Python Tutorial

Klaus Rheinberger

23. März 2025

Vorwort

Dieses Tutorial zeigt einführend, wie Python zum wissenschaftlichen Rechnen verwendet werden kann. Ausführlichere Einführungen finden Sie z. B. in:

- Python Language Companion to the excellent and free book Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares by Stephen Boyd and Lieven Vandenberghe, Cambridge University Press, 2018.
- Programming for Computations - Python. A Gentle Introduction to Numerical Simulations with Python 3.6. von Svein Linge und Hans Petter Langtangen, 2. Auflage, 2020. => PDF-Download im FHV-Netz möglich!

Wer Python von Grund auf und umfassender lernen möchte, kann z. B. diese Bücher und Links verwenden:

- Python - Der Grundkurs von Michael Kofler, 2. Auflage, 2021
- Official Python 3 documentation
- Python Tutorial bei www.w3schools.com

Python als Taschenrechner

In der Python-Shell können Sie intuitiv Rechnungen eintippen. Hier ein paar Beispiele:

```
(42 + 137)/0.815
```

```
219.6319018404908
```

Achtung: Das Dezimaltrennzeichen ist im Englischen ein Punkt, und Potenzieren wird mit ** implementiert!

```
1.23**45
```

```
11110.40818513195
```

Um die elementaren mathematischen Funktionen zu verwenden, müssen Sie zuerst, und nur einmal, das NumPy-Paket importieren. Wir verwenden das Kürzel `np`, um die Funktionen von NumPy anzusprechen.

```
import numpy as np

np.sqrt(np.pi/2)
```

```
np.float64(1.2533141373155001)
```

Kommentare beginnen mit einem `#`.

```
# This is a comment.
np.log(1)  # This also.
```

```
np.float64(0.0)
```

Formatierte Ausgabe

Für eine formatierte Ausgabe verwenden wir die Funktion `print` und sogenannte f-Strings:

```
name  = "Klaus"
like_math = True
age = 50
value = 12.3456789

print(f"Hallo! My name is {name}.")
print(f"I {like_math = }")
print(f"I am {age} years old. These are about {age*365} days.")
print(f"A value: {value:.2f}")  # float with two decimal places
```

```
Hallo! My name is Klaus.
I like_math = True
I am 50 years old. These are about 18250 days.
A value: 12.35
```

Listen und Vektoren

In der Mathematik ist ein Vektor des $\mathbb{R}^n$ eine geordnete Liste von n reellen Zahlen. In Python werden allgemeine Listen (nicht nur von Zahlen) mit eckigen Klammern erzeugt:

```
a = [-3, 5.7, 8]
b = [2.1, 0, -8]
c = ["Karl", "Klaus", "Erna", "Maria"]
```

Allerdings ist z. B. die Addition von Python-Listen mit Zahleneinträgen **nicht** wie die Addition von Vektoren in der Mathematik, nämlich elementweise, definiert:

```
a + b
```

```
[-3, 5.7, 8, 2.1, 0, -8]
```

Um den gewünschten Effekt von Addition und Skalarmultiplikation für Vektoren zu erreichen, machen wir solche Python-Listen zu 1-dimensionalen NumPy-Arrays:

```
v = np.array([-3, 5.7, 8])
w = np.array([2.1, 0, -8])

print(v + w)
print(3*w)
```

```
[-0.9  5.7  0. ]
[  6.3   0.  -24. ]
```

Es gibt einige hilfreiche NumPy-Funktionen, um oft gebrauchte Typen von NumPy-Arrays zu erzeugen, z. B.:

```
# 11 evenly spaced numbers over the interval [0, 50]:
x = np.linspace(0, 50, 11)
print(x)
```

```
[ 0.  5. 10. 15. 20. 25. 30. 35. 40. 45. 50.]
```

```
# evenly spaced numbers over the interval [0, 20] with step size 2:
x = np.arange(0, 20, 2)  # Note, that 20 is not included!
print(x)
```

```
[ 0  2  4  6  8 10 12 14 16 18]
```

NumPy-Arrays

Ein Vektor wird als eindimensionales NumPy-Array implementiert, eine Matrix als zweidimensionales NumPy-Array.

```
v = np.array([-3, 5.7, 8])

print(f"{v}")
print(np.ndim(v)) # number of dimensions
print(len(v))     # length, i. e. number of items, not the geometric length!
print(v.size)     # also number of items
```

```
[-3.   5.7  8. ]
1
3
3
```

```
# A matrix is filled row-wise with lists:
M = np.array([[-3, 5.7, 8],
              [2.1, 0, -8]])
print(f"{M}")
print(np.ndim(M)) # number of dimensions of the array
print(M.shape)    # shape of the array
print(M.size)     # number of elements in the array
```

```
[[-3.   5.7  8. ]
 [ 2.1  0.  -8. ]]
2
(2, 3)
6
```

Indexing und slicing of vectors and lists:

```
x = np.array([1, 2.1, -5])

# accessing vector (=1-dim-array) items works the same as with lists:
print(x[0])     # Python starts indexing with 0.
print(x[1:3])   # items from index 1 included until index 3 excluded
print(x[:2])    # items up to index 2 with index 2 excluded
print(x[1:])    # items starting from index 1 with index 1 included
print(x[-1])    # last item
```

```
print(x[-2])    # second last item
print(x[-2:])   # from the second last item to the end
print(x[-2:-1]) # should be clear now :-)
```

```
1.0
[ 2.1 -5. ]
[1.  2.1]
[ 2.1 -5. ]
-5.0
2.1
[ 2.1 -5. ]
[2.1]
```

Indexing and slicing of matrices:

```
M = np.array([[-3, 5.7, 8],
              [2.1, 0, -8]])
print(f"{M}")

# accessing matrix items works similar:
print("first row, second column:")
print(M[0, 1])
print("first row:")
print(M[0, :])
print("second column:")
print(M[:, 1])
print("second and third column:")
print(M[:, 1:3])
print("first two columns:")
print(M[:, :2])
```

```
[[-3.   5.7  8. ]
 [ 2.1  0.  -8. ]]
first row, second column:
5.7
first row:
[-3.   5.7  8. ]
second column:
[5.7 0. ]
second and third column:
[[ 5.7  8. ]
```

```
 [ 0.  -8. ]]
first two columns:
[[-3.   5.7]
 [ 2.1  0. ]]
```

Beachten Sie, dass beim Slicing das Rückgabe-Array die angepasste Dimension hat, die sich von der Dimension des Ausgangs-Arrays unterscheiden kann. Hier ein Beispiel, wie man das Slicing so anpasst, dass das Rückgabe-Array die gleiche Dimension wie das Ausgangs-Array hat:

```
M = np.array([[-3, 5.7, 8],
              [2.1, 0, -8]])
print(f"{M}")
print("first column as 1-dim array:")
print(M[:, 0])
print("first column as 2-dim array:")
print(M[:, [0]])
```

```
[[-3.   5.7  8. ]
 [ 2.1  0.  -8. ]]
first column as 1-dim array:
[-3.   2.1]
first column as 2-dim array:
[[-3. ]
 [ 2.1]]
```

Transponieren einer Matrix:

```
print(f"{M}")
print(f"{M.T}")
```

```
[[-3.   5.7  8. ]
 [ 2.1  0.  -8. ]]
[[-3.   2.1]
 [ 5.7  0. ]
 [ 8.  -8. ]]
```

Mutability

Listen und NumPy-Arrays sind veränderbar (engl. mutable), d. h. sie können nach ihrer Erzeugung verändert werden.

```
a = np.array([1, 2, 3])
print(f"{a = }")
b = a     # Caution: b references the same object as a
b[1] = 4  # Changing b changes a!
print(f"{a = }")
```

```
a = array([1, 2, 3])
a = array([1, 4, 3])
```

```
# If you do not want this, you have to make a copy:
a = np.array([1, 2, 3])
print(f"{a = }")
b = a.copy() # b is a copy of a
b[1] = 4     # changing b does not change a
print(f"{a = }")
```

```
a = array([1, 2, 3])
a = array([1, 2, 3])
```

Vektor- und Matrizenrechnung

```
v = np.array([-1, 5, 0])  # 1-dim array
w = np.array([ 2, 2, 1])  # 1-dim array

print("inner product:")
print(f"{v@w = } or {np.dot(v, w) = }")
print("cross product:")
print(f"{np.cross(v, w) = }")

print("Caution: These are element-wise operations:")
print(f"{v*w = }")
print(f"{v/w = }")
print(f"{v**2 = }")
```

```
inner product:
v@w = np.int64(8) or np.dot(v, w) = np.int64(8)
cross product:
np.cross(v, w) = array([  5,   1, -12])
Caution: These are element-wise operations:
v*w = array([-2, 10,  0])
v/w = array([-0.5,  2.5,  0. ])
v**2 = array([ 1, 25,  0])
```

Die Matrixmultiplikation wird wie das innere Produkt mit dem `@` Operator durchgeführt:

```
M = np.array([[-3, 5.7, 8],
              [2.1, 0, -8]])
N = np.array([[1, 2],
              [3, 4],
              [5, 6]])
print("M@N = ")
print(f"{M@N}")

print("M@v = ")
print(f"{M@v}")
```

```
M@N =
[[ 54.1  64.8]
 [-37.9 -43.8]]
M@v =
[31.5 -2.1]
```

Berechnung von Rängen, Determinanten, Eigenwerten, Eigenvektoren:

```
M = np.array([[-3, 2, 8],
              [ 2, 0, 1],
              [ 8, 1, 3]])
rank = np.linalg.matrix_rank(M)
print(f"{rank = }")
det = np.linalg.det(M)
print(f"{det = :.2f}")
eigenvalues, eigenvectors = np.linalg.eig(M)
print("1-dim array of eigenvalues:")
print(f"{eigenvalues}")
print("2-dim array of eigenvectors in columns:")
```

```
print(f"{eigenvectors}")

print("Some check by computing M*v_1 - lambda_1*v_1")
print("which should be a numerically zero vector:")
result = M@eigenvectors[:, 0] - eigenvalues[0]*eigenvectors[:, 0]
print(f"{result = }")
```

```
rank = np.int64(3)
det = 23.00
1-dim array of eigenvalues:
[ 8.97566072 -8.68045912 -0.2952016 ]
2-dim array of eigenvectors in columns:
[[ 0.56706349  0.82346126  0.01872296]
 [ 0.21494363 -0.12599762 -0.96846468]
 [ 0.7951341  -0.55320535  0.2484464 ]]
Some check by computing M*v_1 - lambda_1*v_1
which should be a numerically zero vector:
result = array([-1.77635684e-15, -1.11022302e-15, -8.88178420e-16])
```

Lineare Gleichungssysteme

Wir lösen das quadratische lineare Gleichungssystem $Ax = b$ und überprüfen zuerst, ob es eine eindeutige Lösung gibt:

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

(m, n) = A.shape
print(f"rank(A) = {np.linalg.matrix_rank(A)}")
if np.linalg.matrix_rank(A) == n:
    x = np.linalg.solve(A, b)  # Matrix A must be square and of full-rank!
    print(f"unique solution: {x = }")
else:
    print("no unique solution!")
```

```
rank(A) = 2
unique solution: x = array([-4. ,  4.5])
```

Kern (engl. null space, kernel) und Bild (engl. column space, range, image) einer Matrix:

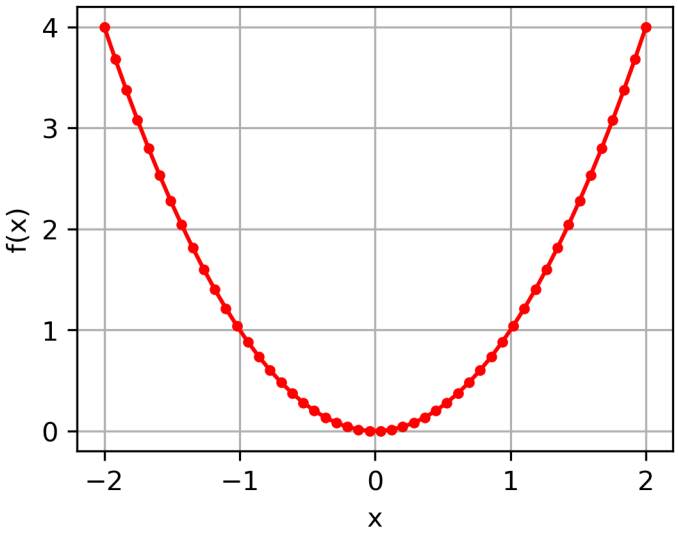
```
import scipy as sp

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

print("matrix A:")
print(f"{A}")
print(f"{np.linalg.matrix_rank(A) = }.")

print("orthonormal basis for the null space (kernel) of A:")
N = sp.linalg.null_space(A)
print(f"{N}")

print("orthonormal basis for the column space (range, image) of A:")
C = sp.linalg.orth(A)
print(f"{C}")

# the orthonormal bases are given as column vectors.
```

```
matrix A:
[[ 1  2 -1]
 [ 2  4  3]]
np.linalg.matrix_rank(A) = np.int64(2).
orthonormal basis for the null space (kernel) of A:
[[ 8.94427191e-01]
 [-4.47213595e-01]
 [-3.33066907e-16]]
orthonormal basis for the column space (range, image) of A:
[[-0.27000134 -0.96285995]
 [-0.96285995  0.27000134]]
```

Plot einer Funktion

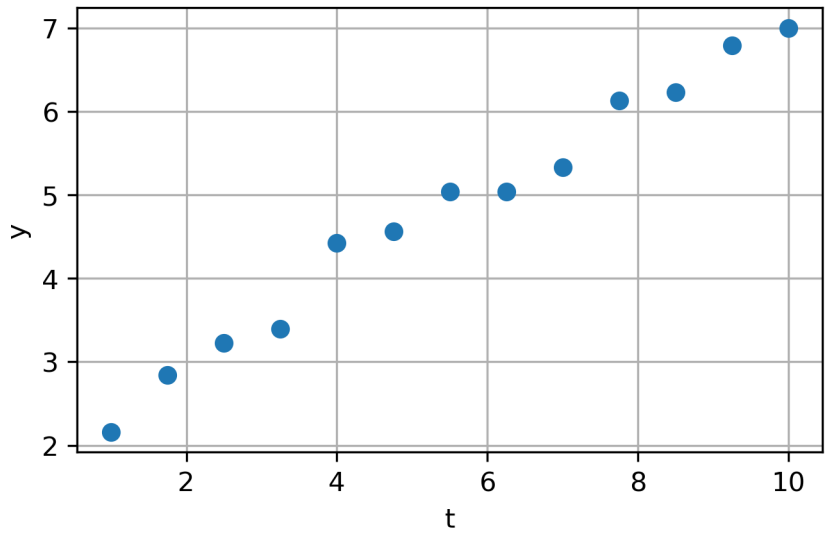
```
# define the function f:
def f(x):
    return x**2

# 50 grid points distributed between -2 and 2 with equal spacing:
x = np.linspace(-2, 2, num=50)
```

```
# plot the function at the grid points:
import matplotlib.pyplot as plt

plt.figure(figsize=(4, 3))
plt.plot(x, f(x), linestyle='-', color='red', marker='.')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.grid(True)
plt.show()
```



Regression

```
n = 13                                  # number of data points
t = np.linspace(1, 10, num = n)         # time points measurements
noise = 0.2*np.random.normal(size = n)  # noise in measurement values
y = 2 + 0.5*t + noise                   # measurement values: line + noise

plt.figure(figsize=(5, 3))
plt.plot(t, y, 'o')
plt.xlabel('t')
plt.ylabel('y')
plt.grid(True)
```




```
col_of_ones = np.ones(n)
A = np.stack((col_of_ones, t), axis = 1)
b = y.reshape(13, 1)

print(f"{A = }")
print(f"{b = }")
```

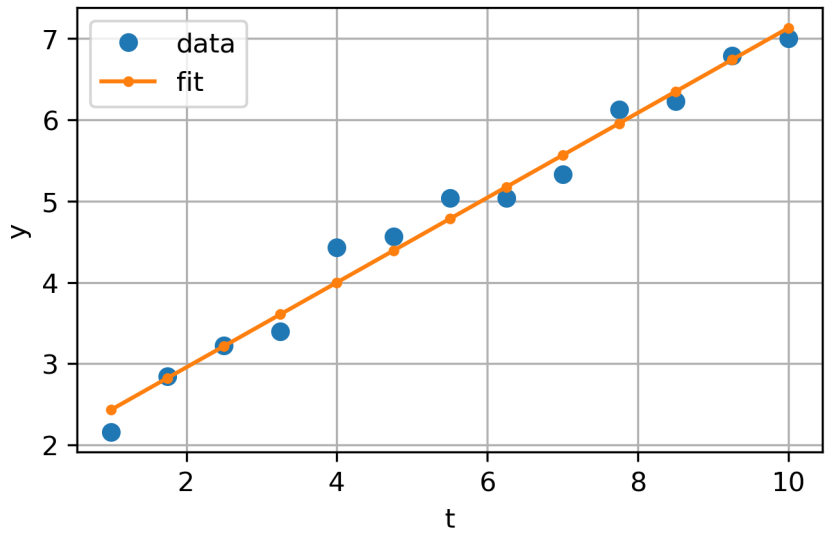
```
A = array([[ 1.  ,  1.  ],
       [ 1.  ,  1.75],
       [ 1.  ,  2.5 ],
       [ 1.  ,  3.25],
       [ 1.  ,  4.  ],
       [ 1.  ,  4.75],
       [ 1.  ,  5.5 ],
       [ 1.  ,  6.25],
       [ 1.  ,  7.  ],
       [ 1.  ,  7.75],
       [ 1.  ,  8.5 ],
       [ 1.  ,  9.25],
       [ 1.  , 10.  ]])
b = array([[2.16154654],
       [2.84443813],
       [3.22371433],
       [3.39677253],
       [4.42724793],
```

```
       [4.56281392],
       [5.0416529 ],
       [5.03728154],
       [5.33037706],
       [6.12762369],
       [6.22837227],
       [6.7897708 ],
       [6.9988643 ]])
```

```python
# via Formel:
x_hat_1 = np.linalg.inv(A.T @ A) @ A.T @ b
print(x_hat_1)

# via Normalgleichungen:
x_hat_2 = np.linalg.solve(A.T @ A, A.T @ b)
print(x_hat_2)

# via lstsq:
x_hat_3 = np.linalg.lstsq(A, b, rcond=None)[0]
print(x_hat_3)
```

```
[[1.91176814]
 [0.52192294]]
[[1.91176814]
 [0.52192294]]
[[1.91176814]
 [0.52192294]]
```

```python
# Fit A*x_hat:
y_hat = A@x_hat_1

plt.figure(figsize=(5, 3))
plt.plot(t, y, 'o', label='data')
plt.plot(t, y_hat, '.-', label='fit')
plt.xlabel('t')
plt.ylabel('y')
plt.legend(loc='best')
plt.grid(True)
```



For-Schleife

Zinseszinsrechnung:

```
B_0 = 1000.0  # initial balance in EUR
q = 1.05      # interest rate factor
n = 5         # number of years

# range: start 1 is included, stop n + 1 is excluded!
for i in np.arange(1, n + 1, 1):
    B_i = B_0*q**i
    print(f"Balance after {i} years = {B_i:.2f} EUR.")
```

```
Balance after 1 years = 1050.00 EUR.
Balance after 2 years = 1102.50 EUR.
Balance after 3 years = 1157.63 EUR.
Balance after 4 years = 1215.51 EUR.
Balance after 5 years = 1276.28 EUR.
```

While-Schleife

Wir berechnen die Fakultät einer natürlichen Zahl $n > 0$:

```python
n = 10
f = 1  # initial value for factorial n!
k = n  # initial decreasing number k
while k > 0:
    print(f"{k = }")
    f = f*k
    k = k - 1

print(f"{n}! = {f}")
```

```
k = 10
k = 9
k = 8
k = 7
k = 6
k = 5
k = 4
k = 3
k = 2
k = 1
10! = 3628800
```

If-Elif-Else Abfrage

Wir bestimmen das Vorzeichen von ein paar Zahlen:

```python
numbers = np.arange(-3, 4, 1)
for n in numbers:
    if n > 0:
        print(f"{n} hat das Vorzeichen +.")
    elif n == 0:
        print(f"{n} hat kein Vorzeichen.")
    else:
        print(f"{n} hat das Vorzeichen -.")
```

```
-3 hat das Vorzeichen -.
-2 hat das Vorzeichen -.
-1 hat das Vorzeichen -.
0 hat kein Vorzeichen.
1 hat das Vorzeichen +.
```

```
2 hat das Vorzeichen +.
3 hat das Vorzeichen +.
```

Beachte: Eine If-Elif-Else Abfrage muss nicht zwingend `elif` oder `else` Teile haben.