mentoring

Freie Universität Berlin

Python Crashkurs – Teil 2

Mentoring – SoSe 2020

Anton Kriese
Elen Niedermeyer
Freie Universität Berlin
Fachbereich Mathematik und Informatik

20. Mai 2020

Inhalt

Inhalt

for-Schleife:

```
1  def sumAll(n):
2      sum = n
3      for i in range(n):
4          sum = sum+i
5          print(i)
6      return sum
```

Source Code 1: for-Schleife

for-Schleifen durchlaufen die Elemente der ihr übergebenen Liste. Der Zähler i wird in jedem Schleifendurchlauf verändert.
Besonderheit: Die Anzahl der Schleifendurchläufe steht zu Anfang fest.

while-Schleife:

```
1  def sumAll(n):
2      i, sum = 1, 0
3      while i <= n:
4          sum = sum+i
5          i = i+1
6      return sum
```

Source Code 2: while-Schleife

Solange die Bedingung erfüllt ist, führen **while-Schleifen** den Schleifenrumpf aus. Der Zähler *i* muss manuell verändert werden, damit die Schleife terminiert.
Besonderheit: Die Anzahl der Schleifendurchläufe braucht nicht festzustehen.

Liste

Listen sind veränderbare Sammlungen von Objekten verschiedener Datentypen. Sie sind als dynamische Arrays implementiert.

Vergleichsperatoren (Länge)	
`==`	Gleichheit
`!=`	Ungleichheit
`<,>`	Kleiner/größer als
`<=, >=`	Kleiner/größer gleich
Listenoperatoren	
`+`	Verkettung
`*`	Wiederholung
`[i]`	i-te Stelle
`[i:j]`	Teilliste i bis j-1
`in`	Test auf Enthaltensein
`not in`	Test auf Nichtenthaltensein

Table 1: Listen-Operatoren

```
1  a = [] # leere Liste
2  b = [1] # 1elementige L.
3  a = b+[2,3,4]
4  1 in a
5  type(a) #<class 'list'>
6  c = a
7  a[0] = '42' # Zuweisung
8  c # gibt c aus
9  4*[True]
```

Source Code 3: Listen-Operatoren

Liste

Um durch eine Liste zu laufen, kann eine *for*-Schleife verwendet werden:

```
1 for n in [1,2,3,4]:
2     print(n) # n nimmt jeden Wert aus [1,2,3,4] an
```

Source Code 4: For-Schleife

```
1 liste = [1,2,3,4] # len(liste) = 4
2 for i in range(0, len(liste)): # 4 ist exklusive
3     print(liste[i]) # i nimmt jeden Wert von 0 bis 3 an
```

Source Code 5: For-Schleife mit Index als Zähler

Inhalt

NumPy importieren

- Um *NumPy* nutzen zu können, ist folgende Zeile notwendig:

```
1 import numpy as np
```

Source Code 6: Numpy importieren

- Sollte *beim importieren* eine Fehlermeldung auftreten, ist *NumPy* vermutlich nicht installiert. *NumPy* lässt sich mit dem Befehl `pip3 install numpy` installieren.
- Sollte es bei der *Installation* Probleme geben, können unter

  https://www.lfd.uci.edu/~gohlke/pythonlibs/

  fehlende Pakete heruntergeladen werden. Diese lassen sich mit `pip3 install <Paketname>` installieren.

Matrix

- Eine $(n \times m)$-Matrix ist eine rechteckige Anordnung von Zahlen in m Spalten und n Zeilen:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \vdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix}$$

- Beispiele:

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \qquad A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- Eine (1×1)-Matrix entspricht einer Zahl. $(1 \times m)$- und $(n \times 1)$-Matrizen werden auch *Vektoren* genannt. Vektoren wird in der Regel ein Kleinbuchstabe zugewiesen (siehe Beispiel).

Matrix erstellen

- Eine Matrix wird mit dem Befehl *np.array(list)* erstellt, wobei *list* eine Liste ist, welche die Dimension und die Einträge einer Matrix vorgibt.
- Beispiele:

```
1  # Erzeugt eine (3 x 4)-Matrix
2  A = np.array([[3, 2, 1, 0],
3                [2, 1, 0, 0],
4                [1, 0, 0, 0]])
5  # Erzeugt einen Vektor ((1 x 3)-Matrix) mit 3 Einträgen
6  x = np.array([1, 2, 3])
```

Source Code 7: Wie man eine Matrix erstellt

Aufgabe A

Erstelle folgende Matrizen (im Terminal):

$$x = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \end{pmatrix} \qquad A = \begin{pmatrix} 1 & -2 & 5 \\ -2 & 7 & 3 \end{pmatrix}$$

Achtung! Lösung:

```
1  x = np.array(list(range(5)))
2  x = np.array([0, 1, 2, 3, 4])

4  A = np.array([[1, -2, 5],
5                [-2, 7, 3]])
```

Source Code 8: Lösung zur Aufgabe A

Eigenschaften einer Matrix

- Sei A eine Matrix.

Eigenschaften einer Matrix	
`A.dtype`	Datentyp der Elemente
`A.ndim`	Dimension der Matrix
`A.shape`	Gestalt der Matrix
`A.size`	Anzahl der Elemente
`A.shape[n]`	Anzahl der Elemente für die n-te Dimension

Table 2: Eigenschaften einer Matrix

Beispiele:

```
1 A = np.array([[1, 2, 3],
2                [4, 5, 6]])
3 A.dtype # dtype('int32')
4 A.ndim # 2
5 A.shape # (2, 3)
6 A.shape[0] # 2 Zeilen
7 A.shape[1] # 3 Spalten
8 A.size # 6
```

Source Code 9: Eigenschaften einer Matrix

Matrix - Operationen

- Für Matrizen gibt es eine Menge von Rechenoperationen.
- Matrizenaddition:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} + \begin{pmatrix} 0 & 2 \\ 1 & 3 \end{pmatrix} = \begin{pmatrix} 1+0 & 3+2 \\ 2+1 & 4+3 \end{pmatrix} = \begin{pmatrix} 1 & 5 \\ 3 & 7 \end{pmatrix}$$

- Matrizensubtraktion:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} - \begin{pmatrix} 0 & 2 \\ 1 & 3 \end{pmatrix} = \begin{pmatrix} 1-0 & 3-2 \\ 2-1 & 4-3 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

Matrix - Operationen

- Skalarmultiplikation:

$$\pi \cdot \begin{pmatrix} -1 & 0 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} \pi \cdot (-1) & \pi \cdot 0 \\ \pi \cdot 2 & \pi \cdot 1 \end{pmatrix} = \begin{pmatrix} -\pi & 0 \\ 2\pi & \pi \end{pmatrix}$$

- Matrixmultiplikation:

$$\begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 3 \cdot 1 + 2 \cdot 1 & 3 \cdot 1 + 2 \cdot 0 \\ 2 \cdot 1 + 1 \cdot 1 & 2 \cdot 1 + 1 \cdot 0 \end{pmatrix} = \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix}$$

- Transposition:

$$\begin{pmatrix} -2 & 3 \\ -1 & 4 \\ 0 & 5 \end{pmatrix}^T = \begin{pmatrix} -2 & -1 & 0 \\ 3 & 4 & 5 \end{pmatrix}$$

Matrix-Operationen

- Sei A eine $(n \times m)$-Matrix und B eine $(k \times l)$-Matrix. Sei c eine beliebige Zahl.

Matrixoperationen (unär)	
`A.T`	Transponiert die Matrix A
Matrixoperationen (binär)	
`A + B`	Matrizenaddition ($n = k$ und $m = l$)
`A - B`	Matrizensubtraktion ($n = k$ und $m = l$)
`A * B`	Komponentenweise Multiplikation ($n = k$ und $m = l$)
`c * A`	Skalarmultiplikation
`A @ B`	Matrizenmultiplikation (m = k)
`A / B`	Komponentenweise Division ($n = k$ und $m = l$)

Table 3: Rechenoperationen auf Matrizen

Matrix-Operationen


```
1  A = np.array([[1, 2, 3],
2                [4, 5, 6]])
3  B = np.array([[0, 1, -1],
4                [1, 0, 2]])
5  A + B
6  A - B
7  A * B # Komponentenweise Multiplikation
8  A / B # Komponentenweise Division
9  # Was ist das Ergebnis von A @ B?
10 # Sind die Ergebnisse unten alle gleich?
11 A @ B.T
12 B @ A.T
13 A.T @ B
14 B.T @ A
```

Source Code 10: Matrix-Operationen

Aufgabe B

Schreibe eine Funktion, die zwei Werte übergeben bekommt, diese quadriert und die Summe der Quadrate zurückgibt:

$$f(x, y) = x^2 + y^2$$

Achte darauf, dass die Funktion auch mit Matrizen als Eingabewerte genutzt werden kann (elementweise Berechnung von $f(X, Y)$).

$$X = \begin{pmatrix} 1 & 0 \\ \frac{1}{\sqrt{2}} & 1 \end{pmatrix}, Y = \begin{pmatrix} 0 & 1 \\ \frac{1}{\sqrt{2}} & 1 \end{pmatrix} \qquad f(X, Y) = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$$

Achtung! Lösung:

```
1 def quad_sum(x, y):
2     return x**2 + y**2
```

Source Code 11: Lösung zur Aufgabe B

Nützliche Matrizen

- Es gibt in *NumPy* mehrere vorgefertigte Funktionen, die euch häufig benötigte Matrizen erstellen.
- Die *Einheitsmatrix* E_n der Dimension n ist eine $(n \times n)$-Matrix mit der Eigenschaft

$$A \cdot E_n = A = E_n \cdot A$$

  für alle $(n \times n)$-Matrizen A.
- Beispiel für $n = 2$:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

- Eine solche Matrix lässt sich mit `np.eye(x)` erstellen, wobei x vom Typ `int` ist:

```
1 >>> np.eye(2)
2 array([[1., 0.],
3        [0., 1.]])
```

Source Code 12: Beispiel für `np.eye(x)`

Nützliche Matrizen

- Matrix mit Einsen an jeder Stelle: `np.ones(x)`. `x` ist hier ein Tupel oder ein nichtnegativer `int` und gibt die Dimension der Matrix an.

```
1 >>> np.ones(3)
2 array([1., 1., 1.])
3 >>> np.ones((2, 2))
4 array([[1., 1.],
5        [1., 1.]])
```

Source Code 13: Beispiel für `np.ones(x)`

- Ebenfalls gibt es einen Befehl, der eine Matrix mit einer Null an jeder Stelle erstellt: `np.zeros(x)`.

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

Source Code 14: Beispiel für `np.zeros(x)`

Aufgabe C

Erstelle folgende (7×7)-Matrix (im Terminal):

$$\begin{pmatrix} 3 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 3 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 3 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 3 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 3 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 3 \end{pmatrix}$$

Achtung! Lösung:

```
1  np.ones((7, 7)) + 2 * np.eye(7)
```

Source Code 15: Lösung zur Aufgabe C

Gitter erstellen

- In der Numerik wird oft ein *äquidistantes Gitter* benötigt. Für ein Intervall $I \subseteq [a, b] \neq \emptyset$ und $n \in \mathbb{N}$ ist das die Menge

$$\{a + k \cdot h \mid k = 0, 1, \ldots, n\}$$

  mit *Schrittweite* $h = (b - a)/n$.
- Für diesen Zweck gibt es zwei verschiedene Befehle: `np.arange(...)` und `np.linspace(...)`.
- Beispiel für `np.arange(...)`:

```
1 >>> np.arange(6) # Die 6 selbst ist exklusiv!
2 array([0, 1, 2, 3, 4, 5])
3 >>> np.arange(2, 8)
4 array([2, 3, 4, 5, 6, 7])
5 # Für nichtganzzahlige Schrittweiten np.linspace(...)
6 # besser verwenden.
7 >>> np.arange(1, 3, 0.25)
8 array([1.  , 1.25, 1.5 , 1.75, 2.  , 2.25, 2.5 , 2.75])
```

Source Code 16: Beispiel für `np.arange(...)`

Gitter erstellen

- Beispiel für *np.linspace(...)*:

```
>>> np.linspace(1, 3, 6)
array([1. , 1.4, 1.8, 2.2, 2.6, 3. ])
# Ohne drittes Argument werden 50 Gitterpunkte
# generiert
>>> np.linspace(1, 50)
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,
       10., 11., 12., 13., 14., 15., 16., 17., 18.,
       19., 20., 21., 22., 23., 24., 25., 26., 27.,
       28., 29., 30., 31., 32., 33., 34., 35., 36.,
       37., 38., 39., 40., 41., 42., 43., 44., 45.,
       46., 47., 48., 49., 50.])
```

Source Code 17: Beispiel für *np.linspace()*

Indizierung von Matrizen

- An dieser Stelle gehen wir von einem Vektor aus (= $(1 \times m)$- oder $(n \times 1)$-Matrix).

Indizierung von Vektoren	
`[i]`	i-ter Eintrag
`[:i]`	Teilvektor bis Position $(i-1)$
`[i:]`	Teilvektor ab Position i
`[i:j]`	Teilvektor ab Pos. i bis Pos. $(j-1)$
`[i:j:k]`	Jeder k-te Eintrag ab Pos. i bis Pos. $(j-1)$

Table 4: Zugriff auf Elemente

Beispiele:

```
1  a = np.array([2, 1, 3, 4,
       6, 5, 8, 7])
2  a[0] # Erstes Element
3  a[-1] # Letztes Element
4  a[5:] # [5, 8, 7]
5  a[:2] # [2, 1]
6  a[1::3] # [1, 6, 7]
7  a[2:6:2] # [3, 6]
8  a[:4:2] # [2, 3]
9  a[::3] # [2, 4, 8]
10 a[::-1] # a rückwärts
```

Source Code 18: Zugriff auf Elemente

Indizierung von Matrizen

- Nun nehmen wir eine $(n \times m)$-Matrix mit $n, m \geq 2$ an.

Indizierung von Matrizen	
`[i, j]`	Eintrag in i-ter Zeile und j-ter Spalte
`[i, :]`	Ganze i-te Zeile
`[:, j]`	Ganze j-te Spalte
`[:, :]`	Kopie einer Matrix

Table 5: Zugriff auf Elemente

Beispiele:

```
1 A = np.array([[1, 2, 3],
2                [4, 5, 6],
3                [7, 8, 9]])
4 # Eintrag in 3.ter Zeile,
5 # 2.ter Spalte
6 A[2, 1]
7 A[1, :] # Zweite Zeile
8 A[:, 0] # Erste Spalte
```

Source Code 19: Zugriff auf Elemente

Indizierung von Matrizen

Indizierung von Matrizen (Fortsetzung)	
`[i:j, :]`	i-te bis $(j-1)$-te Zeile
`[::k, :]`	Jede k-te Zeile
`[i:j:k, :]`	Jede k-te Zeile von der i-ten bis zur $(j-1)$-ten Zeile
`[i::k, :]`	Jede k-te Zeile ab der i-ten Zeile
`[:j:k, :]`	Jede k-te Zeile bis zur $(j-1)$-ten Zeile

Table 6: Zugriff auf Elemente

- Die oben angegebenen Indizierungen funktionieren analog für die Spalten.

Aufgabe D

Schreibe eine Funktion, die alle Werte einer Matrix multipliziert.
Achtung! Lösung:

```
1  def multiply_matrix(A):
2      mul = 1
3      for i in range(len(A)):
4          for j in range(len(A[i])):
5              mul *= A[i, j]
6      return mul;
7  # oder:
8  def multiply_matrix(A):
9      mul = 1
10     for row in A:
11         for j in row:
12             mul *= j
13     return mul;
```

Source Code 20: Lösung zur Aufgabe D

Indizierung von Matrizen

- Mit der Funktion *np.ix_(...)* lassen sich Matrizen sehr flexibel indizieren.
- Beispiel:

```
>>> A # erstellte Matrix
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25]])
>>> A[np.ix_([2, 4, 1], [0, 2])]
array([[11, 13],
       [21, 23],
       [ 6,  8]])
```

Source Code 21: Beispiel mit *np.ix_(...)*

Inverse einer Matrix

- *Wissen:* Für $x \in \mathbb{R} \setminus \{0\}$ gibt es ein eindeutig bestimmtes inverses Element x^{-1}, so dass

$$x \cdot x^{-1} = 1 = x^{-1} \cdot x.$$

.

- Für eine *reguläre*, also eine „ausreichend nette“, $(n \times n)$-Matrix A gibt es eine eindeutig bestimmte Inverse A^{-1}, so dass

$$A \cdot A^{-1} = E_n = A^{-1} \cdot A.$$

.

Inverse einer Matrix – Beispiel

- Seien

$$A = \begin{pmatrix} 3 & -1 \\ -2 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix}$$

- Dann ist

$$\begin{aligned} A \cdot B &= \begin{pmatrix} 3 & -1 \\ -2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix} \\ &= \begin{pmatrix} 3 \cdot 1 + (-1) \cdot 2 & 3 \cdot 1 + (-1) \cdot 3 \\ (-2) \cdot 1 + 1 \cdot 2 & (-2) \cdot 1 + 1 \cdot 3 \end{pmatrix} \\ &= \begin{pmatrix} 3 - 2 & 3 - 3 \\ -2 + 2 & -2 + 3 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = E_2 \end{aligned}$$

- Analog folgt $B \cdot A = E_2$. Wir erhalten $A^{-1} = B$.

- In *NumPy* lässt sich eine Matrix A mit `numpy.linalg.inv(A)` invertieren.

```
>>> A = np.array([[3, -1], [-2, 1]])
>>> B = np.linalg.inv(A) # Matrix wird invertiert!
>>> A
array([[ 3, -1],
       [-2,  1]])
>>> B
array([[1., 1.],
       [2., 3.]])
>>> A @ B
array([[ 1.0000000e+00, -4.4408921e-16],
       [ 0.0000000e+00,  1.0000000e+00]])
>>> B @ A
array([[ 1.0000000e+00,  0.0000000e+00],
       [-8.8817842e-16,  1.0000000e+00]])
```

Source Code 22: Invertieren einer Matrix

Lineare Gleichungssysteme

- Mit Matrizen lassen sich *lineare Gleichungssysteme* beschreiben.
- Beispiel: Das lineare Gleichungssystem

$$\begin{aligned} 4 \cdot x + 2 \cdot y &= 1 \\ x - y &= 1 \end{aligned}$$

lässt sich folgendermaßen mit einer Matrix beschreiben:

$$\begin{pmatrix} 4 & 2 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Multiplizieren wir beide Seiten mit der inversen Matrix, erhalten wir

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 4 & 2 \\ 1 & -1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1/6 & 2/6 \\ 1/6 & -4/6 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1/2 \\ -1/2 \end{pmatrix}.$$

- Lineare Gleichungssysteme mittels *np.linalg.inv(A)* zu lösen ist in der Praxis nicht empfehlenswert. Dafür verwendet man

  np.linalg.solve(A, b)

- Beispiel:

```
1  >>> A = np.array([[4, 2], [1, -1]])
2  >>> b = np.ones(2)
3  >>> x = np.linalg.solve(A, b)
4  >>> x
5  array([ 0.5, -0.5])
6  >>> A @ x
7  array([1., 1.])
```

Source Code 23: Lösen eines LGS

Inhalt

Mathematische Konstanten

Die Definition von π und e sind in *Math* und *NumPy* gleich.

Konstanten	
`np.e`	Eulersche Zahl
`np.pi`	Kreiszahl

Table 7: In *Numpy* definierte Konstanten

```
1 >>> np.e
2 2.71828182845904 5
3 >>> np.pi
4 3.14159265358979 3
```

Source Code 24: Konstanten

Trigonometrische Funktionen

- Im Gegensatz zu *Math*, können die trigonometische Funktionen in *NumPy* auf einzelne Werte und auf Matrizen angewandt werden.
- Die Funktionen werden bei Matrizen *komponentenweise* angewandt.

Trigonometrische Funktionen	
`np.sin(A)`	Sinus
`np.cos(A)`	Cosinus
`np.tan(A)`	Tangens
`np.arcsin(A)`	Arkussinus
`np.arccos(A)`	Arkuscosinus
`np.arctan(A)`	Arkustangens

Table 8: Trigonometrische Funktionen

```
1 A = np.array([[0.25,
      0.5], [0.75, 1]])
2 np.sin(A)
3 np.cos(A)
4 np.tan(A)
5 np.arcsin(A)
6 np.arccos(A)
7 np.arctan(A)
```

Source Code 25: Funktionen

Weitere Funktionen

Exponentialfunktion und Logarithmen	
`np.exp(A)`	Exponentialfunktion
`np.log(A)`	Natürlicher Logarithmus
`np.log2(A)`	Dualer Logarithmus
`np.log10(A)`	Dekadischer Logarithmus

Weitere Funktionen	
`np.sqrt(A)`	Quadratwurzel
`np.ceil(A)`	Aufrunden
`np.floor(A)`	Abrunden
`np.degrees(A)`	Wandelt Radiant in Grad für jeden Eintrag um
`np.radians(A)`	Wandelt Grad in Radiant für jeden Eintrag um
`A.cumsum()`	Bildet die *kumulative Summe* der Einträge

Table 9: Weitere Funktionen aus *NumPy*

Inhalt

Matplotlib importieren

- Zum Plotten von Daten und Funktionen nutzen wir *Matplotlib*:

```
1 import matplotlib.pyplot as plt
```

Source Code 26: Matplotlib importieren

- *Matplotlib* lässt sich mit `pip3 install matplotlib` installieren. Falls bei der Installation Probleme auftreten, lässt sich *Matplotlib* auch unter

  https://www.lfd.uci.edu/~gohlke/pythonlibs/

  herunterladen (vgl. Folie 9).

Erster Plot

- Mit den Funktionen `plt.plot(...)` und `plt.show()` lässt sich bereits ein erster simpler Plot erstellen.

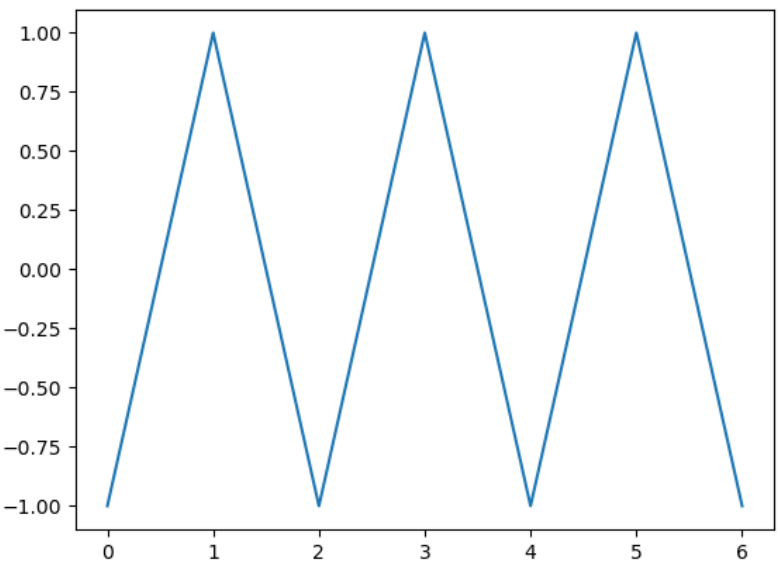

Figure 1: Hütchenfunktion

Code:

```
1 # Indizes dienen hier als
2 # Koordinaten für die
3 # x-Achse
4 plt.plot([-1, 1, -1, 1,
5           -1, 1, -1])
6 # Plot anzeigen
7 plt.show()
```

Source Code 27: Erster Plot

Plot mit Beschriftungen

- Die Funktionen `plt.xlabel(...)` und `plt.ylabel(...)` ermöglichen es die Beschriftungen an den Achsen festzulegen.
- Man kann mit `plt.title(...)` zusätzlich den Titel eines Plots festlegen.

Code:

```
1  # Indizes dienen hier als
2  # Koordinaten für die
3  # x-Achse
4  plt.plot([-1, 1, -1, 1,
5            -1, 1, -1])
6  # x-Achse beschriften
7  plt.xlabel("x-Werte")
8  # y-Achse beschriften
9  plt.ylabel("y-Werte")
10 # Titel festlegen
11 plt.title("Zweiter Plot")
12 # Plot anzeigen
13 plt.show()
```

Source Code 28: Beschriftungen festlegen

Plot mit Beschriftungen

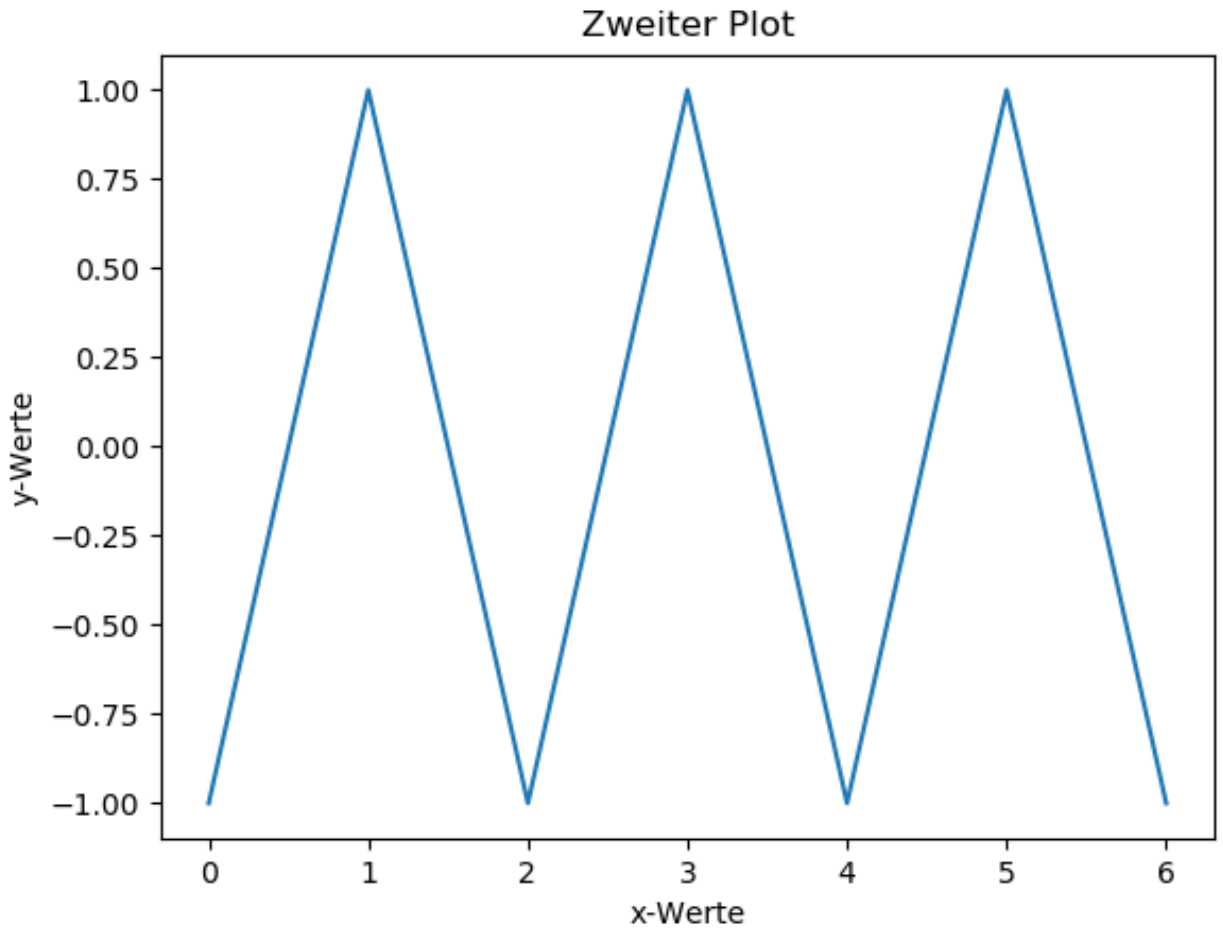

Figure 2: Plot mit Beschriftungen

x-Koor. & Mehrere Graphen

- Die x-Koordinaten lassen sich festlegen, indem man der Funktion `plt.plot(...)` einen zweiten Vektor übergibt. Während der erste Vektor die x-Werte festlegt, gibt der zweite Vektor die y-Werte an,
- Ruft man `plt.plot(...)` mehrfach vor dem Aufruf von `plt.show()` auf, dann erscheinen in einem Plot mehrere Graphen.
- Mit `plt.legend(...)` kann man eine Legende erstellen.

Code:

```
1  # Gitter erstellen
2  x = np.linspace(1, 2)
3  # Mehrere Graphen plotten
4  plt.plot(x, x)
5  plt.plot(x, np.exp(x))
6  plt.plot(x, np.log(x)-5)
7  # Legende erstellen
8  plt.legend(["x", "exp(x)"
       , "log(x) - 5"])
9  # Weitere Einstellungen
10 plt.xlabel("x")
11 plt.ylabel("y")
12 plt.title("Plot")
13 # Plot anzeigen
14 plt.show()
```

Source Code 29: Mehrere Graphen

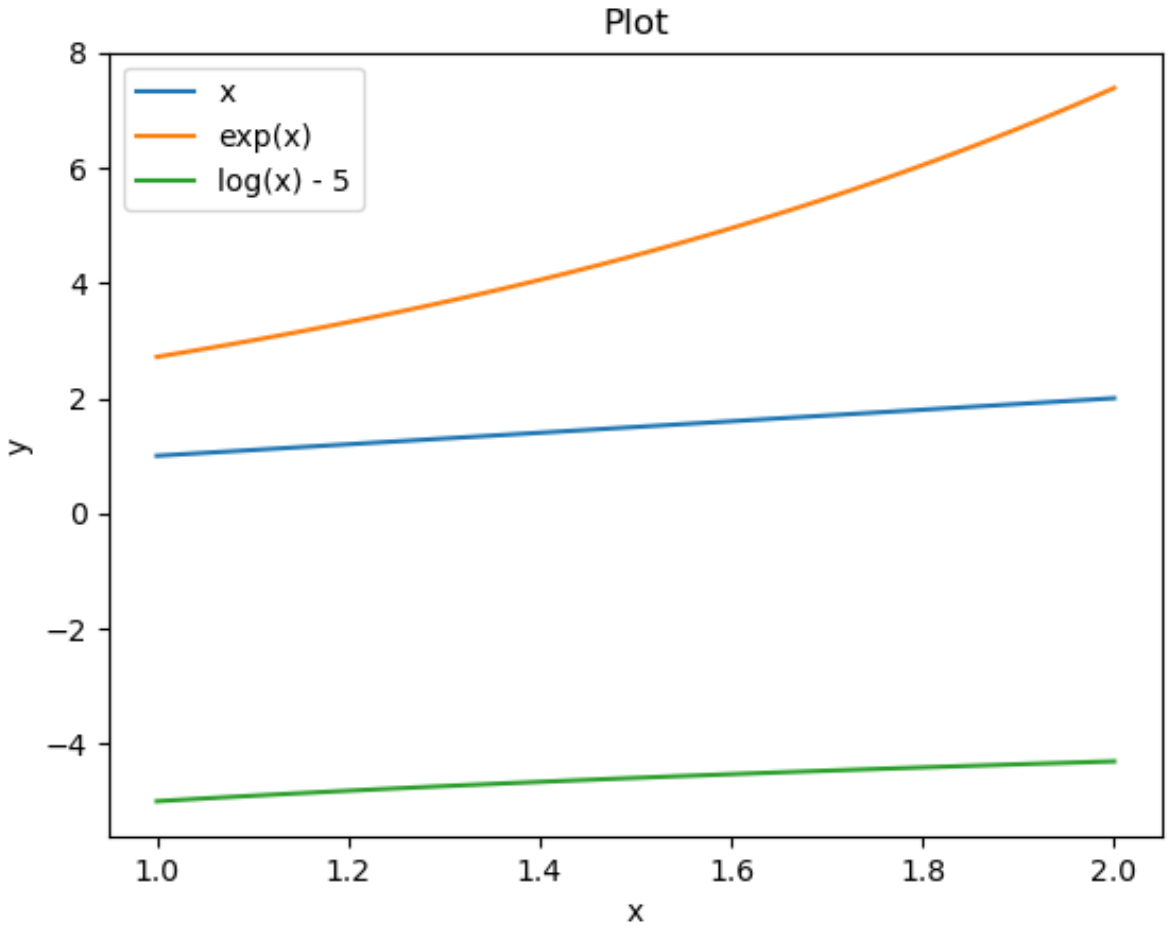

Figure 3: Mehrere Graphen in einem Plot mit Legende

- Die Zeilen

```
1 plt.plot(x, x)
2 plt.plot(x, np.exp(x))
3 plt.plot(x, np.log(x)-5)
```

Source Code 30: Mehrere `plt.plot(...)` Aufrufe

sind äquivalent zur Zeile

```
1 plt.plot(x, x, x, np.exp(x), x, np.log(x)-5)
```

Source Code 31: Einziger `plt.plot(...)` Aufruf

Aufgabe E

Erstelle einen Plot mit den Funktionen x^2 und x^3 in einem Koordinatensystem mit $2 \leq x \leq 5$. Beschrifte die Achsen und den Plot sinnvoll.
Achtung! Lösung:

```
1 x = np.linspace(2, 5)
2 plt.plot(x, x**2)
3 plt.plot(x, x**3)
4 plt.legend(["x^2", "x^3"])
5 plt.xlabel("x")
6 plt.ylabel("y")
7 plt.title("My Plot")
8 plt.show()
```

Source Code 32: Lösung zur Aufgabe E

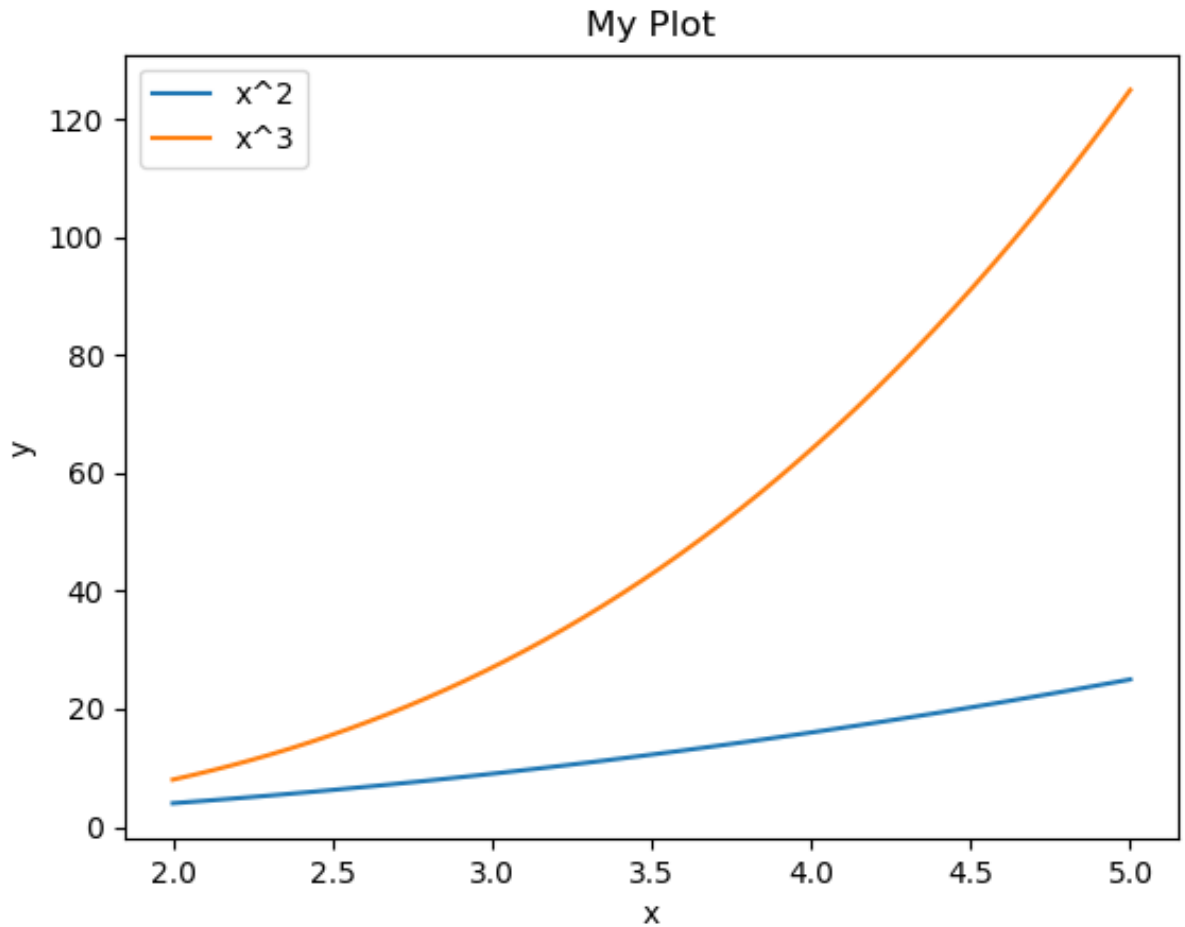

Figure 4: Lösung zur Aufgabe E

Näheres zu `plt.plot(...)`

- In `plt.plot(...)` lässt sich über einen *Format-String* der Form

  `'[marker][line][color]'`

  die Gestalt von einem Graphen festlegen.
- Beispiele:
  - `'.'` – Gepunktete Linie
  - `'g'` – Grüne Linie
  - `'xr'` – Rote Kreuze
  - `'D-y'` – Durchgezogene Linie mit gelben Diamanten
  - `'v--m'` – Margentafarbene gestrichelte Linie mit Dreiecken

Code:

```
1  x = np.linspace(-2, 2,
2                  10)
3  z = np.zeros(10)
4  plt.plot(x, z, ".")
5  plt.plot(x, x-4, "g")
6  plt.plot(x, x**2, "xr")
7  plt.plot(x, x**3, "D-y")
8  plt.plot(x, -3*x, "v--m")
9  plt.legend(["0", "$x-4$",
         "$x^2$", "$x^3$", "$
       -3\cdot x$"])
10 plt.show()
```

Source Code 33: Verschiedene Format-Strings

Näheres zu `plt.plot()`

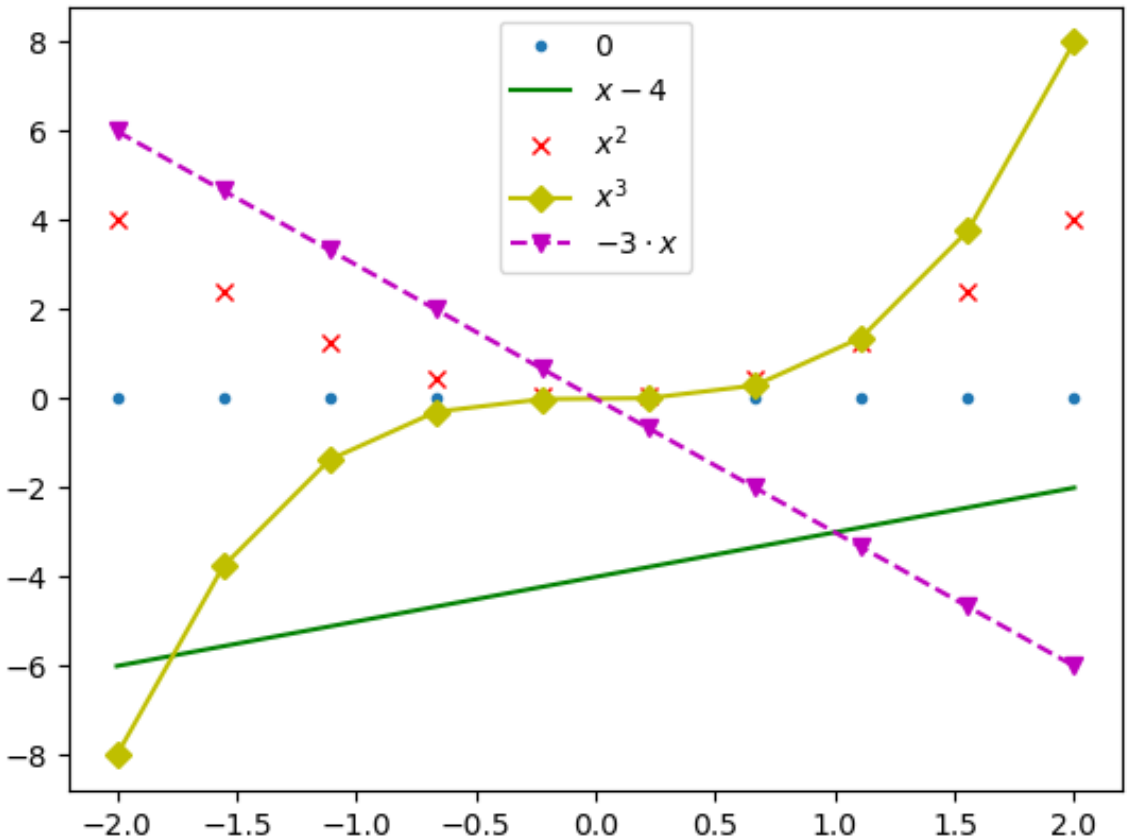

Figure 5: Farbiger Plot

Näheres zu `plt.plot()`

`fmt = '[marker][line][color]'`

marker	
`'.'`	Punkt für jeden Gitterpunkt
`'o'`	Kreis für jeden Gitterpunkt
`'s'`	Quadrat für jeden Gitterpunkt
`'*'`	Stern für jeden Gitterpunkt
`'x'`	Kreuz für jeden Gitterpunkt
`'D'`	Diamand für jeden Gitterpunkt
line	
`-`	Durchgezogene Linie
`--`	Gestrichelte Linie
`-.`	Strichpunktierte Linie
`:`	Gepunktete Linie

Table 10: Optionen für Format-String

Näheres zu `plt.plot()`

`fmt = '[marker][line][color]'`

color	
`'b'`	Blau
`'g'`	Grün
`'r'`	Rot
`'c'`	Cyan
`'m'`	Margenta
`'y'`	Gelb
`'k'`	Schwarz
`'w'`	Weiß

Table 11: Optionen für Format-String

Aufgabe F

Ändere in deinem Plot aus Aufgabe E (x^2 und x^3 mit $2 \leq x \leq 5$) die Farben. x^3 soll eine gestrichelte Linie werden.
Achtung! Lösung:

```
1 x = np.linspace(2, 5)
2 plt.plot(x, x**2, "-g")
3 plt.plot(x, x**3, "--c")
4 plt.legend(["x^2", "x^3"])
5 plt.xlabel("x")
6 plt.ylabel("y")
7 plt.title("My Plot")
8 plt.show()
```

Source Code 34: Lösung zur Aufgabe F

Aufgabe F

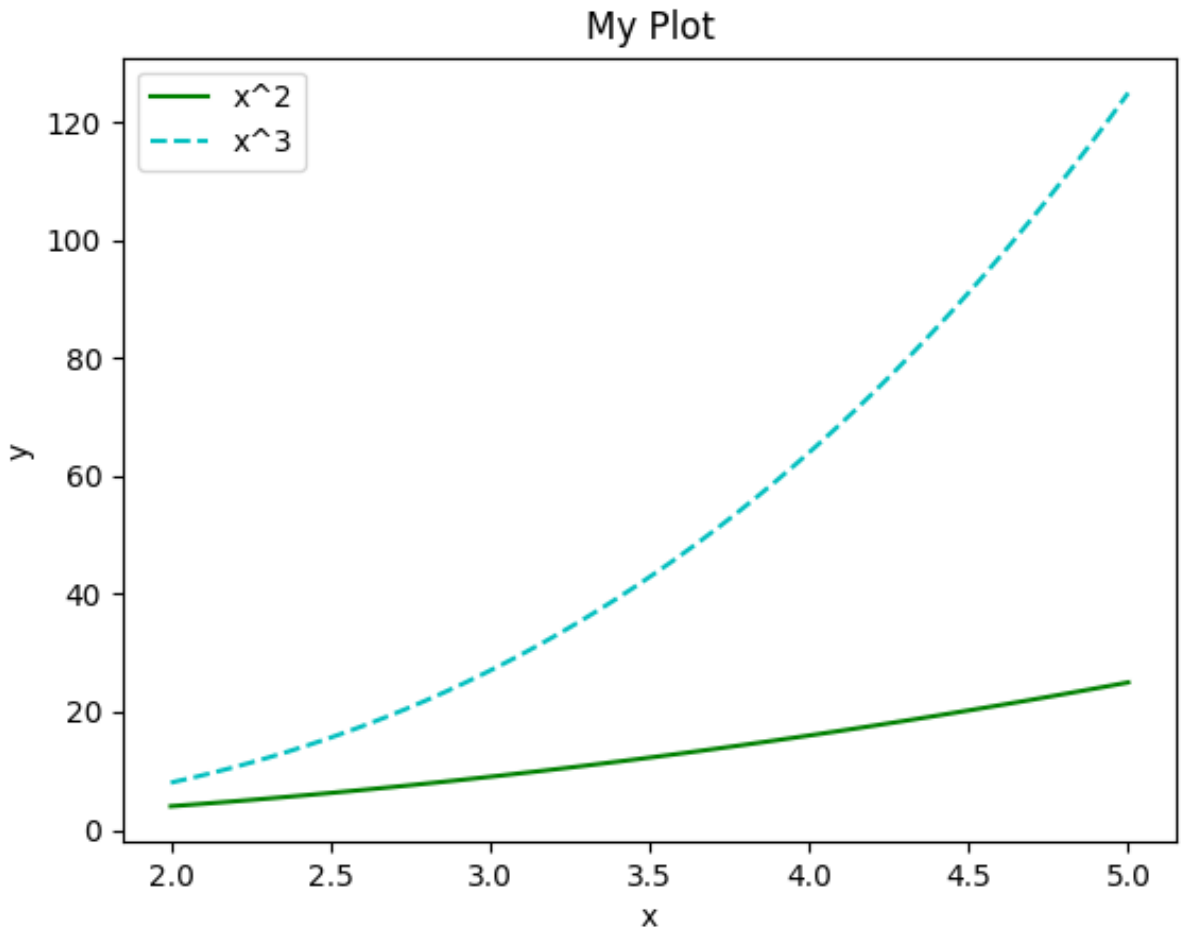

Figure 6: Lösung zur Aufgabe F

Weitere Plot-Funktionen

- Zur besseren Darstellung eines Graphen bietet es sich in manchen Situationen an die *x*- oder *y*-Achse *logarithmisch* zu skalieren.
- Den Funktionen in der Tabelle kann ein *Format-String* übergeben werden.

color	
`plt.semilogx(...)`	Plot mit logarithmisch skalierter *x*-Achse
`plt.semilogy(...)`	Plot mit logarithmisch skalierter *y*-Achse
`plt.loglog(...)`	Plot mit logarithmisch skalierter *x*- und *y*-Achse

Table 12: Nahe Verwandten von `plt.plot(...)`

- Beispiel:

```
1  x = np.arange(20)
2  y = np.array([np.math.factorial(n) for n in x])
3  plt.semilogy(x, y, "o-r")
4  plt.show()
```

Source Code 35: Beispiel für `plt.semilogy(...)`

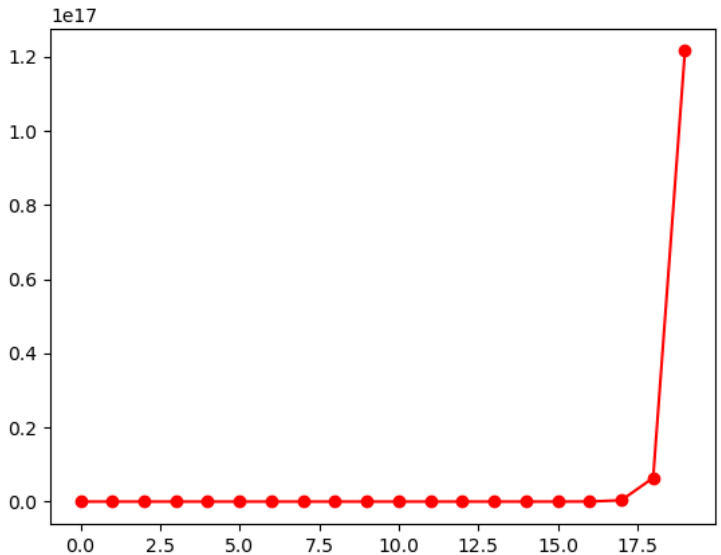

Figure 7: Fakultätsfunktion mit `plt.plot(x, y)`

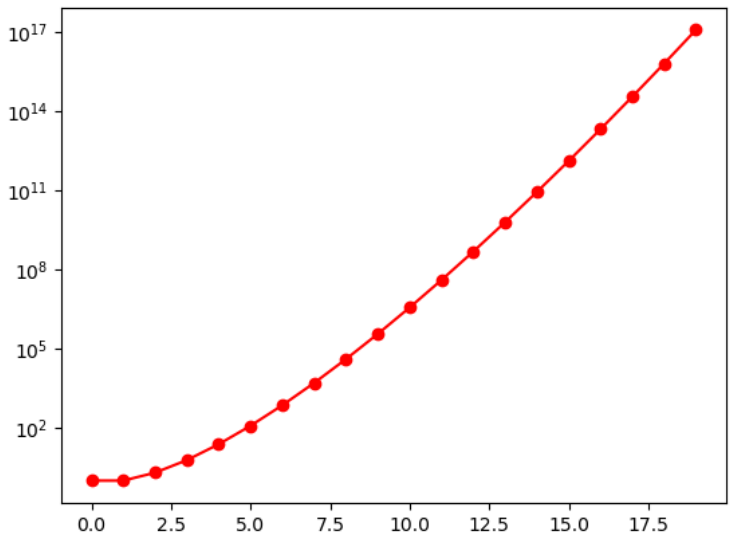

Figure 8: Fakultätsfunktion mit `plt.semilogy(x, y)`

Gitter

- In einem Plot lässt sich mit `plt.grid(...)` ein Gitter anzeigen.

Code:

```
1 x = np.linspace(-50, 50)
2 plt.grid()
3 plt.plot(x, x)
4 plt.show()
```

Source Code 36: Gitter anzeigen

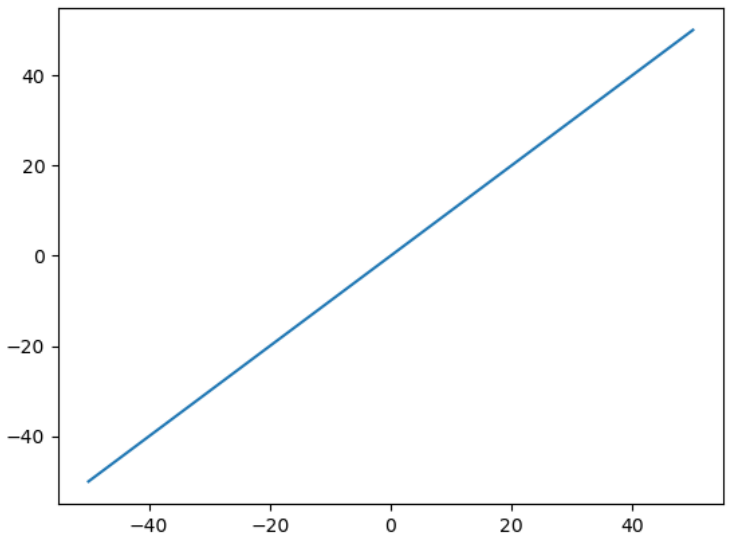

Figure 9: Ohne *plt.grid()*

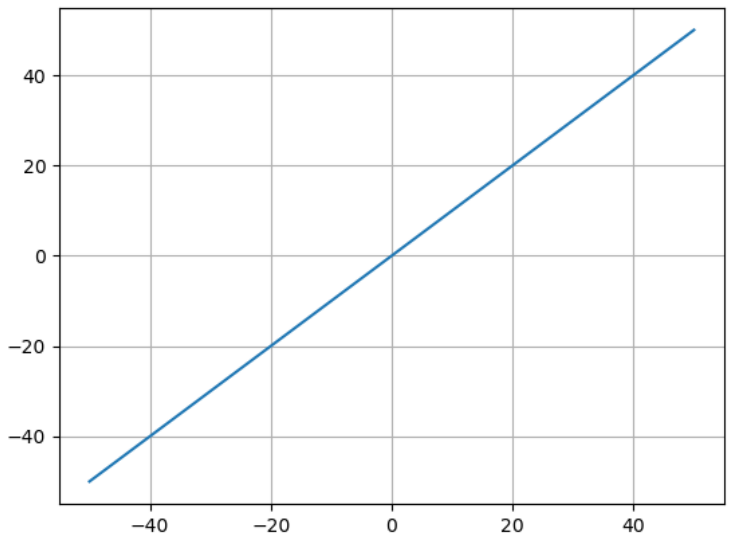

Figure 10: Mit *plt.grid()*

Skalen festlegen

- Die Achsenskalierungen lassen sich mit der Funktion

  `plt.axis(...)`

  festlegen.

Code:

```
1  x = np.linspace(-1, 1)
2  y = np.exp(x)
3  # x-Achse geht von
4  # -2 bis 2 y-Achse
5  # geht von 0 bis 3
6  xmin, xmax = -2, 2
7  ymin, ymax = 0, 3
8  plt.axis([xmin, xmax,
9            ymin, ymax])
10 plt.plot(x, y)
```

Source Code 37: Achsenskallierung

Skalen festlegen

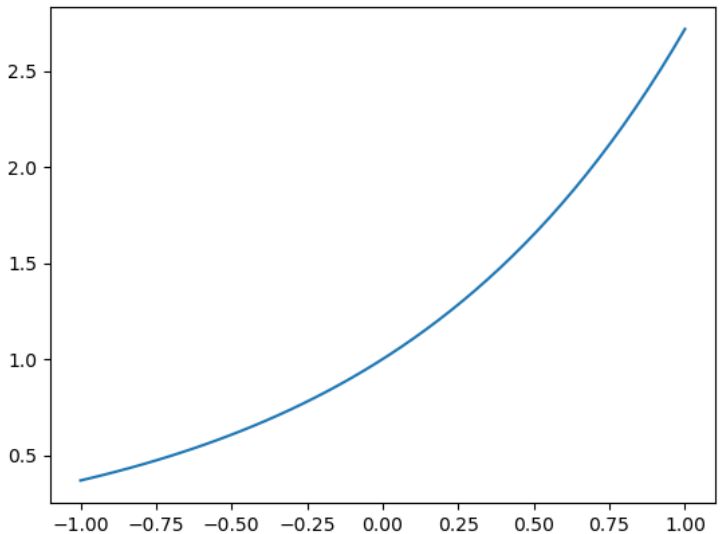

Figure 11: Ohne `plt.axis([-2, 2, 0, 3])`

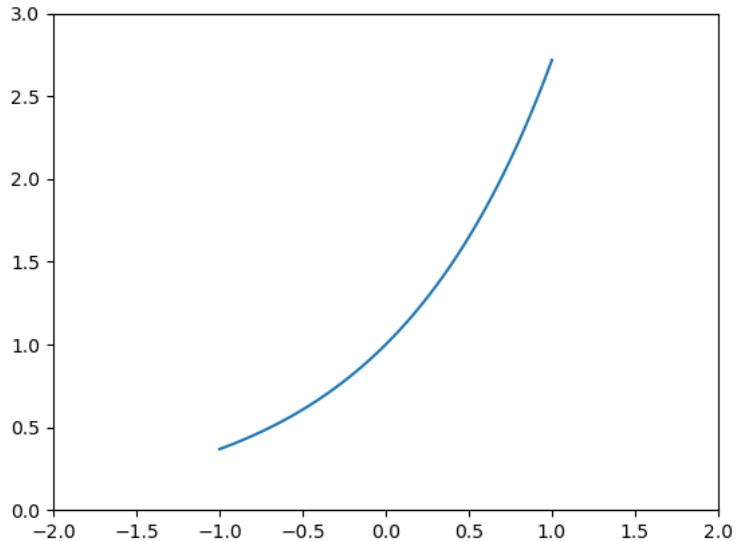

Figure 12: Mit `plt.axis([-2, 2, 0, 3])`

Subplots

- Für mehrere Subplots in einem Plot benutzt man `plt.subplot(xyz)`.
- Die Subplots werden über eine Ziffernfolge *xyz* gesteuert, wobei
  - x = Anzahl von Spalten,
  - y = Anzahl von Zeilen und
  - z = Nummer des Subplots

  meint.
- Beispiel: Möchte man ein (2×3)-Schema von Plots, muss $x = 2$ und $y = 3$ sein. Möchte man den fünften Subplot ansprechen, muss man $z = 5$ setzen. Mit dem Befehl `plt.subplot(235)` kann man den fünften Subplot im (2×3)-Schema bearbeiten.

Subplots


```
1  x = np.linspace(0, 1, 200)
2  plt.subplot(121)
3  plt.plot(x, x**2, "r", x, np.sqrt(x), "g")
4  plt.title("Parabel und Wurzel")
5  plt.xlabel("x")
6  plt.ylabel("Wert")
7  plt.legend(["$x^2$", "sqrt(x)"])
8  plt.grid()
9  plt.subplot(122)
10 plt.plot(x, np.exp(x), "-.m")
11 plt.title("Exponentialfunktion")
12 plt.xlabel("x")
13 plt.ylabel("Wert")
14 plt.legend(["exp(x)"])
15 plt.grid()
16 plt.show()
```

Source Code 38: Beispiel für *plt.subplot(...)*

Subplots

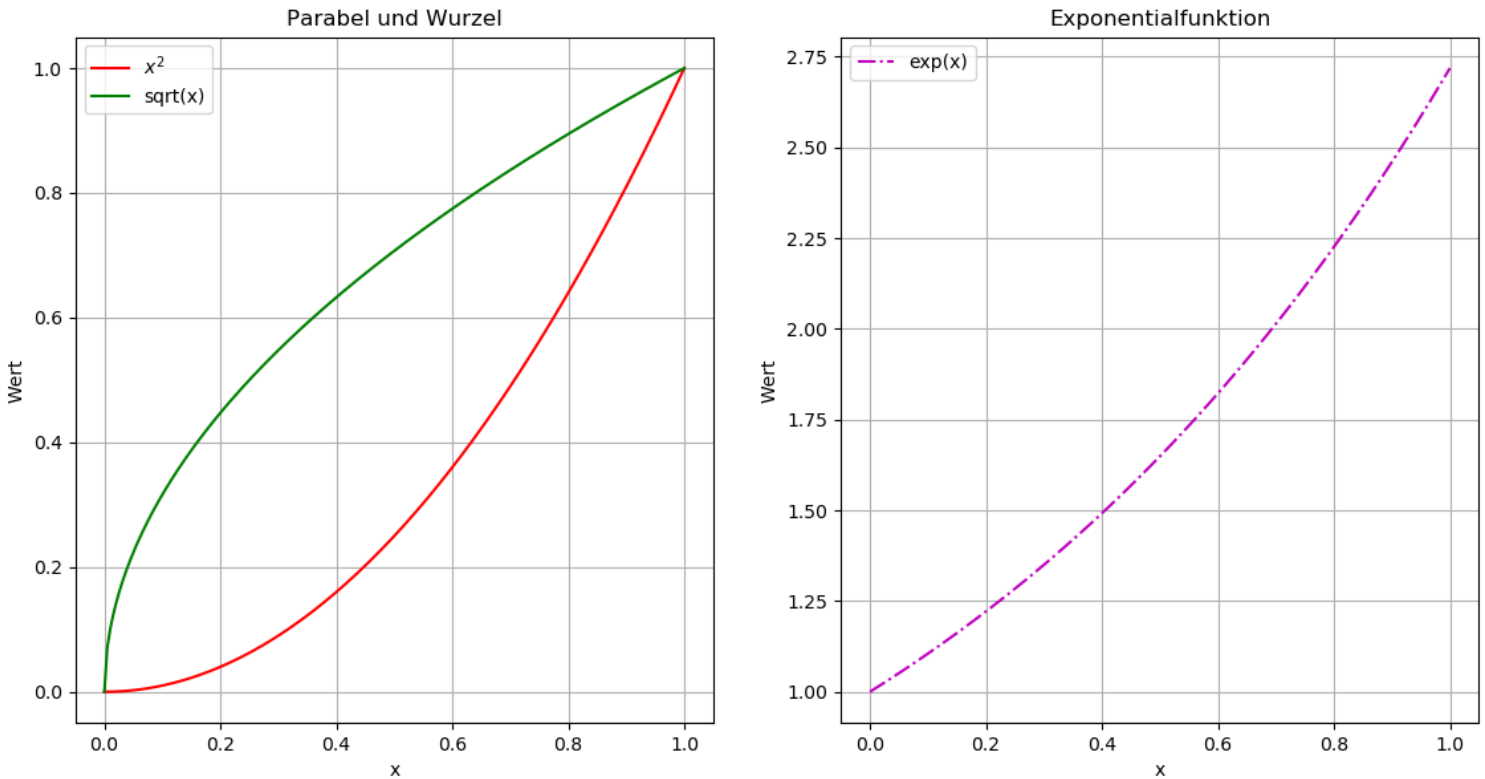

Figure 13: Subplots

Aufgabe G

Extrahiere deine Graphen aus Aufgabe F (x^2 und x^3 mit $2 \leq x \leq 5$) in zwei nebeneinanderstehende Subplots.
Achtung! Lösung:

```
1  x = np.linspace(2, 5)
2  plt.subplot(121)
3  plt.plot(x, x**2, "-g")
4  plt.legend(["x^2"])
5  plt.xlabel("x")
6  plt.ylabel("y")
7  plt.title("My Plot 1")
8  plt.subplot(122)
9  plt.plot(x, x**3, "--c")
10 plt.legend(["x^3"])
11 plt.xlabel("x")
12 plt.ylabel("y")
13 plt.title("My Plot 2")
14 plt.show()
```

Source Code 39: Lösung zur Aufgabe G

Aufgabe G

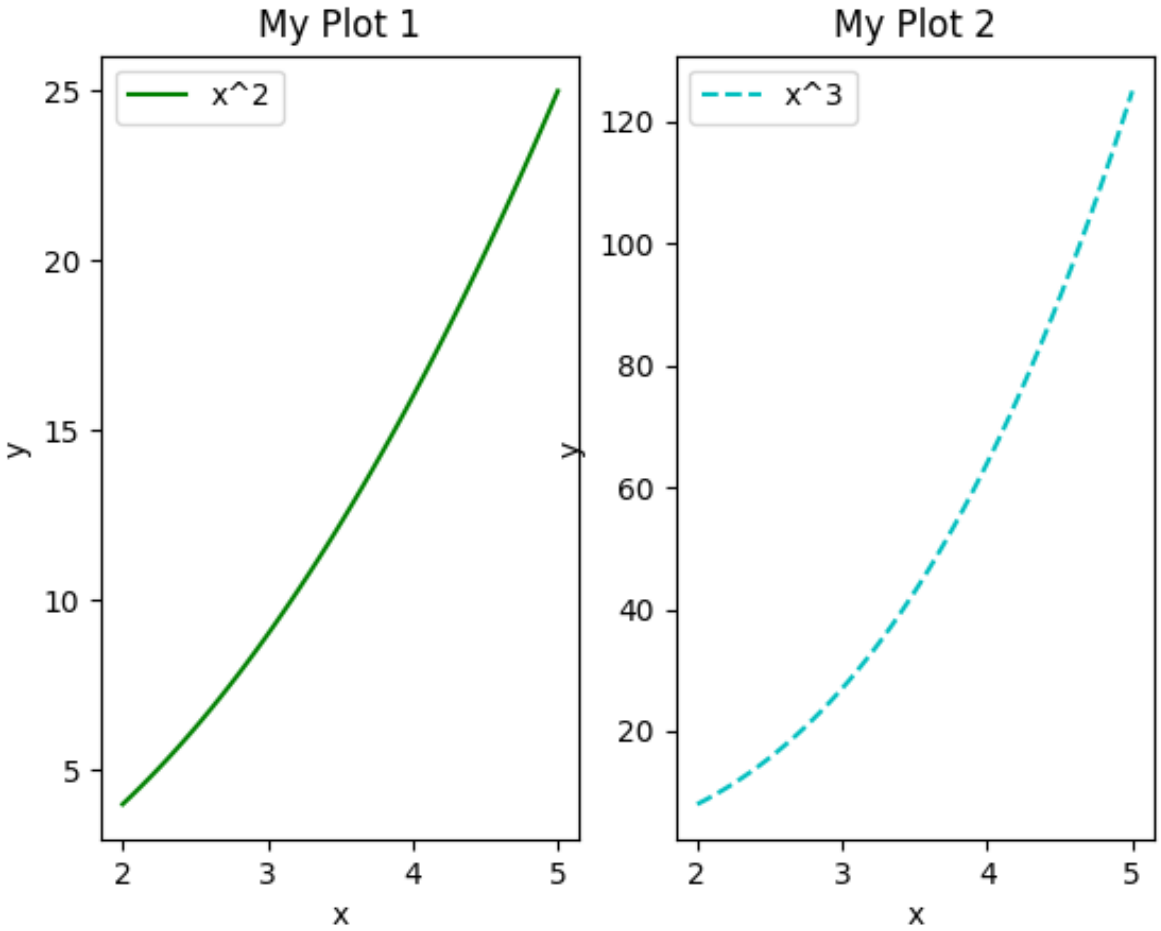

Figure 14: Lösung zur Aufgabe G

Noch Fragen?