WWU
MÜNSTER

Einführung in die Programmierung zur Numerik mit Python – E-Learning-Version

Stephan Rave

26. März 2020

wissen.leben

Kursinhalte

Tag 1:

- Warum Python?
- Erste Python-Programme

Tag 2:

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

Kursinhalte (fortges.)

Tag 3:

- Funktionen
- Module

Tag 4:

- Effiziente numerische Algorithmen mit NumPy

Tag 5:

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

Tag 1

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 $f \in L^2(\Omega)$:

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

Schwache Formulierung

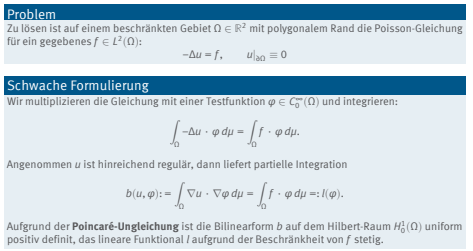
Wir multiplizieren die Gleichung mit einer Testfunktion $\varphi \in C_0^\infty(\Omega)$ und integrieren:

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

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 Beschränkheit von f stetig.



Natürlich müssen Sie hier nicht alle Details verstehen. Die wesentlich Botschaft ist, dass wir eine partielle Differentialgleichung, die Poisson-Gleichung, mit Mitteln der Funktionalanalysis umformulieren können zu dem linearen Gleichungssystem

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

auf dem unendlichdimensionalen Vektorraum $H_0^1(\Omega)$. (u ist die gesuchte Lösung, und jede Testfunktion φ liefert eine linear Gleichung, die u erfüllen muss.)

Zahlreiche physikalische Gesetze lassen sich durch die Poisson-Gleichung beschreiben. Beispielsweise beschreibt sie stationäre Wärmeleitung, Diffusionsprozesse oder das elektrostatische Feld.

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

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

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

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

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

Programmieren in der Numerik - Ein Beispiel

Schwache Formulierung (fortges.)

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

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

Finite-Elemente-Diskretisierung

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

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

$$b(u_h, \varphi_h) = l(\varphi_h) \qquad \forall \varphi \in S^1_{0,h}.$$

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

Aufgabe der Numerik ist es nun, endlichdimensionale Approximationen für u zu finden. Durch Wahl eines geeigneten endlichdimensionalen Teilraums $\mathcal{S}^1_{h,0} \subset H_0^1(\Omega)$ reduzieren wir das Ursprungsproblem auf ein Problem der linearen Algebra, die Lösung eines (endlichdimensionalen) linearen Gleichungssystems.

Programmieren in der Numerik - Ein Beispiel

Wie rechnet man das aus??

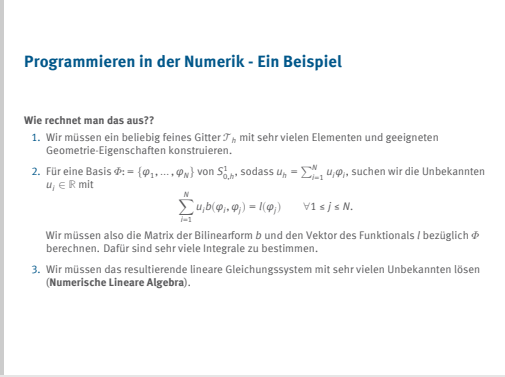
1. Wir müssen ein beliebig feines Gitter $\mathcal{T}_h$ mit sehr vielen Elementen und geeigneten Geometrie-Eigenschaften konstruieren.

2. Für eine Basis $\Phi := \{\varphi_1, \dots, \varphi_N\}$ von $S^1_{0,h}$, sodass $u_h = \sum_{i=1}^N u_i \varphi_i$, suchen wir die Unbekannten $u_i \in \mathbb{R}$ mit

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

   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**).



1. Das Generieren von geeigneten Gittern ist ein eigenständiges Forschungsgebiet an der Schnittstelle von Mathematik und Informatik.

2. Ebenso ist das effiziente Aufstellen des konkreten linearen Gleichungssystems, insbesondere für Finite-Elemente-Räume $\mathcal{S}^p_{0,h}$ höherer Ordnung eine herausfordernde Programmieraufgabe. Zahlreiche komplexe Softwarepakete existieren hierfür. In unserer Arbeitsgruppe sind wir insbesondere an der Entwicklung von DUNE beteiligt.

3. Zwar lässt sich theoretisch jedes lineare Gleichungssystem mit dem Gaußverfahren lösen, jedoch ist dieses auch mit dem Computer bei vielen Millionen Unbekannten, wie sie bei der Diskretisierung partieller Differentialgleichungen leicht auftreten, nicht mehr praktikabel. Es müssen daher Algorithmen gefunden und effizient implementiert werden, die die Struktur der auftretenden Gleichungssysteme möglichst gut ausnutzen.

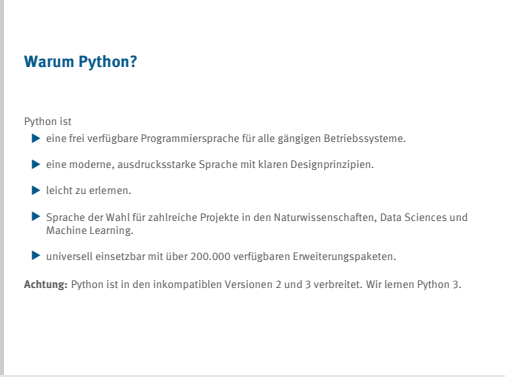
   Das Lösen linearer Gleichungssystem ist eine fundamentale Aufgabe des wissenschaftlichen Rechnens. Die Effizienz beim Lösen dieser Gleichungssysteme ist daher auch ein grundlegender Benchmark für jeden Supercomputer.

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 200.000 verfügbaren Erweiterungspaketen.

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



Dass Python eine bedeutende Programmiersprache ist, finden nicht nur wir:

- Der Tiobe Index gilt wohl als das wichtigste Ranking für Programmiersprachen weltweit. Gegenwärtig belegt Python dort, hinter Java und C den dritten Platz. Insbesondere ist Python damit die höchstplatzierte dynamische Programmiersprache (Skriptsprache) im Index.

- Die IEEE sieht Python in ihrem Ranking sogar an Platz 1.

Hallo Welt

1. Geben Sie die folgenden Zeilen in ein Editor-Fenster in Thonny ein:

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 durch Klick auf den Play-Button in der Symbolleiste aus.

3. Führen Sie das Programm auch mit dem integrierten Debugger von Thonny aus.

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


Hallo Welt

1. Geben Sie die folgenden Zeilen in ein Editor-Fenster in Thonny ein:

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 durch Klick auf den Play-Button in der Symbolleiste aus.
3. Führen Sie das Programm auch mit dem integrierten Debugger von Thonny aus.
4. Was passiert, wenn Sie `laenge` statt `str(laenge)` in Zeile 7 schreiben?

- Ein `Python`-Programm ist eine Textdatei, die eine Aneinanderreihung von Anweisungen enthält.

- Die Anweisungen werden zeilenweise von oben nach unten abgearbeitet. Eine Anweisung kann eine (Zeilen 1, 2, 6, 7) oder mehrere Zeilen lang sein (Zeilen 3-4). Ein `Python`-Programm kann beliebig viele Leerzeilen beinhalten, die keinen Einfluss auf den Programmfluss haben.

3. Mit dem Debugger von `Thonny` können Sie Schritt für Schritt die Ausführung eines Programmes nachvollziehen. Mit ‘Step Over’ (F6) springen Sie zur nächsten Programmzeile. ‘Step into’ (F7) wertet Ausdrücke in Einzelschritten aus und springt in selbstdefinierte Funktionen. Besonders nützlich ist die ‘Variablen’-Ansicht die Sie im Menü ‘View’ (‘Ansicht’) auswählen können.

4. Sie erhalten Sie einen Fehler, der in etwa wie folgt aussehen sollte:

```
Traceback (most recent call last):
  File "/home/stephan/PythonKurs/slides/tag_1/hello.py", line 7, in <module>
    print('Ihr Name ist ' + laenge + ' Zeichen lang.')
TypeError: can only concatenate str (not "int") to str
```

Die eigentliche Fehlermeldung steht immer ganz unten! Hier, besagt die Meldung, dass ein `int` nicht an ein `str` gehängt werden kann. Alle Objekte, die Python verarbeitet haben einen **Typ**, hier `int` (eine ganze Zahl) und `str` (eine Zeichenkette). **Der Typ eine Objektes bestimmt sein Verhalten und welche Operationen an/mit ihm ausgeführt werden können.**

Beachten Sie, dass auch die **Programmzeile** angegeben wird, in der der Fehler aufgetreten ist (hier Zeile 7). Verwenden Sie Thonny, so öffnet sich automatisch ein ‘Assistent’, der Ihnen oft nützliche Tipps zu der Fehlermeldung gibt.

Ein weiteres Beispiel

1. Führen Sie das folgende Programm aus (auch mit dem Debugger):

summe.py

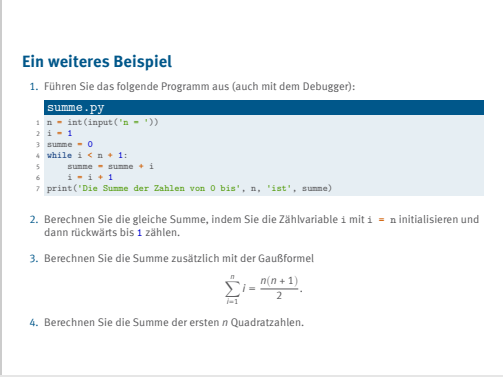
```
1 n = int(input('n = '))
2 i = 1
3 summe = 0
4 while i < n + 1:
5     summe = summe + i
6     i = i + 1
7 print('Die Summe der Zahlen von 0 bis', n, 'ist', summe)
```

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.



- Die Argumente von Funktionsaufrufen dürfen wiederum Funktionsaufrufe beinhalten (Zeile 1).
- Die `print`-Funktion konvertiert automatisch ihre Argumente in Zeichenketten (Zeile 7). Anders als mathematische Funktionen können manche `Python`-Funktionen eine variable Anzahl an Argumenten erlauben. Im Falle der `print`-Funktion werden die einzelnen Argumente durch Leerzeichen getrennt ausgegeben (Zeile 7).

1. Lösung:

```
n = int(input('n = '))
i = n
summe = 0
while i > 0:
    summe = summe + i
    i = i - 1
print('Die Summe der Zahlen von 0 bis', n, 'ist', summe)
```

2. Lösung:

```
summe = n * (n + 1) / 2
```

3. Hier muss Zeile 5 von `summe.py` ersetzt werden durch:

```
summe = summe + i * i
```

   Statt `i * i` kann auch `i ** 2` geschrieben werden.

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

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

```
i = 1
while i < 100 + 1:
    if i % (3 * 5) == 0:   # '%' ist der 'modulo' Operator
        print('FizzBuzz')
    elif i % 3 == 0:
        print('Fizz')
    elif i % 5 == 0:
        print('Buzz')
    else:
        print(i)
    i = i + 1
```

- Dies ist eine sehr bekannte Aufgabe, die angeblich häufig bei Vorstellungsgesprächen für Programmierer gestellt wird.

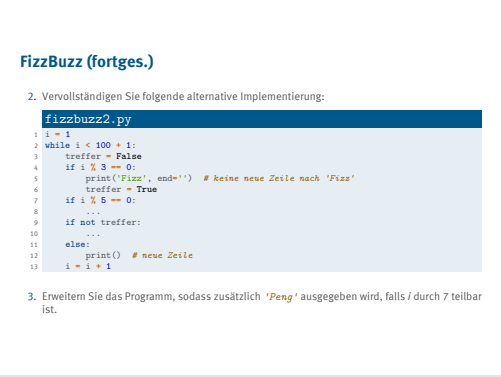
FizzBuzz (fortges.)

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.


FizzBuzz (fortges.)

2. Vervollständigen Sie folgende alternative Implementierung:

fizzbuzz2.py

```
i = 1
while i < 100 + 1:
    treffer = False
    if i % 3 == 0:
        print('Fizz', end='')  # keine neue Zeile nach 'Fizz'
        treffer = True
    if i % 5 == 0:
        ...
    if not treffer:
        ...
    else:
        print()  # neue Zeile
    i = i + 1
```

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

2. Lösung:

```
i = 1
while i < 100 + 1:
    treffer = False
    if i % 3 == 0:
        print('Fizz', end='')  # keine neue Zeile nach 'Fizz'
        treffer = True
    if i % 5 == 0:
        print('Buzz', end='')
        treffer = True
    if not treffer:
        print(i)
    else:
        print()  # neue Zeile
    i = i + 1
```

3. Anders als beim ersten Lösungsansatz, brauchen wir hier nur den folgenden Block an die geeignete Stelle einzufügen:

```
if i % 7 == 0:
    print('Peng', end='')
    treffer = True
```

Tag 2

Warmup

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

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

1. Führen Sie das folgende Programm 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)
```

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.

Warmup

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

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

1. Führen Sie das folgende Programm aus:

heron.py

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

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.

Lösung zu 2. und 3.:

```
a = int(input('a = '))
niter = int(input('niter = '))
x = 1
i = 0
while i < niter:
    x = 0.5 * (x + a/x)
    i = i + 1
    print(x, x*x - a)
```

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** (**Variablen**) die auf Objekte verweisen, sowie **Schlüsselwörter**, die Anweisungen und Ausdrücke strukturieren.*

Beispiele:

- `if`, `else`, `elif`, `while` sind Schlüsselworte.
- `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.

fak.py

```
1 fak = 1
2 i = 1
3 while True:
4     fak = fak * i
5     print(fak)
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** (Variablen)

  ```
  <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.
- Schlüsselworte dürfen nicht als Namen verwendet werden.
- Die Zeichen `-` und `.` dürfen nicht verwendet werden.
- Unterschiedliche Groß-/Kleinschreibung führt zu verschiedenen Namen.

Beispiele:

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

Python Schlüsselworte

```
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

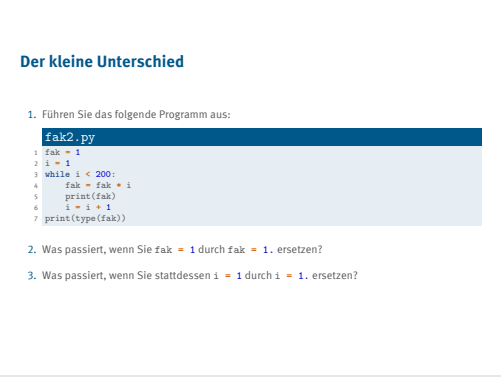
1. Führen Sie das folgende Programm aus:

fak2.py

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

2. Was passiert, wenn Sie `fak = 1` durch `fak = 1.` ersetzen?

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



2. Sie erhalten `inf`, also ‘unendlich’ als Ergebnis. Durch den Punkt hinter `1` interpretiert Python die Zahl als Dezimalbruch und legt ein `float`-Objekt an, welches so genannte Fließkommazahlen speichert. Anders als `int`-Objekte, die beliebig große (so lange der Speicher reicht) ganze Zahlen speichern können, speichert ein `float` Zahlen in der Darstellung

   $$(-1)^s \cdot m \cdot 2^e$$

   wobei $1 \leq m < 2$ ein Binärbruch mit einer festen Anzahl an Stellen ist und e eine ganze Zahl mit einem maximalen Absolutbetrag. Die Details hängen vom verwendeten Mikroprozessor und Betriebssystem ab. In der Regel handelt es sich jedoch um IEEE 754 doubles, für die die Mantisse m 52 Nachkommastellen hat und der Exponent Werte zwischen −1022 und 1023 annehmen kann. Zusätzlich sind spezielle Werte vorgesehen: `inf`, `-inf`, `nan` stehen für ‘plus unendlich’, ‘minus unendlich’ und ‘not a number’. Letzteren Wert erhält man z.B., wenn 0 durch 0 geteilt wird. (Bei Python-`floats` führt Division durch 0 immer zu einem Fehler, bei NumPy ist das Verhalten konfigurierbar.)

3. Egal ob ein `int` mit einem `float` oder ein `float` mit einem `int` multipliziert wird, das Ergebnis ist immer ein `float`.

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

1. Führen Sie das folgende Programm aus:

fibolist.py

```
fibs = [1, 1]
i = 2
while i < 10:
    fibs.append(fibs[i-2] + fibs[i-1])
    i = i + 1
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.

Lösung von 2. und 3.:

```
fibs = [1, 1]
quots = [1]
i = 2
while i < 10:
    fibs.append(fibs[-2] + fibs[-1])
    quots.append(fibs[-1] / fibs[-2])
    i = i + 1
print(fibs)
print(quots)
```

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

1. Führen Sie das folgende Programm aus:

prod.py

```
l = [4, 6, 3, 2]
prod = 1
for x in l:
    prod = prod * x
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.

Lösung von 2. und 3.:

```
l = [4, 6, 3, 2]
prod = 1
s = 0
min = l[0]
max = l[0]
for x in l:
    prod = prod * x
    s = s + x
    if x < min:
        min = x
    if x > max:
        max = x
print(prod, s, min, max)
```

Listen und for-Schleifen (fortges.)

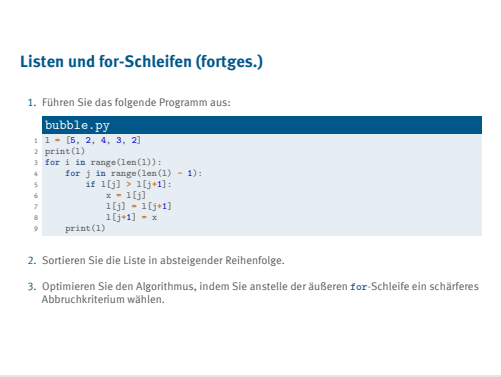
1. Führen Sie das folgende Programm aus:

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.



- Bubblesort ist eines der einfachsten, aber auch eines der langsamsten Sortierverfahren. Seinen Namen hat es erhalten, weil es in seiner Funktionsweise an im Wasser aufsteigende Luftblasen erinnert.

- Lösung zu 2. und 3.:

```
l = [5, 2, 4, 3, 2]
print(l)
vertauscht = True
while vertauscht:
    vertauscht = False
    for j in range(len(l) - 1):
        if l[j] < l[j+1]:
            x = l[j]
            l[j] = l[j+1]
            l[j+1] = x
            vertauscht = True
    print(l)
```

Identität und Gleichheit

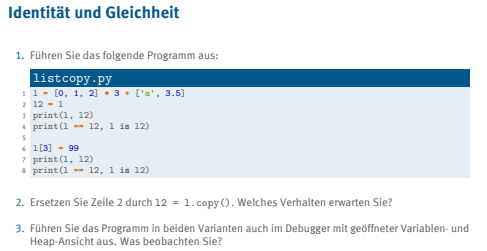
1. Führen Sie das folgende Programm aus:

listcopy.py

```
1 l = [0, 1, 2] * 3 + ['a', 3.5]
2 l2 = l
3 print(l, l2)
4 print(l == l2, l is l2)
5
6 l[3] = 99
7 print(l, l2)
8 print(l == l2, l is l2)
```

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

3. Führen Sie das Programm in beiden Varianten auch im Debugger mit geöffneter Variablen- und Heap-Ansicht aus. Was beobachten Sie?



1. Anders als `int`-, `float`- und `str`-Objekte können `list`-Objekte modifiziert werden. Durch die Zuweisung in Zeile 2 wird lediglich dem existierenden `list`-Objekt mit Namen `l` ein weiterer Name, `l2`, gegeben. Dementsprechend führt die Modifikation von `l` in Zeile 6 auch zu einer Änderung von `l2`, da es sich bei `l2` eben gerade um dasselbe Objekt handelt.

2. Die `copy()`-Method von `list`-Objekten erzeugt ein neues `list`-Objekt mit denselben Einträgen. Eine Modifikation von `l` bewirkt daher keine Änderung des neuen `list`-Objekts, auf das nun `l2` verweist. Mit dem `is`-Operator können Sie prüfen, ob es sich bei zwei Objekten um dasselbe Objekt handelt. Der `==`-Operator prüft hingegen, ob die Objekte gleich sind.

3. Wenn Sie in Thonny die Heap-Ansicht öffnen, sehen sie im Heap-Fenster die im Speicher (Heap) befindlichen Python-Objekte mit ihren Speicheradressen (ID). In der Variablen-Ansicht sehen Sie nun deutlich, dass Variablen (Namen) in Python, lediglich Verweise auf Python-Objekte im Speicher sind. Insbesondere können Sie sehen, dass im unmodifizierten Programm `l` und `l2` auf dasselbe Objekt verweisen, im modifizierten Programm jedoch auf verschiedene Objekte.

Python-Anatomie: Objekte

*Ein Python-Objekt enthält **Daten** einer bestimmen 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 dieselben 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: 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.

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), 'Längen von v und w müssen übereinstimmen'
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 `v` ein Element weglassen?

3. Berechnen Sie zusätzlich das Skalarprodukt der Vektoren `v` und `w`.

2. Das Programm bricht mit einer Fehlermeldung ab:

```
Traceback (most recent call last):
  File "/home/stephan/PythonKurs/slides/tag_3/vecadd.py", line 5, in <module>
    assert len(v) == len(w), 'Längen von v und w müssen übereinstimmen'
AssertionError: Längen von v und w müssen übereinstimmen
```

   Ohne die assert-Anweisung wäre das Programm einfach weiter gelaufen und hätte ein falsches Ergebnis geliefert.

   assert-Anweisungen sind nützlich, um schnell Bedingungen zu überprüfen, die erfüllt sein müssen, damit der folgende Programmcode sinnvoll arbeitet.

3. Lösung:

```
v = [ 1., -3., 0.4,  5.]
w = [ 2.,  2.,  3.,  3.]
z = []
s = 0

assert len(v) == len(w), 'Längen von v und w müssen übereinstimmen'
for i in range(len(v)):
    z.append(v[i] + w[i])
    s = s + v[i] * w[i]
print(z, s)
```

Listen in Listen

1. Führen Sie das folgende Programm aus. Was beobachten Sie?

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?

1. `list`-Objekte können beliebige Python-Objekte als Elemente beinhalten, auch `list`-Objekte. In Zeile 3 wird das `list`-Objekt mit Namen `l2` als weiteres Element an die Liste angehängt. Wie Namen sind auch die Einträge in `list`-Objekten lediglich Verweise auf andere Python-Objekte. Die `append`-Methode kopiert keinerlei Daten. Verändern wir in Zeile 6 also die Liste `l2`, sehen wir dies auch in `l`, das dieses `list`-Objekt als Element enthält.

2. Die `extend`-Method hängt die Elemente von `l2` als einzelne Elemente an `l` an. Da jedoch die Zuweisung in Zeile 6 nicht das Objekt verändert, auf den Eintrag `1` von `l2` verweist (dies ist auch gar nicht möglich, da `int`-Objekte unveränderlich sind), sondern lediglich Eintrag `1` von `l2` auf ein neues Objekt verweisen lässt (ein `int`-Objekt mit Wert 99), hat diese Zuweisung keine Auswirkungen auf die Liste `l`.

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.

Lösung:

```
def fakultaet(n):
    fak = 1
    for k in range(1, n+1):
        fak = fak * k
    return fak

def binom(n, k):
    fn = fakultaet(n)
    fk = fakultaet(k)
    fnk = fakultaet(n-k)
    return fn // (fk * fnk)

n = int(input('n = '))
k = int(input('k = '))
assert n >= k, 'n muss größer als k sein!'

c = binom(n, k)
print(c)
```

Lokale und globale Namen

1. Was passiert, wenn Sie folgendes Programm ausführen? (Nutzen Sie den Debugger!)

global.py

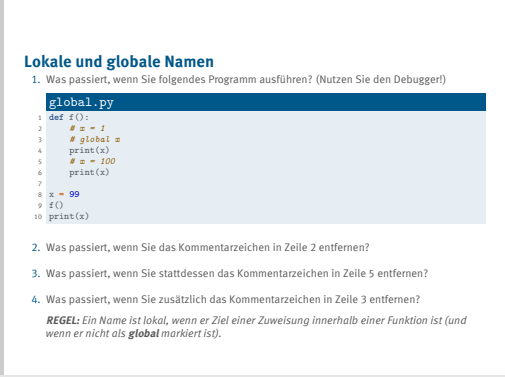
```
1  def f():
2      # x = 1
3      # global x
4      print(x)
5      # x = 100
6      print(x)
7
8  x = 99
9  f()
10 print(x)
```

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).*


Lokale und globale Namen

1. Was passiert, wenn Sie folgendes Programm ausführen? (Nutzen Sie den Debugger!)

global.py

```
1  def f():
2      # x = 1
3      # global x
4      print(x)
5      # x = 100
6      print(x)
7
8  x = 99
9  f()
10 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).*

1. Python hat, leicht vereinfacht, drei **Namensräume**, in denen Namen definiert sein können. Im `builtins`-Namensraum befinden sich alle von Python vordefinierten Namen wie `print` oder `input`. Im `globals`-Namensraum befinden sich die Namen, die Sie in Ihrem Programm definieren. Namen, die Sie innerhalb einer Funktion definieren, werden jedoch im `locals`-Namensraum abgelegt. Wann immer eine Funktion ausgeführt wird, wird ein neuer lokaler Namensraum angelegt, und nach Ausführung der Funktion wieder zerstört.

   In einer Funktion können Sie auch auf globale Namen zugreifen. Daher funktioniert das Programm, und Sie erhalten dreimal 99 als Ausgabe.

2. Wird einem Namen (hier `x`) innerhalb einer Funktion etwas zugewiesen, wird der Name automatisch lokal und überdeckt den möglicherweise vorhandenen gleichen globalen Namen. Insbesondere behält der globale Name `x` den Wert 99, und Sie erhalten 1, 1, 99 als Ausgabe.

3. Auch hier wird der Name `x` durch die Zuweisung in Zeile 5 automatisch ein lokaler Name in `f`, der in Zeile 4 jedoch noch nicht definiert wurde. Daher erhalten Sie eine Fehlermeldung.

   Es wäre auch denkbar, Python stattdessen einen Namen als global betrachten zu lassen *bis* ihm in der Funktion ein Wert zugewiesen wird. Das wäre jedoch selten nützlich und würde das Tor für viele schwer zu findende Programmierfehler öffnen.

4. Mit der `global`-Deklaration in Zeile 3 wird Python mitgeteilt, dass der Name `x` in `f` global sein soll, *obwohl* ihm zugewiesen wird. Damit ist `x` in Zeile 4 wieder definiert, und die Zuweisung in Zeile 5 ändert den Wert des globalen Namens `x`. Somit erhalten Sie als Ausgabe 99, 100, 100.

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 für 30, 1000, 40.

2. Führen Sie ihr Programm mit dem Debugger für `n = 7` aus. (Verwenden Sie immer `step into` (F7).)

3. Zählen Sie in einer globalen Variable `aufrufe` die einzelnen Aufrufe von `fib`, und geben Sie den Wert am Ende des Programms aus.

1. Rekursion, also eine Funktion sich selbst aufrufen zu lassen, ist ein elementares Konzept der Programmierung. Viele Algorithmen lassen sich elegant rekursiv ausdrücken. Auch in der Mathematik werden häufig rekursive (induktive) Definitionen verwendet.

   Funktionsaufrufe in Python sind jedoch, sowohl was Laufzeit als auch Speicherbedarf angeht, recht teuer, weshalb Python die maximale Rekursionstiefe auf einen recht niedrigen (jedoch änderbaren) Wert begrenzt. Jeder rekursive Algorithmus lässt sich in einen nicht-rekursiven Algorithmus umformulieren. Im Falle der Fibonaccifolge ist dies sehr einfach (siehe Aufgabe 1, Blatt 1).

2. Lösung:

```
def fib(n):
    global calls
    calls = calls + 1
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

n = int(input('n = '))
calls = 0
print(fib(n), calls)
```

   Die Zahl der Aufrufe von `fib` wächst exponentiell mit `n`. Für `n = 30` sind bereits 2.692.537 Funktionsaufrufe vonnöten.

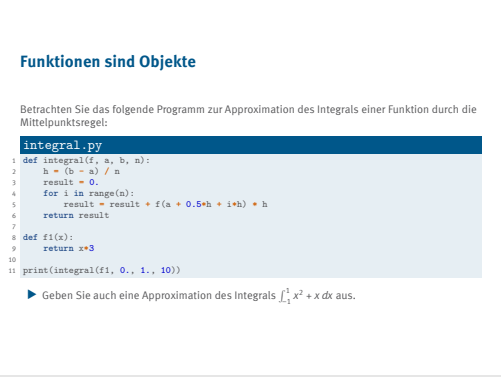
Funktionen sind Objekte

Betrachten Sie das folgende Programm zur Approximation des Integrals einer Funktion durch die 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.



- Die Mittelpunktsregel approximiert das Integral einer Funktion f von a bis b durch den Flächeninhalt des Rechtecks mit Seitenlängen $b - a$ und $f((b-a)/2)$. Um eine gute Approximation des Integrals zu erhalten, wird dabei der Integrationsbereich in viele (hier `n`) kleine Teilbereiche aufgeteilt.

- Funktionen sind Python-Objekte, die wie jedes andere Objekt auch als Argument einer Funktion übergeben werden können.

- Lösung:

```
def integral(f, a, b, n):
    h = (b - a) / n
    result = 0.
    for i in range(n):
        result = result + f(a + 0.5*h + i*h) * h
    return result

def f1(x):
    return x*3

def f2(x):
    return x**2 + x

print(integral(f1, 0., 1., 10))
print(integral(f2, -1, 1., 10))
```

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 print(integral(lambda x: x**4 + 1, 0., 1., 10))
```

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

lambda-Funktionen

Betrachten Sie das folgende Programm:

integrallambda.py

```
def integral(f, a, b, n):
    h = (b - a) / n
    result = 0.
    for i in range(n):
        result = result + f(a + 0.5*h + i*h) * h
    return result

print(integral(lambda x: x**4 + 1, 0., 1., 10))
```

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

- Lambda-Funktionen sind in Python lediglich von kosmetischem Nutzen, da diese immer durch eine `def`-Anweisung ersetzt werden können. Insbesondere zur Definition mathematischer Funktionen ist die kurze Schreibweise jedoch manchmal praktisch.

- Lösung:

```
def integral(f, a, b, n):
    h = (b - a) / n
    result = 0.
    for i in range(n):
        result = result + f(a + 0.5*h + i*h) * h
    return result

def konvergenztabelle(f, a, b, K):
    for k in range(K+1):
        print('n=10^', k, ' integral=', integral(f, a, b, 10**k))

konvergenztabelle(lambda x: x**4, 0., 1., 8)
```

- Die Konvergenz dieses Verfahrens ist nicht sehr schnell. In der Numerischen Analysis lernen Sie effizientere Integrationsmethoden kennen.

Module

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

pi.py

```
1  from random import uniform
2
3  N = int(input('N = '))
4  assert N > 0
5
6  treffer = 0
7  for i in range(N):
8      x = uniform(-1, 1)
9      y = uniform(-1, 1)
10     if x**2 + y**2 < 1:
11         treffer = treffer + 1
12
13 pi_approx = 4 * (treffer / N)
14 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.

- Das Monto-Carlo-Verfahren ist eine sehr einfache Methode zur Approximation von π, die sich auch auf zahlreiche andere Probleme anwenden lässt. Leider ist auch hier die Konvergenzgeschwindigkeit sehr niedrig.

- Mit `import` können Sie ein Erweiterungsmodul von Python-Objekten in den globalen oder lokalen Namensraum laden. Ein Objekt mit Namen `b` im Modul mit Namen `a` ist dann unter dem Namen `a.b` ansprechbar. Alternativ können Sie mit `from a import b` das Objekt `b` unter dem Namen `b` direkt in den jeweiligen Namensraum laden.

- Lösung

```
from random import uniform
import math

N = int(input('N = '))
assert N > 0
treffer = 0
for i in range(N):
    x = uniform(-1, 1)
    y = uniform(-1, 1)
    if x**2 + y**2 < 1:
        treffer = treffer + 1

pi_approx = 4 * (treffer / N)
print(pi_approx, pi_approx - math.pi)
```

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 Funktionsarguments 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` **(Dictionary, iterierbar)**

  Speichert ungeordnete Paare ‘Schlüssel -› Wert’ von **Objekt-Referenzen**. Konstruktion über Dictionary-Literale, z.B.

```
{}                              # Leeres Dictionary
{1: '1', 2: '2', 3: '3'}        # Dictionary mit 3 Einträgen
{1: f(1),                       # Dictionary-Literale dürfen beliebe Ausdrücke
 2: f(2)}                       # 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']                       # -> 2
d[4]                            # -> 16
d['4']                          # 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 hinzgefügt
```

- `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])   # Tupel-Literale dürfen beliebe 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.

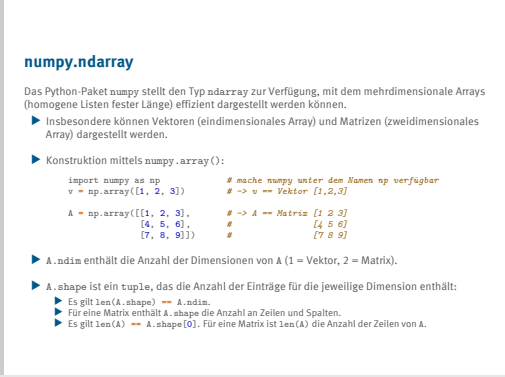
- Insbesondere können Vektoren (eindimensionales Array) und Matrizen (zweidimensionales Array) dargestellt werden.

- 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`.



Die Python-Bibliothek NumPy bietet Ihnen im Wesentlichen drei Vorteile:

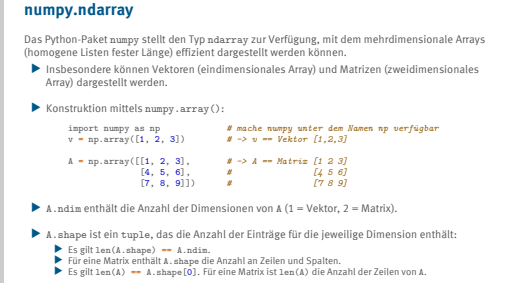
1. Da Sie mit `ndarray`-Objekten Operationen auf ganzen Vektoren oder Matrizen ausführen können, lassen sich viele grundlegende Operationen der linearen Algebra mit einem einzigen Befehl ausdrücken, während Sie mit Standard-Python aufwändig Schleifen programmieren müssten.

2. Als ‘general purpose’-Skriptsprache ist Python nicht auf die schnelle Verarbeitung numerischer Algorithmen ausgelegt. Auch wenn sich das bei kleineren Problemen nicht bemerkbar macht, können Schleifen mit Millionen von Durchläufen ihr Programm für größere Probleme schnell zum erliegen bringen. Vektorisierte NumPy-Operationen zu verwenden ist wesentlich effizienter.

3. NumPy bringt zahlreiche numerische Algorithmen mit, die Sie direkt nutzen können. NumPy ist auch die Basis für zahlreiche weitere Numerikbibliotheken für Python. Insbesondere SciPy bietet eine Vielzahl weiterer Algorithmen für verschiedenste Anwendungen.

- Beachten Sie, dass NumPy nicht mit Python mitgeliefert wird und separat installiert werden muss. Für Thonny gehen Sie dafür im Menü ‘Tools’ (‘Extras’) auf ‘Manage packages’, geben in das Textfeld ‘numpy’ ein und klicken auf ‘Find package from PyPI’ (Paket von PyPI suchen). Nachdem das Paket gefunden wurde, klicken Sie weiter unten auf ‘Install’ (‘Installiere’). Nach dem Ende der Installation können Sie den Dialog mit Klick auf ‘Close’ (‘Schließen’) wieder schließen.

- Es ist allgemein üblich, NumPy mittels `import numpy as np` zu importieren. Wenn Sie in fremden Code `np.xyz` lesen, können Sie davon ausgehen, dass mit `np` das `numpy`-Modul gemeint ist.



- **Während der gesamten NumPy-Einführung sollten Sie alle Beispiele direkt in die Python-Shell eingeben und mit den Objekten selbst experimentieren. Nur so bekommen Sie wirklich ein Gefühl für das Verhalten von NumPy!**

- Beginnen Sie direkt mit der Definition von `v` und `A`. Schauen Sie sich die `ndim`- und `shape`-Attribute beider Arrays an.

- Eine Matrix übergeben Sie dem `array`-Konstruktor als eine Liste von Listen gleicher Länge. Natürlich muss nicht jede Zeile der Matrix in eine neue Quellcodezeile geschrieben werden. Dies dient lediglich der Übersichtlichkeit.

- Anders als in Matlab, wo ein n-dimensionaler Vektor immer explizit ein Zeilen- oder Spaltenvektor (also eine $1 \times n$- oder eine $n \times 1$-Matrix) ist, werden Vektoren in NumPy üblicherweise als `array` mit `ndim == 1` dargestellt. Ein solches Array hat kein Konzept von Zeilen oder Spalten, da es wie eine `list` nur mit einer Zahl indiziert wird, und kann sowohl einen Zeilen- als auch einen Spaltenvektor darstellen.

Spezielle Arrays

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

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

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

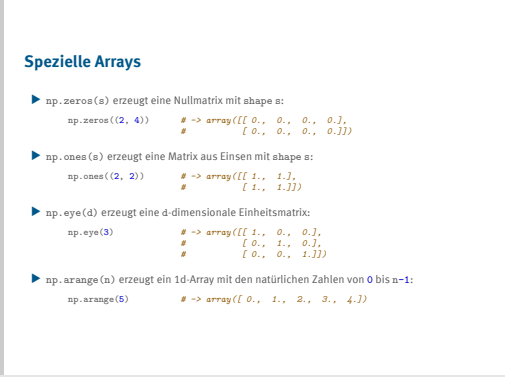
```
np.ones((2, 2))         # -> array([[ 1.,  1.],
                        #           [ 1.,  1.]])
```

- `np.eye(d)` erzeugt eine `d`-dimensionale Einheitsmatrix:

```
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`:

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



- Beachten Sie unbedingt die doppelten Klammern! Das gewünschte `shape` des Arrays muss als einzelnes Argument als `tuple` übergeben wird. Andernfalls erhalten Sie einen auf den ersten Blick nichtssagenden Fehler:

```
>>> import numpy as np
>>> np.ones(2, 2)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "/home/stephan/.virtualenvs/thonny/lib/python3.7/site-
packages/numpy/core/numeric.py", line 207, in ones
    a = empty(shape, dtype, order)
TypeError: data type not understood
```

  Wann immer Sie die Fehlermeldung `TypeError: data type not understood` lesen, sollten Sie zunächst davon ausgehen, dass Sie die zusätzlichen Klammern vergessen haben. Es ist auch erlaubt, anstelle eines `tuples` eine Liste zu verwenden:

```
np.ones([2, 2])
```

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 Norm der Einträge von `A`.

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 Norm der Einträge von `A`.

- Spielen Sie mit all diesen Operationen, und verstehen Sie was passiert!
- Was passiert z.B., wenn Sie versuchen Arrays verschiedener shapes zu addieren, zum Beispiel `np.ones((2,3)) + np.ones((3,3))`?

Indizierung / Slicing

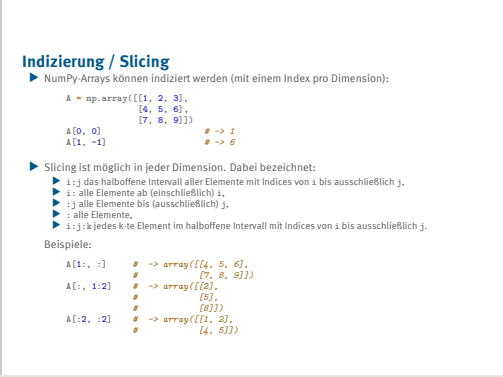
- NumPy-Arrays können indiziert werden (mit einem Index pro Dimension):

```
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,
  - `i:j:k` jedes k-te Element im halboffene Intervall mit Indices von `i` bis ausschließlich `j`.

  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]])
```



- Beachte: Wie bei `list` und `tuple` wird auch by NumPy-Arrays immer bei `0` angefangen zu zählen. Bei der Übertragung mathematischer Formeln müssen Sie daher häufig `1` von jedem Index abziehen.

- Wenn Sie sich unsicher fühlen, hilft es vielleicht, zunächst die Formel so umzuschreiben, dass mit dem Zählen bei `0` begonnen wird.

- Slicing ist eine wichtige Technik im Umgang mit NumPy-Arrays. Versuchen Sie, das Verhalten möglichst genau zu verstehen.

- Halboffene Intervalle beim Slicing machen es insbesondere einfach, ein Array in zwei Hälften aufzuteilen. Mit

  ```
  A1 = A[:4, :]
  A2 = A[4:, :]
  ```

  wird die Matrix `A` zerlegt in die Teilmatrix mit den ersten vier Zeilen und in die Teilmatrix mit den restlichen Zeilen.

- Durch Slicing ändert sich nie die Dimension eines Arrays. Insbesondere bleibt eine Matrix eine Matrix, auch wenn die Teilmatrix vielleicht nur eine oder sogar gar keine Zeilen/Spalten mehr hat.

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])
```

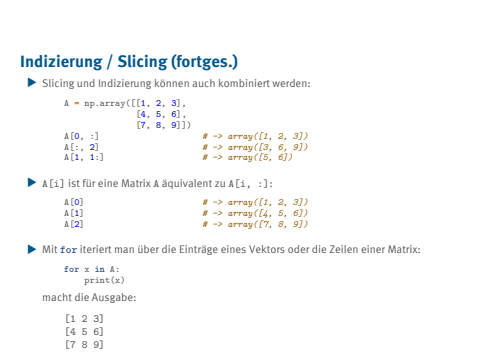
- Mit `for` iteriert man über die Einträge eines Vektors oder die Zeilen einer Matrix:

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

macht die Ausgabe:

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

Indizierung / Slicing (fortges.)



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]
```

- Das Indizieren mit einer Zahl reduziert die Dimension des Arrays stets um 1.

  `A[0, :]`

  liefert die erste Zeile von `A` als 1d-Array (Vektor), während

  `A[0, :]`

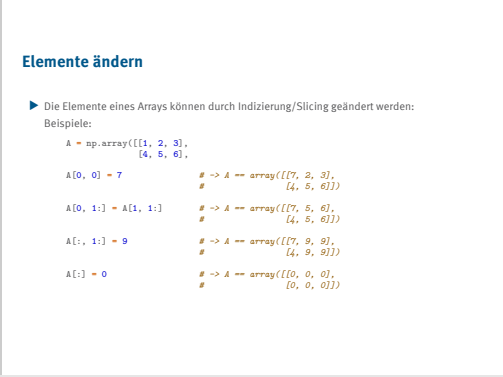
  ein 2d-Array (Matrix) mit einer Zeile ist.

- Merke: `A[i, :]` oder `A[i]` liefert Zeile `i` als Vektor, `A[:, j]` liefert Spalte `j` als Vektor.

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]])
```



- Generell müssen die `shapes` von beiden Seiten der Zuweisung übereinstimmen. Es ist daher überraschend, dass die letzten beiden Zuweisungen funktionieren. Der Grund dafür ist **Broadcasting**, das weiter unten diskutiert wird.

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 die ℓ^2-Norm von v.
- Berechnen Sie die ersten 10 Kubikzahlen ohne eine Schleife zu benutzen.
- Berechnen Sie $sin(x_n)$ für $x_n = 2\pi \frac{n}{N}$ mit $0 \leq n \leq N$, $N = 100$ ohne eine Schleife zu benutzen.

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 die ℓ^2-Norm von v.
- Berechnen Sie die ersten 10 Kubikzahlen ohne eine Schleife zu benutzen.
- Berechnen Sie $sin(x_n)$ für $x_n = 2\pi\frac{n}{N}$ mit $0 \leq n \leq N$, $N = 100$ ohne eine Schleife zu benutzen.

Lösungen:

```
import numpy as np

# 1. Aufgabe
A = np.array([[1, 2, 7, 4],
              [0, 9, 1, 1],
              [0, 0, 0, 6],
              [1, 1, 1, 1]])
v = np.array([1, 0, 1, 0])

# 2. Aufgabe
print(A.dot(v))

# 3. Aufgabe
A[0,  :] = v
A[:, -1] = 2*v
print(A)

# 4. Aufgabe
print(np.linalg.norm(v))
```

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 die ℓ^2-Norm von v.
- Berechnen Sie die ersten 10 Kubikzahlen ohne eine Schleife zu benutzen.
- Berechnen Sie $sin(x_n)$ für $x_n = 2\pi\frac{n}{N}$ mit $0 \leq n \leq N$, $N = 100$ ohne eine Schleife zu benutzen.

```python
# 5. Aufgabe
print((np.arange(10) + 1)**3)

# 6. Aufgabe
N = 100
xn = 2 * np.pi * np.arange(N+1) / N
print(np.sin(xn))
```

Views und Kopien

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

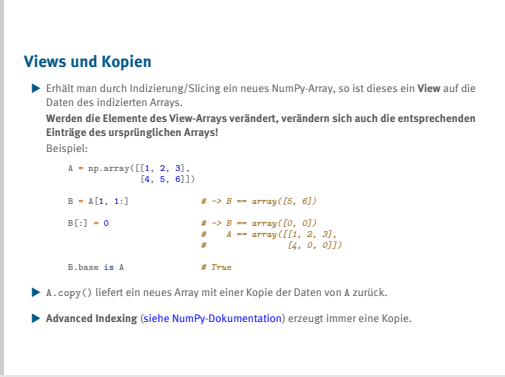
  Werden die Elemente des View-Arrays verändert, verändern sich auch die entsprechenden Einträge des ursprünglichen Arrays!

  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.



- Der wesentliche Vorteil von Views ist, dass bei ihrem Anlegen keine Daten kopiert werden. Das Anlegen eines Views geht daher sehr schnell und es wird kein zusätzlicher Speicher benötigt. Gerade beim Arbeiten mit großen Datenmengen kann dies ein entscheidender Vorteil sein.

- Gerade für Programmieranfänger sind Views aber auch eine Quelle für Programmierfehler. Wann immer sich überraschend Daten in Ihren Arrays verändern, sollten Sie überlegen, ob Sie unbewusst einen View verändern.

- Faustregel: Wann immer Sie das Ergebnis von Slicing/Indexing einem neuen Namen zuweisen, sollten Sie überlegen, ob das Array verändert werden kann/soll und ob Sie eine Kopie benötigen.

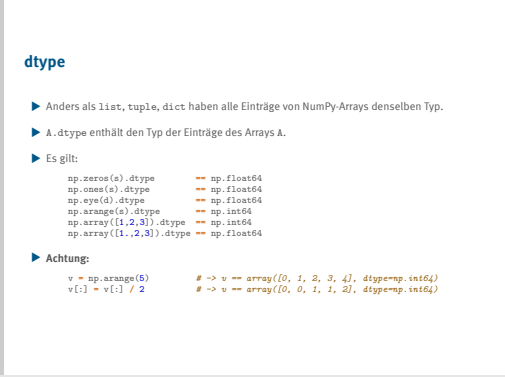
dtype

- Anders als `list`, `tuple`, `dict` haben alle Einträge von NumPy-Arrays denselben 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)
```



- Dass alle Elemente eines Arrays vom gleichen Typ sind, ist ein wesentlicher Grund für die Performance von NumPy. Damit ist, ohne weitere Fallunterscheidungen pro Element, sofort klar, wie die gewünschte Operation auf dem Element ausgeführt werden kann. Das ist auch der wesentliche Grund für die (oft) höhere Performance von statisch typisierten Sprachen wie C/C++ oder Fortran. Hier ist schon beim Schreiben des Programms festgelegt, welche Typen die Variablen (oder die Einträge von Arrays) im Programm haben.

- Außerdem sind die Elemente von NumPy-Arrays (in der Regel) keine Python-Typen, sondern hardwarenahe Datenformate, auf denen der Mikroprozessor direkt operieren kann.

- Wenn Sie Einträgen in einem `np.int64`-Array gebrochene Werte zuweisen, wird immer abgerundet! Der `dtype` des Arrays ändert sich nicht. Dies ist eine häufige Fehlerquelle. Insbesondere, wenn Sie das Gauß-Verfahren implementieren, müssen Sie darauf achten, dass ihre Systemmatrix den `dtype` `np.float64` hat. Wenn Sie ein NumPy-Array aus einer Liste (von Python Listen) konstruieren, können Sie dies z.B. erreichen, indem Sie mindestens einen `float` in der Liste verwenden.

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` is 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-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` is 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`.

- Achten Sie auf die zusätzlichen Klammern by `np.hstack` und `np.vstack`. Auch diesen Funktionen wird nur ein einziges Argument übergeben, nämlich eine Liste der Arrays, die aneinander gehängt werden sollen.

Array-Achsen / Broadcasting

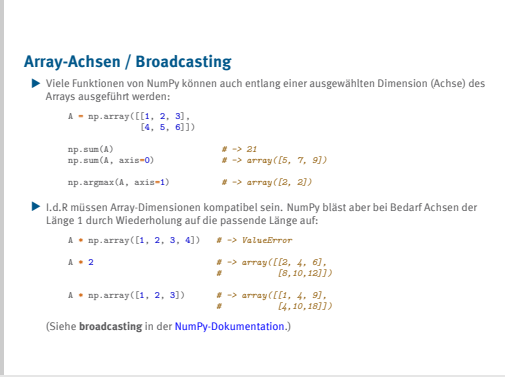
- Viele Funktionen von NumPy können auch entlang einer ausgewählten Dimension (Achse) des Arrays ausgeführt werden:

```
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.)



- Eine Matrix `A` (ein NumPy-Array mit `ndim == 2`) hat also zwei Achsen. In dem Ausdruck `A[i, j]` ist `i` der Index für Achse `0`, und `j` der Index für Achse `1`. Ein Vektor (ein NumPy-Array mit `ndim == 1`‘) hat nur eine Achse.

- Broadcasting ist ein mächtiges Konzept, mit dem sich viele Operationen elegant ausdrücken lassen. Es lohnt sich, die genauen Regeln zu verinnerlichen. Eine genaue Definition der Regeln finden Sie hier in der NumPy-Dokumentation.

- Broadcasting kann aber auch dazu führen, dass Programmierfehler später auffallen. Wird z.B. versehentlich ein falsches Array verwendet, können aufgrund von Broadcasting Operationen erfolgreich sein, die ohne Broadcasting zu einem Fehler führen würden, durch den es früher auffallen würde, dass ein falsches Array verwendet wurde.

==, 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!')
```

Mehr Fingerübungen

- Berechnen Sie die Summe der ersten 10.000 Kubikzahlen ohne eine Schleife zu verwenden. Messen Sie die Laufzeit. Vergleichen Sie die Laufzeit mit einem Schleifen-basierten Algorithmus.
- Berechnen Sie $A \cdot v$ ohne die `dot`-Methode. (Tipp: broadcasting)
- Finden Sie in der NumPy-Dokumentation einen Weg, die Determinante von A, $A^T \cdot v$ sowie $A^{-1} \cdot v$ zu berechnen.
- Legen ein NumPy-Array an, welches die Matrix

$$S \in \mathbb{R}^{11x11},\ S_{ij} = (i + j) \mod 2$$

enthält. (Tipp: reshape)

Mehr Fingerübungen

- Berechnen Sie die Summe der ersten 10.000 Kubikzahlen ohne eine Schleife zu verwenden. Messen Sie die Laufzeit. Vergleichen Sie die Laufzeit mit einem Schleifen-basierten Algorithmus.
- Berechnen Sie $A \cdot v$ ohne die dot-Methode. (Tipp: broadcasting)
- Finden Sie in der NumPy-Dokumentation einen Weg, die Determinante von A, $A^T \cdot v$ sowie $A^{-1} \cdot v$ zu berechnen.
- Legen ein NumPy-Array an, welches die Matrix

  $$S \in \mathbb{R}^{11 \times 11},\ S_{ij} = (i + j) \mod 2$$

  enthält. (Tipp: reshape)

Lösungen:

```
import numpy as np
A = np.array([[1, 2, 7, 4],
              [0, 9, 1, 1],
              [0, 0, 0, 6],
              [1, 1, 1, 1]])
v = np.array([1, 0, 1, 0])

# 1. Aufgabe
import time
tic = time.time()
s = np.sum(np.arange(1, 10001)**3)
print(s, time.time() - tic)

tic = time.time()
s = 0
for i in range(1, 10001):
    s = s + i**3
print(s, time.time() - tic)

# 2. Aufgabe
print(np.sum(A * v, axis=1))
```

Mehr Fingerübungen

- Berechnen Sie die Summe der ersten 10.000 Kubikzahlen ohne eine Schleife zu verwenden. Messen Sie die Laufzeit. Vergleichen Sie die Laufzeit mit einem Schleifen-basierten Algorithmus.
- Berechnen Sie $A \cdot v$ ohne die `dot`-Methode. (Tipp: broadcasting)
- Finden Sie in der NumPy-Dokumentation einen Weg, die Determinante von A, $A^T \cdot v$ sowie $A^{-1} \cdot v$ zu berechnen.
- Legen ein NumPy-Array an, welches die Matrix

  $$S \in \mathbb{R}^{11x11},\ S_{ij} = (i + j) \mod 2$$

  enthält. (Tipp: reshape)

```
# 3. Aufgabe
print(np.linalg.det(A))
print(A.T.dot(v))
print(np.linalg.solve(A, v))

# 4. Aufgabe
print((np.arange(11*11) % 2).reshape((11,11)))
```

Tag 5

Der ganze Rest

- Auf den folgenden Folien finden Sie einige bisher ausgelassene Details sowie einen Ausblick auf weitere Sprachfeatures und Erweiterungsbibliotheken.
- Der Stoff wird nicht für die Bearbeitung der Hausaufgabe benötigt.

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 beliebe 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'}
```

  `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 (`list`, `dict`, `set`) und Zeichenketten (`str`) 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 Operatoren 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 `list`-Objekte `l`, `l2` ist z.B. die Anweisung `l += l2` äquivalent zu `l.extend(l2)`.

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}
```

Iterable Unpacking

- Folgende Syntax erlaubt es, die Elemente eines Tupels oder eines anderen iterierbaren Objekts 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.

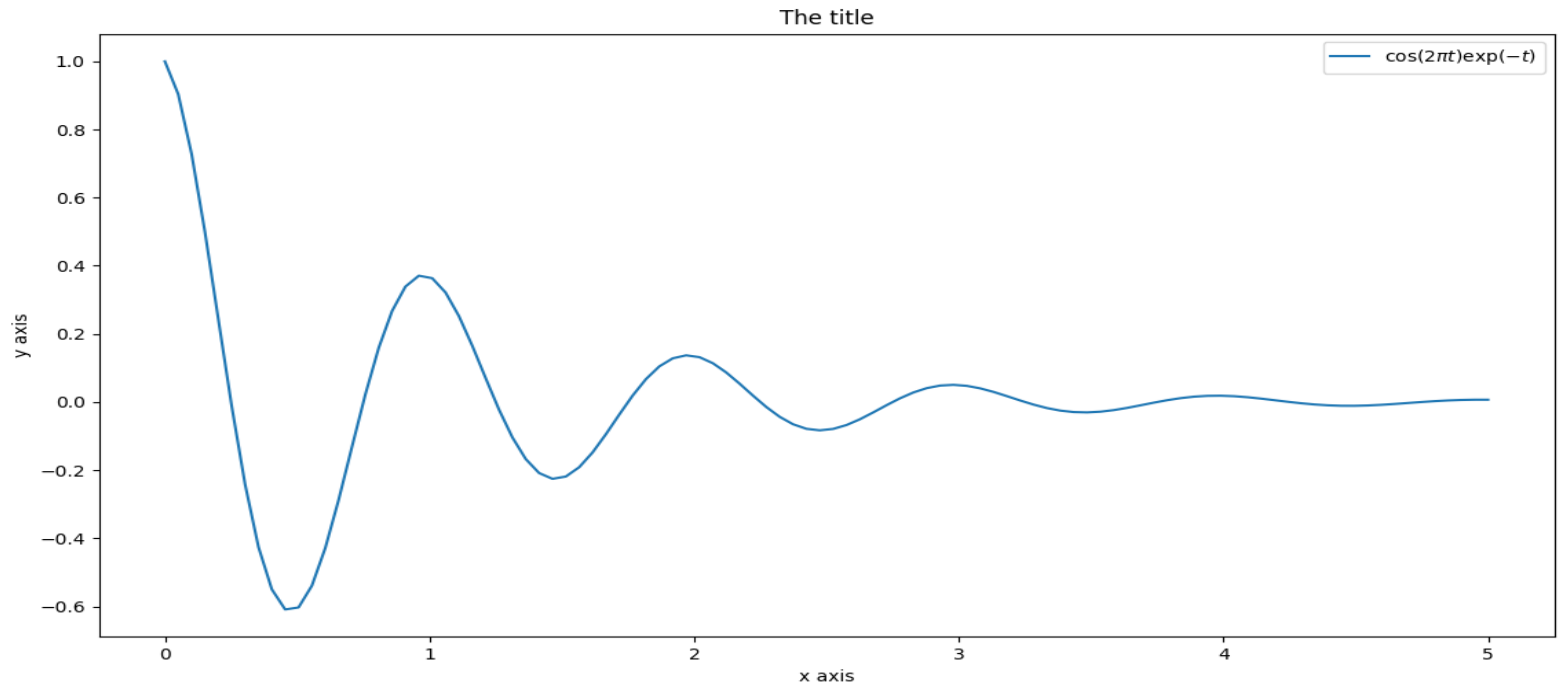
Ein Beispiel:

decay.py

```
1  import numpy as np
2  import matplotlib.pyplot as plt  # üblicherweise wird pyplot as plt importiert
3
4  x = np.linspace(0.0, 5.0, 100)                    # alle x-Werte für den Plot
5  y = np.cos(2 * np.pi * x) * np.exp(-x)                  # zugehörige y-Werte
6
7  plt.plot(x, y, label=r'$\cos(2 \pi t) \exp(-t)$')
8  plt.title('The title')
9  plt.xlabel('x axis')
10 plt.ylabel('y axis')
11 plt.legend()
12 plt.show()
```

Funktionsgraphen mit `matplotlib`

Das Ergebnis:



- Weiterführende Tutorials finden sich hier und hier.
- Oft ist es am einfachsten, ein vorhandenes Beispiel an die eigenen Bedürfnisse anzupassen.

Einige wichtige Bibliotheken

- 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
- TensorFlow, PyTorch : Künstliche neuronale Netze
- mpi4py: MPI-Bindings für Python
- dask: Verteilte Arrays und Task-Scheduling in HPC-Umgebungen
- jupyter: Interaktive Python-Notebooks im Webbrowser
- ...

Was sonst noch fehlt

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