Vektoren `v` und `w`.

Listen in Listen

1. Führen Sie das folgende Programm aus:

```
listinlist.py
```

```
1 l = [0, 1, 2]
2 l2 = [3, 4]
3 l.append(l2)
4
5 print(l)
6 l2[1] = 99
7 print(l)
```

2. Was passiert, wenn Sie in Zeile 3 `append` durch `extend` ersetzen?

Interaktive Nutzung von Python

- ▶ Wird das Programm `python3` ohne Argument aufgerufen, können interaktiv Python-Anweisung vom Benutzer eingegeben werden, die dann von Python ausgeführt werden.
- ▶ Werden dieselben Anweisungen in einer Datei gespeichert und mit `python3` ausgeführt, erhält man das gleiche Ergebnis, bis auf folgende Unterschiede:
 1. Fehler führen nicht zum Absturz des Programms.
 2. Die Ergebnisse von Auswertungs-Anweisungen werden mit `print` im Terminal ausgegeben.
- ▶ Das Programm `ipython3` kann als komfortablere Alternative zu `python3` verwendet werden.
- ▶ Mit `help(o)` kann die eingebaute Python-Hilfe zu einem Objekt `o` angezeigt werden.

Benutzerdefinierte Funktionen

1. Führen Sie das folgende Programm aus:

binom.py

```
1 def fakultaet(n):
2     fak = 1
3     for k in range(1, n+1):
4         fak = fak * k
5     return fak
6
7 n = int(input('n = '))
8 k = int(input('k = '))
9 assert n >= k, 'n muss größer als k sein!'
10
11 fn = fakultaet(n)
12 fk = fakultaet(k)
13 fnk = fakultaet(n-k)
14 c = fn // (fk * fnk)
15 print(c)
```

2. Schreiben Sie eine weitere Funktion `binom(n, k)`, in welche Sie die Berechnung des Binomialkoeffizienten c für gegebenes n , k auslagern.

Lokale und globale Namen

1. Was passiert, wenn Sie folgendes Programm ausführen?

```
global.py
```

```
1 def f():  
2     # x = 1  
3     # global x  
4     print(x)  
5     # x = 100  
6  
7 x = 99  
8 f()  
9 print(x)
```

2. Was passiert, wenn Sie das Kommentarzeichen in Zeile 2 entfernen?
3. Was passiert, wenn Sie stattdessen das Kommentarzeichen in Zeile 5 entfernen?
4. Was passiert, wenn Sie zusätzlich das Kommentarzeichen in Zeile 3 entfernen?

REGEL: Ein Name ist lokal, wenn er Ziel einer Zuweisung innerhalb einer Funktion ist (und wenn er nicht als **global** markiert ist).

Rekursion

Betrachten Sie das folgende Programm zur Berechnung der Fibonaccizahlen:

fibrec.py

```
1 def fib(n):
2     if n < 2:
3         return 1
4     else:
5         return fib(n-1) + fib(n-2)
6
7 n = int(input('n = '))
8 assert n >= 0
9
10 print(fib(n))
```

1. Testen Sie ihr Programm (mindestens) für 30, 1000, 40
2. Führen Sie ihr Programm mit `pvdb` (für `n = 100`) aus.
3. Zählen Sie in einer globalen Variable `aufrufe` die einzelnen Aufrufe von `fib`, und geben Sie den Wert am Ende des Programms aus.

Funktionen sind Objekte

Betrachten Sie das folgende Programm zur Approximation des Integrals einer Funktion durch die erweiterte Mittelpunktsregel:

integral.py

```
1 def integral(f, a, b, n):
2     h = (b - a) / n
3     result = 0.
4     for i in range(n):
5         result = result + f(a + 0.5*h + i*h) * h
6     return result
7
8 def f1(x):
9     return x*3
10
11 print(integral(f1, 0., 1., 10))
```

- ▶ Geben Sie auch eine Approximation des Integrals $\int_{-1}^1 x^2 + x \, dx$ aus.

lambda-Funktionen

Betrachten Sie das folgende Programm:

integrallambda.py

```
1 def integral(f, a, b, n):
2     h = (b - a) / n
3     result = 0.
4     for i in range(n):
5         result = result + f(a + 0.5*h + i*h) * h
6     return result
7
8 poly = lambda x: x**4 + 1
9
10 print(integral(poly, 0., 1., 10))
```

- ▶ Schreiben Sie eine Funktion `konvergenztabelle(f, a, b, K)`, die Approximationen von $\int_a^b f(x) dx$ für $n \in \{10^0, 10^1, \dots, 10^K\}$ ausgibt.
- ▶ Testen Sie ihr Programm zum Beispiel für $a = 0$, $b = 1$, $f(x) := x^4$, $K = 8$.

Module

Betrachten Sie das folgende Programm zur Approximation von $\pi \approx 4 \cdot \frac{\text{treffer}}{N}$ mit Hilfe der Monte-Carlo-Methode:

pi.py

```
1 import sys
2 from random import uniform
3
4 assert len(sys.argv) >= 2, 'Benutzung: python3 ' + sys.argv[0] + ' n'
5 N = int(sys.argv[1])
6 assert N > 0
7
8 treffer = 0
9 for i in range(N):
10     x = uniform(-1, 1)
11     y = uniform(-1, 1)
12     if x**2 + y**2 < 1:
13         treffer = treffer + 1
14
15 pi_approx = 4 * (treffer / N)
16 print(pi_approx)
```

Importieren Sie zusätzlich das Modul `math`, in dem die Kreiszahl π als `math.pi` zu finden ist. Geben Sie neben der Approximation auch den Approximationsfehler aus.

Python-Anatomie: assert-Anweisung

assert-Anweisung

```
assert <Ausdruck1>
```

Ausführung: Werte Ausdruck1 aus. Falls **False**, breche das Programm mit Fehlermeldung ab.

assert-Anweisung mit Ausgabe

```
assert <Ausdruck1>, <Ausdruck2>
```

Ausführung: Werte Ausdruck1 aus. Falls **False**, werte Ausdruck2 aus und gebe das Ergebnis mit `print` als Fehlermeldung aus. Breche das Programm ab.

Python-Anatomie: def-Anweisung

def-Anweisung

```
def <Name0>(<Name1>, <Name2>, ..., <NameN>):  
    <Block>
```

Ausführung:

1. Erzeuge neues Funktionsobjekt mit N Parametern, das, wenn aufgerufen, die Anweisungen in Block ausführt. Dabei wird dem Namen NameK der Wert des K-ten Funktionsparameters zugewiesen.
2. Weise das erzeugte Funktionsobjekt dem Namen Name0 zu.

return-Anweisung

```
return <Ausdruck>
```

Ausführung:

1. Werte Ausdruck zu Objekt o aus.
2. Beende die Ausführung der Funktion und gebe o als Rückgabewert zurück.

Python-Anatomie: import-Anweisung

import-Anweisung

```
import <Name>
```

Ausführung: Lade das Modul `Name`, und weise das resultierende Modul-Objekt dem Namen `Name` zu.

from-import-Anweisung

```
from <Name1> import <Name2>
```

Ausführung: Lade das Modul `Name1`, und weise das Attribut des Moduls mit dem Namen `Name2` dem Namen `Name2` zu.

import-as-Anweisung

```
import <Name1> as <Name2>
```

Ausführung: Lade das Modul `Name1`, und weise das resultierende Modul-Objekt dem Namen `Name2` zu. Beispiel:

```
import math as m
print(m.pi)
```

Python-Anatomie: Ausdrücke (fortges.)

► lambda-Funktion

```
lambda <Name1>, <Name2>, ..., <NameN>: <Ausdruck>
```

Ergebnis der Auswertung des lambda-Ausdrucks ist ein Funktionsobjekt mit N Parametern, das, wenn aufgerufen, `Ausdruck` auswertet und als Funktionsergebnis zurückgibt. Bei der Auswertung von `Ausdruck` wird dabei dem Namen `NameK` der Wert des K -ten Funktionsparameters zugewiesen.

Python-Anatomie: Datentypen (fortges.)

► dict (**D**ictionary, **i**terierbar)

Speichert ungeordnete Paare 'Schlüssel -> Wert' von **Objekt-Referenzen**. Konstruktion über Dictionary-Literale, z.B.

```
{}
```

```
{1: '1', 2: '2', 3: '3'}
```

```
{1: f(1),  
 2: f(2)}
```

Leeres Dictionary
Dictionary mit 3 Einträgen
Dictionary-Literale dürfen beliebige Ausdrücke
als Schlüssel oder Werte enthalten

Mittels `dict(x)` kann aus den Elementen eines iterierbaren Objekts `x` von Paaren ein Dictionary erzeugt werden, z.B.

```
dict([[1, 2], [3, 4]])
```

-> {1: 2, 3: 4}

Achtung: Als Schlüssel sind i.A. nur unveränderliche Objekte erlaubt.

Die Anzahl der Einträge eines Dictionaries `d` erhalten wir mit `len(d)`. Dictionaries können mit Schlüsseln indiziert werden:

```
d = {'Hund': 2, 'Katze': 4, 4: 16}
```

```
d['Hund']
```

```
d[4]
```

```
d['4']
```

-> 2
-> 16
Fehler!

Python-Anatomie: Datentypen (fortges.)

► dict (fortges.)

Die nützlichsten Methoden von dict:

```
d.keys()           # -> iterierbares Objekt aller Schlüssel in d
d.values()         # -> iterierbares Objekt aller Werte in d
d.items()          # -> iterierbares Objekt aller (Schlüssel, Wert)-Paare in d
d.pop(x)           # -> Wert zum Schlüssel x, Eintrag wird in d entfernt
d.update(d2)       # -> Einträge von Dictionary d2 zu d hinzugefügt
```

► tuple (Tupel, immutable, iterierbar)

Speichert **unveränderliche** geordnete Listen von **Objekt-Referenzen** beliebiger Länge.
Konstruktion über Tuple-Literale, z.B.

```
()                # Leeres Tupel
(1,)              # Tupel mit einem Element (beachte das ',')
(1, len('asdf'), 'a', [1, 2]) # Tuple-Literale dürfen beliebige Ausdrücke
                               # als Elemente enthalten
```

Mittels `tuple(x)` kann aus den Elementen eines iterierbaren Objekts `x` ein Tupel erzeugt werden. Indizierung wie bei Listen.

Kann, anders als `list`, als Schlüssel in Dictionaries verwendet werden!



Tag 4

numpy.ndarray

Das Python-Paket `numpy` stellt den Typ `ndarray` zur Verfügung, mit dem mehrdimensionale Arrays (homogene Listen fester Länge) effizient dargestellt werden können.

- ▶ Insbesondere können Vektoren (eindimensionales Array) und Matrizen (zweidimensionales Array) dargestellt werden.
- ▶ Konstruktion mittels `numpy.array()`:

```
import numpy as np          # mache numpy unter dem Namen np verfügbar
v = np.array([1, 2, 3])     # -> v == Vektor [1,2,3]

A = np.array([[1, 2, 3],    # -> A == Matrix [1 2 3]
              [4, 5, 6],    #           [4 5 6]
              [7, 8, 9]])   #           [7 8 9]
```

- ▶ `A.ndim` enthält die Anzahl der Dimensionen von `A` (1 = Vektor, 2 = Matrix).
- ▶ `A.shape` ist ein `tuple`, das die Anzahl der Einträge für die jeweilige Dimension enthält:
 - ▶ Es gilt `len(A.shape) == A.ndim`.
 - ▶ Für eine Matrix enthält `A.shape` die Anzahl an Zeilen und Spalten.
 - ▶ Es gilt `len(A) == A.shape[0]`. Für eine Matrix ist `len(A)` die Anzahl der Zeilen von `A`.

Spezielle Arrays

- ▶ `np.zeros(s)` erzeugt eine Nullmatrix mit `shape s`.

Beispiel:

```
np.zeros((2, 4))      # -> array([[ 0.,  0.,  0.,  0.],  
                      #           [ 0.,  0.,  0.,  0.]])
```

- ▶ `np.ones(s)` erzeugt eine Matrix aus Einsen mit `shape s`.

Beispiel:

```
np.ones((2, 2))      # -> array([[ 1.,  1.],  
                      #           [ 1.,  1.]])
```

- ▶ `np.eye(d)` erzeugt eine d-Dimensionale Einheitsmatrix.

Beispiel:

```
np.eye(3)            # -> array([[ 1.,  0.,  0.],  
                      #           [ 0.,  1.,  0.],  
                      #           [ 0.,  0.,  1.]])
```

- ▶ `np.arange(n)` erzeugt ein 1d-Array mit den natürlichen Zahlen von 0 bis `n-1`.

Beispiel:

```
np.arange(5)        # -> array([ 0.,  1.,  2.,  3.,  4.]])
```

Grundlegende Operationen

- ▶ Die arithmetischen Operatoren `+`, `-`, `*`, `/`, `**` operieren **elementweise** auf NumPy-Arrays.
- ▶ Das `numpy`-Modul enthält zahlreiche weitere Funktionen, die elementweise auf NumPy-Arrays operieren. Z.B.
`np.sqrt`, `np.sin`, `np.cos`, `np.abs`, `np.exp`, `np.floor`, `np.ceil`, `np.round`, ...
- ▶ `A.dot(B)` berechnet das Matrixprodukt (Matrix-Vektor-Produkt) von A mit B.
- ▶ `np.min(A)`, `np.max(A)` berechnen das Minimum/Maximum aller Einträge von A.
- ▶ `np.linalg.norm(A)` berechnet die euklidische ℓ^2 -Norm der Einträge von A.

Indizierung / Slicing

- ▶ NumPy-Arrays können indiziert werden (mit einem Index pro Dimension):

```
A = np.array([[1, 2, 3],  
             [4, 5, 6],  
             [7, 8, 9]])  
  
A[0, 0]      # -> 1  
A[1, -1]    # -> 6
```

- ▶ Slicing ist möglich in jeder Dimension. Dabei bezeichnet:

- ▶ `i:j` das halboffene Intervall aller Elemente mit Indices von `i` bis ausschließlich `j`,
- ▶ `i:` alle Elemente ab (einschließlich) `i`,
- ▶ `:j` alle Elemente bis (ausschließlich) `j`,
- ▶ `:` alle Elemente.

Beispiele:

```
A[1:, :]      # -> array([[4, 5, 6],  
                    [7, 8, 9]])  
A[:, 1:2]    # -> array([[2],  
                    [5],  
                    [8]])  
A[:2, :2]   # -> array([[1, 2],  
                    [4, 5]])
```

Indizierung / Slicing (fortges.)

- ▶ Slicing und Indizierung können auch kombiniert werden:

```
A = np.array([[1, 2, 3],  
             [4, 5, 6],  
             [7, 8, 9]])  
  
A[0, :]          # -> array([1, 2, 3])  
A[:, 2]         # -> array([3, 6, 9])  
A[1, 1:]        # -> array([5, 6])
```

- ▶ `A[i]` ist für eine Matrix `A` äquivalent zu `A[i, :]`:

```
A[0]            # -> array([1, 2, 3])  
A[1]            # -> array([4, 5, 6])  
A[2]            # -> array([7, 8, 9])
```

- ▶ Mit `for` iteriert man über die Einträge eines Vektors oder die Zeilen einer Matrix:

```
for x in A:  
    print(x)
```

macht die Ausgabe:

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

Elemente ändern

- Die Elemente eines Arrays können durch Indizierung/Slicing geändert werden:

Beispiele:

```
A = np.array([[1, 2, 3],  
             [4, 5, 6]],
```

```
A[0, 0] = 7 # -> A == array([[7, 2, 3],  
                        [4, 5, 6]])
```

```
A[0, 1:] = A[1, 1:] # -> A == array([[7, 5, 6],  
                        [4, 5, 6]])
```

```
A[:, 1:] = 9 # -> A == array([[7, 9, 9],  
                        [4, 9, 9]])
```

```
A[:] = 0 # -> A == array([[0, 0, 0],  
                        [0, 0, 0]])
```

Views und Kopien

- ▶ Erhält man durch Indizierung/Slicing ein neues NumPy-Array, so ist dieses ein **View** auf die Daten des indizierten Arrays.

Werden die Elemente des View-Arrays verändert, verändern sich auch die entsprechenden Einträge des ursprünglichen Arrays!

Beispiel:

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])

B = A[1, 1:]          # -> B == array([5, 6])

B[:] = 0             # -> B == array([0, 0])
                    #   A == array([[1, 2, 3],
                    #             [4, 0, 0]])

B.base is A         # True
```

- ▶ `A.copy()` liefert ein neues Array mit einer Kopie der Daten von A zurück.
- ▶ *Advanced Indexing* (siehe NumPy-Dokumentation) erzeugt immer eine Kopie.

dtype

- ▶ Anders als `list`, `tuple`, `dict` haben alle Einträge von NumPy-Arrays den selben Typ.
- ▶ `A.dtype` enthält den Typ der Einträge des Arrays `A`.
- ▶ Es gilt:

```
np.zeros(s).dtype      == np.float64
np.ones(s).dtype       == np.float64
np.eye(d).dtype        == np.float64
np.arange(s).dtype     == np.int64
np.array([1,2,3]).dtype == np.int64
np.array([1.,2,3]).dtype == np.float64
```

- ▶ **Achtung:**

```
v = np.arange(5)      # -> v == array([0, 1, 2, 3, 4], dtype=np.int64)
v[:] = v[:] / 2      # -> v == array([0, 0, 1, 1, 2], dtype=np.int64)
```

Array-Konstruktionen

- ▶ `np.hstack([A1, A2])`, `np.vstack([A1, A2])` erlauben es, NumPy-Arrays horizontal/vertikal zu neuen Arrays zusammenzufügen:

```
np.hstack([np.arange(3), np.ones(3)]) # -> array([0, 1, 2, 1, 1, 1])
```

```
np.vstack([np.arange(3), np.ones(3)]) # -> array([[0, 1, 2],  
#          [1, 1, 1]])
```

- ▶ `A.reshape(s)` erlaubt es, die Elemente eines Arrays in einem neuen Array anders anzuordnen:

```
A = np.arange(6) # -> A == array([0, 1, 2, 3, 4, 5])  
B = A.reshape((2, 3)) # -> B == array([[0, 1, 2],  
#          [3, 4, 5]])  
B.base is A # -> True
```

Achtung: Das Ergebnis von `reshape` ist in der Regel ein View auf das Ursprungsarray.

- ▶ `A.ravel()` liefert ein eindimensionales Array mit den Elementen von A:

```
np.eye(2).ravel() # -> array([1, 0, 0, 1])
```

Achtung: Das Ergebnis ist in der Regel ein View auf A.

Array-Achsen / Broadcasting

- ▶ Viele Funktionen von NumPy können auch entlang einer ausgewählten Dimension (Achse) des Arrays ausgeführt werden:

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])

np.sum(A)                # -> 21
np.sum(A, axis=0)        # -> array([5, 7, 9])
np.argmax(A, axis=1)     # -> array([2, 2])
```

- ▶ I.d.R müssen Array-Dimensionen kompatibel sein. NumPy bläst aber bei Bedarf Achsen der Länge 1 durch Wiederholung auf die passende Länge auf:

```
A * np.array([1, 2, 3, 4]) # -> ValueError

A * 2                       # -> array([[2, 4, 6],
                                         [8, 10, 12]])

A * np.array([1, 2, 3])     # -> array([[1, 4, 9],
                                         [4, 10, 18]])
```

(Siehe **broadcasting** in der NumPy-Dokumentation.)

==, all und any

- ▶ ==, !=, <, <=, >, >= operieren ebenfalls elementweise auf NumPy-Arrays:

```
v = np.array([1, 2, 3, 4])
w = np.array([1, 2, 5, 4])
v == w                # -> array([True, True, False, True])
```

Ein solches Array hat keinen eindeutigen Wahrheitswert, daher führt

```
if v == w:            # -> Fehler! Wahrheitswert nicht eindeutig!
    print('Gleich')
```

zu einem Fehler.

- ▶ `np.all(A)` ist **True**, wenn alle Elemente von A wahr sind.
`np.any(A)` ist **True**, wenn mindestens ein Element von A wahr ist.

Wir können also z.B. schreiben:

```
if np.all(v == w):
    print('alles gleich!')
```

Fingerübungen

- ▶ Stellen Sie folgende Matrix und folgenden Vektor als NumPy-Array dar:

$$A = \begin{pmatrix} 1 & 2 & 7 & 4 \\ 0 & 9 & 1 & 1 \\ 0 & 0 & 0 & 6 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad v = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

- ▶ Berechnen Sie das Produkt $A \cdot v$.
- ▶ Ersetzen Sie die erste Zeile von A durch v .
Ersetzen sie die letzte Spalte von A durch $2v$.
- ▶ Berechnen Sie $A \cdot v$ ohne die `dot`-Methode.
- ▶ Berechnen Sie die ℓ^2 -Norm von v .
- ▶ Finden Sie in der NumPy-Dokumentation einen Weg, die Determinante von A , sowie $A^{-1} \cdot v$ zu berechnen.



Tag 5

Python-Anatomie: Datentypen (fortges.)

► set (Menge, iterierbar)

Speichert (ungeordnete) Mengen von **Objekt-Referenzen** beliebiger Größe.
Konstruktion über Mengen-Literale, z.B.

```
{1, 2, 3}           # Menge mit drei Elementen
{1, 2, 1}         # Menge mit zwei Elementen
{1, len('asdf'), 'a', [1, 2]} # Mengen-Literale dürfen beliebige Ausdrücke
                        # als Elemente enthalten
```

Mittels `set(x)` kann aus den Elementen eines iterierbaren Objekts `x` eine Menge erzeugt werden, z.B.

```
set('foo')        # -> {'f', 'o', 'o'}
```

`set()` erzeugt eine leere Menge. Die Mächtigkeit einer Menge `s` erhalten wir mit `len(s)`.

Die nützlichsten Methoden von `set`:

```
s.add(x)          # -> x als Element hinzugefügt
s.update(s2)      # -> Elemente von s2 hinzugefügt
s.intersection(s2) # -> Schnittmenge von s und s2
s.issubset(s2)    # -> True, falls s Teilmenge von s2
s.isdisjoint(s2)  # -> True, falls s und s2 disjunkt
```

Was ist wahr?

- ▶ `bool(x)` ordnet jedem Python-Objekt `x` einen Wahrheitswert zu.

- ▶ Es ist

```
if x:  
    <Block>
```

äquivalent zu

```
if bool(x):  
    <Block>
```

Wir können also z.B. schreiben:

```
s = input('Eingabe')  
if s:  
    print('wahr')
```

Dabei wird 'wahr' ausgegeben, wenn `bool(s)` den Wahrheitswert `True` ergibt.

Was ist wahr? (fortges.)

- ▶ Jede Zahl außer 0 ist **True**:

```
bool(0)      == False
bool(0.)     == False
bool(7)      == True
bool(0.1)    == True
```

- ▶ Container sind **False** genau dann, wenn Sie leer sind:

```
bool([])     == False
bool('')     == False
bool({})     == False
bool([False]) == True
bool('False') == True
```

- ▶ `bool(None)` == **False**

Logische Operatoren

▶ `<Ausdruck1> or <Ausdruck2>`

Auswertung:

1. Werte Ausdruck1 zu Objekt o1 aus. Ergebnis ist o1, falls `bool(o1) == True`.
2. Andernfalls werte Ausdruck2 zu Objekt o2 aus. Ergebnis ist o2.

Diese Regel erlaubt Abkürzungen wie z.B.

```
name = input('Ihr Name? ') or 'Namenloser'
```

Es gilt stets:

```
bool(<Ausdruck1> or <Ausdruck2>) == bool(<Ausdruck1>) or bool(<Ausdruck2>)
```

Wie in der Mathematik ist `or` in Python ein inklusives Oder.

▶ `<Ausdruck1> and <Ausdruck2>`

Auswertung:

1. Werte Ausdruck1 zu Objekt o1 aus. Ergebnis ist o1, falls `bool(o1) == False`.
2. Werte Ausdruck2 zu Objekt o2 aus. Ergebnis ist o2.

▶ `not <Ausdruck>`

Auswertung: Werte Ausdruck zu Objekt o aus. Ergebnis ist `True`, falls `bool(o) == False`, sonst `True`.

Inplace-Operatoren

- ▶ Python hat 'inplace'-Varianten der arithmetischen Operatoren `+`, `-`, `*`, `/`, `//`, nämlich `+=`, `-=`, `*=`, `/=`, `//=`.
- ▶ Für unveränderliche (immutable) Typen wie `int`, `float`, `str`, `tuple` sind diese Methoden dabei so implementiert, dass gilt
 - ▶ `x += y` ist äquivalent zu `x = x + y`
 - ▶ `x -= y` ist äquivalent zu `x = x - y`
 - ▶ `x *= y` ist äquivalent zu `x = x * y`
 - ▶ `x /= y` ist äquivalent zu `x = x / y`
 - ▶ `x //= y` ist äquivalent zu `x = x // y`
- ▶ Für veränderliche Typen wie `list`, `numpy.ndarray` wird hingegen das jeweilige Objekt `x` **verändert!**
Für Listen `l`, `12` ist z.B. die Anweisung `l += 12` äquivalent zu `l.extend(12)`.

Optionale Argumente

► In

```
def f(a, b=42):  
    return a - b
```

ist `b` ein optionales Argument welches, wenn nicht beim Aufruf angegeben, den Wert `42` hat:

```
f(50, 1)           # -> 49  
f(50)              # -> 8
```

► Die Argumente einer Funktion können beim Aufruf auch mit Namen spezifiziert werden (**keyword argument**):

```
f(b=1, a=50)      # -> 49
```

Dies ist besonders nützlich, wenn die Funktion viele optionale Argumente hat.

► Die Funktion

```
def g(*args, **kwargs):  
    print(args, kwargs)
```

nimmt beliebig viele (keyword) Argumente and, die in `g` in der Liste `args` und dem Dictionary `kwargs` verfügbar sind:

```
g(1, 2, 3, a=4, b=5)      # -> Ausgabe: [1, 2, 3] {'a': 4, 'b': 5}
```

Tuple unpacking

- ▶ Folgende Syntax erlaubt es, die Elemente eines Tupels / einer Liste schnell mehreren Namen zuzuweisen:

```
a, b, c = [1, 2, 3]           # -> a == 1, b == 2, c == 3
```

- ▶ Bei Tupeln können oft die Klammern weggelassen werden. Insbesondere können wir so leicht die Werte zweier Namen vertauschen:

```
a, b = b, a
```

- ▶ Dies ist auch nützlich, wenn eine Funktion mehrere Rückgabewerte hat:

```
def summe_produkt(a, b):  
    return a + b, a * b
```

```
s, p = summe_produkt(2, 3)   # -> s == 5, p == 6
```

List- und Dictionary-Comprehensions

- ▶ Anstelle von

```
l = []  
for x in l2:  
    l.append(f(x))
```

können wir auch eine kürzere **List-Comprehension** verwenden:

```
l = [f(x) for x in l2]
```

- ▶ Das geht ähnlich auch für Dictionaries:

```
d = {x: x**2 for x in range(4)} # -> d == {0: 0, 1: 1,  
#           2: 4, 3: 9}
```

Funktionsgraphen mit matplotlib

Mit matplotlib lässt sich eine Vielzahl von Datenvisualisierungen erstellen.

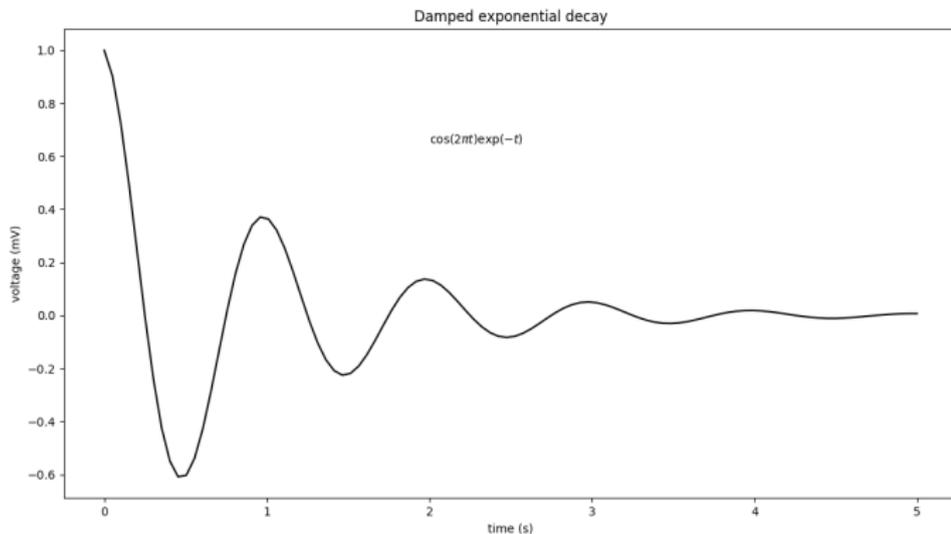
Ein Beispiel:

decay.py

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(0.0, 5.0, 100)
5 y = np.cos(2 * np.pi * x) * np.exp(-x)
6
7 plt.plot(x, y, 'k')
8 plt.title('Damped exponential decay')
9 plt.text(2, 0.65, r'$\cos(2 \pi t) \exp(-t)$')
10 plt.xlabel('time (s)')
11 plt.ylabel('voltage (mV)')
12 plt.show()
```

Funktionsgraphen mit matplotlib

Das Ergebnis:



Mehr Beispiele unter <https://matplotlib.org/gallery.html>

Einige wichtige Erweiterungsmodulare

- ▶ `SciPy`: Mehr numerische Algorithmen, Sparse-Matrizen
- ▶ `Pandas`: Statistische Datenanalyse ähnlich zu R
- ▶ `SymPy`: Computer Algebra mit Python
- ▶ `pillow`, `scikit-image`: Bildbearbeitung
- ▶ `scikit-learn`: Machine Learning
- ▶ `keras`: Neuronale Netze
- ▶ `jupyter`: interaktive Python-Notebooks im Webbrowser
- ▶ ...

Was sonst noch fehlt

- ▶ Benutzerdefinierte Typen (class-Anweisung)
- ▶ Dekoratoren
- ▶ Metaklassen
- ▶ Exceptions
- ▶ Closures
- ▶ Context Manager
- ▶ Generatoren
- ▶ Speicherverwaltung: Reference Counting und Garbage Collection
- ▶ ...



Zusammenfassung

Zusammenfassung, Teil 1

Grundlegende Datentypen:

- ▶ Zahlen in \mathbb{Z} : `int` `-1, 0, int('3')`
- ▶ Zahlen in \mathbb{R} : `float` (approximativ) `-1.3, 3e-5, float('3.5')`
- ▶ Zeichenketten: `str` `'Wort', 'Ein ganzer Satz!', "Noch einer", str(4)`
- ▶ Wahrheitswerte: `bool` `True, False, bool('r')`

Arithmetische Operatoren:

- ▶ `+, -, *, /`
- ▶ Ganzzahl Division: `//` (z.B. `3 // 2 == 1`)
- ▶ Modulo Operator: `%` (z.B. `5 % 2 == 1`)

Zusammengesetzte Datentypen (Container):

- ▶ Listen von beliebigen Objekten: `list` `[1], [2, 'foo', 4 + 3]`
- ▶ Dictionary: `{'Teilnehmer': 13, 'Fenster': 7}`

Methoden von Objekten:

- ▶ `'schrei mich nicht an'.upper()` `# 'SCHREI MICH NICHT AN'`
- ▶ `[1, 2, 3].append('Foo')` `# [1, 2, 3, 'Foo']`

Zusammenfassung, Teil 2

Die `while`-Anweisung zur Wiederholung, bis ... (Vorsicht: Endlos-Schleife!):

```
x = -200
while x < -10 or x > 10:
    x = uniform(-100, 100)
print(x)                                # x is now between -10 and 10
```

Die `for` Schleife zum Iterieren über alle Elemente:

```
for element in [1, 3, 'Foo']:           |   for factor, exponent in {2: 1, 17: 2, 23: 3}:
    print(element)                       |       print(factor, exponent)
```

(Zähl-)Schleifen:

```
i = 0                                   |
while i < 10:                           |   for i in range(10):
    print(i)                              |       print(i)
    i = i + 1                             |
```

Zusammenfassung, Teil 3

Vergleichs-Operatoren:

▶ `==, !=, <, <=, >, >=`

Interaktive Eingaben einlesen:

▶ `x = int(input('Zahl: '))`

Die `if`-Anweisung:

```
if x < 0:
    print('Negativ!')
elif x == 17:
    print('Was für eine Zahl!')
else:
    print('Das darf nicht wahr sein!')
```

Zusammenfassung, Teil 3

Bestehende Funktionalität aus Modulen nutzen:

```
from random import uniform
print('A random nummer: ', uniform(0, 1))

import math as m
print(m.sin(m.pi))
```

Kommandozeilenargumente entgegen nehmen:

```
import sys
print('Name: ', sys.argv[0])
print('Anzahl Argumente: ', len(sys.argv) - 1)
for i in range(1, len(sys.argv)):
    print('Argument ', i, ': ', sys.argv[i])
```

```
# python3 hallo.py 1.5 Hey
# Name: hallo.py
# Anzahl Argumente: 2
# Argument 1: 1.5
# Argument 2: Hey
```

Zusammenfassung, numpy

```
import numpy as np
```

1-dimensionale Arrays: Vektoren

▶ `v = np.array([1, 2, 3])` $\in \mathbb{Z}^3$

```
print(v.ndim) # 1
print(len(v)) # 3
```

▶ `w = np.zeros(17)` $\in \mathbb{R}^{17}$

2-dimensionale Arrays: Matrizen

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])
print(A.ndim)           # 2
print(A.shape)          # (2, 3)
```

```
zeilen, spalten = A.shape
```

```
B = np.eye(3) # [[1, 0, 0],
                # [0, 1, 0],
                # [0, 0, 1]]
```

Zusammenfassung, numpy (forges.)

Arithmetische Operationen agieren elementweise!

```
A = np.array([[1, 2, 3],  
             [4, 5, 6]])
```

```
print(A * 2) # [[2, 4, 6],  
             # [8, 10, 12]]
```

Matrix-Vektor-Produkt und inneres Produkt (Skalarprodukt):

```
A.dot(v)  
v.dot(v)
```

Vielen numpy Funktionen agieren elementweise: `np.sin(A)` verhält sich wie

```
import math as m
```

```
np.array([[m.sin(1), m.sin(2), m.sin(3)],  
        [m.sin(4), m.sin(5), m.sin(6)]])
```

Slicing gibt ein *View* zurück, mit dem das ursprüngliche Array verändert wird:

```
B = A[:2, :2]  
B[:] = 0
```

```
print(A) # [[0, 0, 3],  
          # [0, 0, 6]]
```

Zusammenfassung, numpy (forges.)

numpy enthält nützliche Funktionen ...

- ▶ `np.sin`, `np.exp`, `np.sqrt`, ...
- ▶ `np.max(A)`, `np.argmax(A)`
- ▶ `np.linalg.norm(A)`
- ▶ `np.linalg.inv(A)`

... und noch viel mehr!



Funktionen und Funktionsgraphen

siehe Transskript