



Statistik und Wahrscheinlichkeitsrechnung

Kapitel 00: Einführung in Python & Numpy

Prof. Dr. Adrian Ulges

B.Sc. *Informatik*
Fachbereich DCSM
Hochschule RheinMain



Statistik und Wahrscheinlichkeitsrechnung

Kapitel 00: Einführung in Python & Numpy

Prof. Dr. Adrian Ulges

B.Sc. *Informatik*
Fachbereich DCSM
Hochschule RheinMain

Programmieren in Python

Was / Wann ?

- ▶ Python-Crashkurs in erster Sprechstunde und ersten Übungen, dann ca. alle **zwei Wochen** eine Aufgabe.
- ▶ Klausurrelevant.

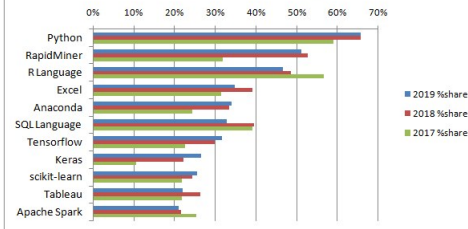
Warum?

- ▶ Praktische Datenanalyse ist sehr gefragt.
- ▶ Python ist mit Abstand die **verbreitetste Lösung**.

Was ist Python?

- ▶ objektorientierte Skriptsprache mit funktionalen Elementen
- ▶ einfach erlernbar, mächtige Datenstrukturen + Bibliotheken
- ▶ sehr weit verbreitet – die meist genutzte Skriptsprache (*Rang 5 4 3 2 1 in den TIOBE Top Ten¹*).

Top Analytics, Data Science, Machine Learning Software 2017-2019, KDnuggets Poll



¹<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

Data Science mit Python

Python bietet ein komplettes **Ökosystem** für Datenanalyse:

Numpy

- ▶ Numerisches Rechnen mit Vektoren/Matrizen/Tensoren
- ▶ Operationen auf Arrays (schleifen-frei)
- ▶ Slicing, Filtern, Aggregieren, Kennzahlen

Pandas

- ▶ Datenanalyse (“data wrangling”)
- ▶ \approx Excel

Matplotlib

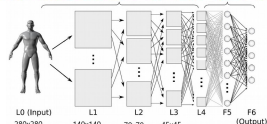
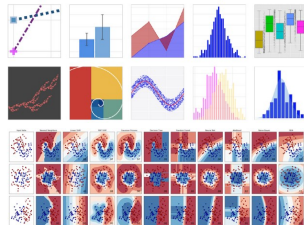
- ▶ Datenvisualisierung

Sklearn, Tensorflow

- ▶ Maschinelles Lernen, Deep Learning



	Rank		Title	Worldwide gross	Year
0	1		Avatar	\$2,787,965,067	2009
1	2		Titanic	\$2,186,772,302	1997
2	3		Star Wars: The Force Awakens	\$2,069,223,624	2015
3	4		Jurassic World	\$1,671,713,208	2015
4	5		The Avengers	\$1,518,812,568	2012
5	6		Furious 7	\$1,516,045,911	2015
6	7		Avengers: Age of Ultron	\$1,405,403,694	2015
7	8		Harry Potter and the Deathly Hallows – Part 2	\$1,341,511,219	2011
8	9		Frozen	\$1,297,000,000	2013





Python gibt es in Version 2 (veraltet) und 3 (derzeit Standard).

Python kann man in der **Shell**, in **Programmen** ("Skripts") oder **Notebooks** benutzen.

```
ulges@ulges-nb:~$ python3.7
Python 3.7.0 (default, Sep 10
>>> s = 'Hallo Welt!'
>>> print(s)
Hallo Welt!
>>> █
```

Python Basics

Dieses Notebook bietet eine Reihe von Übungen zur Einführung in die Programmiersprache Python. Bearbeiten Sie zuerst:

- Das Notebook 02_python101 für einen einfachen Syntax-Beispiel
- Das Python-Naming-Tutorial der University of Stanford

1. Methoden und Strings

Schreiben Sie eine Methode `len(s)`, die - gegeben einen String `s` - ein Tupel `(subj, count)` zurückgibt. Hierbei soll `s` die Länge des Strings sein, `count` die Anzahl der Vorkommen, und `count` die letzten vier Buchstaben. Berechnen Sie sich hier und im Folgenden um eine möglichst kompakte Lösung.

Hinweis: Python bietet eine eigene `sum()`-Funktion, z.B. `sum([1,4,5]) = 10`.

```
def zeros(s):
    return None # TODO
```

`zeros("1002x789402x12")` # ruft die Funktion auf.



- ▶ Python ist **dynamisch getypt**. Semikolons sind nicht nötig.

```
1 # ein Kommentar
2 x = 3.14159
3 x = not( ( True and False ) or 5<10) --> True
4 x = 2*100 + 3 + True --> 204
```

- ▶ If-Statements: Keine Klammern, aber **Einrückung** (auch bei Schleifen etc.)!

```
1 if x < 2:
2     print('Yeah', False)
3 else:
4     print('Zahl ist: %d' %x)
```



- Funktionen definieren mit def. **Default-Parameterwerte** sind möglich.

```
1     def foo(a, b=5):  
2         return a*b  
3  
4     print( foo(4) )           --> 20  
5     print( foo(4, b=7) )     --> 28
```

- Tupel und Listen sind fertig eingebaut.

```
1     mylist = ['el', 11, 'eggos'] # Liste (mutable)  
2     mylist = ('el', 11, 'eggos') # Tuple (immutable)
```

- Zugriff mit Array-Syntax, viele nette Tricks

```
1     mylist[1]                --> 11  
2     mylist[2]==mylist[-1]    --> True # -1 = letztes Element  
3     name,age,food = mylist    # 'assignment unpacking'
```

► Schleifen: iterator-mäßig

```
1 kids = ['eleven', 'will', 'justin']
2 for kid in kids:
3     print(kid)
4
5 > eleven
6 > will
7 > justin
```

► Zählschleifen mit range()

```
1 for x in range(10,14):
2     print(x)
3
4 > 10
5 > 11
6 > 12
7 > 13
```

- Mehrere Listen **simultan** durchlaufen mit `zip()`

```
1 kids = ['eleven', 'will', 'justin']
2 ages = [11, 12, 13]
3 foods = ['eggos', 'pizza', 'noodles']
4
5 for (kid,age,food) in zip(kids,ages,foods):
6     print(kid, '(', age, ') likes', food)
7
8 > eleven ( 11 ) likes eggos
9 > will ( 12 ) likes pizza
10 > justin ( 13 ) likes noodles
```

- **List Comprehension**: Listen in einer Zeile verarbeiten!

```
1 squares = [x**2 for x in ages]
2
3 > 121
4 > 144
5 > 169
```



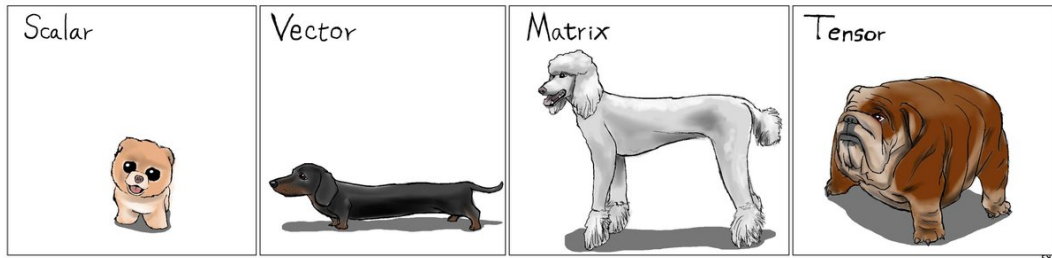

- ▶ List Comprehension (cont'd)

```
1 kids = ['eleven', 'will', 'justin']
2 ages = [11, 12, 13]
3
4 [kid for (kid,age) in zip(kids,ages) if age<13]
5
6 > eleven
7 > will
```

Es geht noch viel mehr...

- ▶ Dictionaries
- ▶ Klassen
- ▶ Module (importierbar)
- ▶ ...

Numpy = Einfaches Programmieren mit Arrays



Numpy-Arrays können darstellen:

- ▶ Skalare (*shape* (1))
- ▶ Vektoren (*shape* (n))
- ▶ Matrizen (*shape* (n,m))
- ▶ generelle Tensoren (*shape* (n,m,p,r,q,...))

Numpy = Einfaches Programmieren mit Arrays



► Rechenoperationen auf Arrays

```
1 c = a + b
```

$$\begin{array}{r} \text{a} \quad \begin{bmatrix} 1 & 4 & -3 & 5 & 4 & 6 \end{bmatrix} \\ + \text{b} \quad \begin{bmatrix} 0 & 2 & 3 & 1 & 3 & 2 \end{bmatrix} \\ \hline \text{c} \quad \begin{bmatrix} 1 & 6 & 0 & 6 & 7 & 8 \end{bmatrix} \end{array}$$

► Lineare Algebra

```
1 x = np.linalg.solve(A,b)
```

$$\begin{array}{ccc} \begin{bmatrix} 1 & 4 & 3 \\ 5 & 4 & 6 \\ 0 & 2 & 1 \end{bmatrix} & * & \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix} \\ \text{A} & & \text{x} \quad \text{b} \end{array}$$

► Deskriptive Statistik

```
1 meds = np.median(A, axis=1)
```

$$\begin{array}{ccc} \begin{bmatrix} 1 & 1 & 3 & 4 & 8 \\ 4 & 4 & 4 & 4 & 5 \\ 0 & 2 & 1 & 5 & 2 \end{bmatrix} & \rightarrow & \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix} \\ \text{A} & & \text{meds} \end{array}$$

► Filtern / Slicing

```
1 A_top = A[:3,:]
```

$$\begin{array}{ccc} \begin{bmatrix} 1 & 1 & 3 \\ 4 & 4 & 4 \\ 0 & 2 & 1 \\ 5 & 2 & 8 \\ 3 & 1 & 0 \end{bmatrix} & \rightarrow & \begin{bmatrix} 1 & 1 & 3 \\ 4 & 4 & 4 \\ 0 & 2 & 1 \end{bmatrix} \\ & & \text{A_top} \end{array}$$

Numpy: Arrays erstellen



	km	year	ps	damaged	price
car1	105.089	2008	75	0	4.320
car2	25.047	2016	95	1	6.740

- **Importiere** Numpy zur Verwendung.

```
1 import numpy as np
```

- Erstelle ein Arrays aus einer Liste von Zahlen.

Das Array hat eine **Form** und einen **Zahlentyp**:

```
1 car1 = np.array([105089,2008,75,0,4320])
2 print(car1.shape)           -> (5,)
3 print(car1.dtype)           -> 'int64'
```

- Erstelle eine **2×5-Matrix** aus zwei Listen:

```
1 cars = np.array([[105089, 2008, 75, 0, 4320],
2                  [ 25047, 2016, 95, 1, 6740]])
3 print(cars.shape)           -> (2,5)
```

Numpy: Arrays erstellen



- Es gibt verschiedene weitere Konstruktoren für Arrays:

```
1 np.zeros(shape=(2,4))      -> [[0,0,0,0],  
2                             [0,0,0,0]]
```

Numpy: Rechnen mit Arrays



	km	year	ps	damaged	price
car1	105.089	2008	75	0	4.320
car2	25.047	2016	95	1	6.740

Numpy unterstützt **elementweise Operationen**:

- ▶ “Addiere” beide Autos

```
1 x = car1 + car2      -> [130136,4025,170,1,11060]
```

- ▶ Verdopple alle Einträge

```
1 x = 2 * car1         -> [210178,4016,150,0,8640]
```

- ▶ Bilde das “Durchschnittsauto” aus beiden

```
1 x = (car1 + car2)/2  -> [65068,2012,85,0.5,5530]
```

- ▶ Elementweises Quadrieren / Logarithmieren / ...

```
1 x = car1**2
2 x = np.log(car1)
```

	km	year	ps	damaged	price
car1	105.089	2008	75	0	4.320
car2	25.047	2016	95	1	6.740

- Wir greifen mit der bekannten Array-Syntax auf Werte zu:

```
1 km = car1[0]      -> 105089
2 year = car1[1]    -> 2008
3
```

- Negative Indizes zählen rückwärts ...

```
1 price = car1[-1]   -> 4320
2 dam    = car1[-2]   -> 0
```

- Wir können **Subarrays** "ausschneiden": `array[von:bis]`

```
1 slice = car1[1:4]   # Werte 1-3: -> [2008,75,0]
2 slice = car1[0:-1]  # alle ausser letztem Wert.
```



	km	year	ps	damaged	price
car1	105.089	2008	75	0	4.320
car2	25.047	2016	95	1	6.740

- Lässt man die Grenzen von und bis weg, gelten die **Default-Werte** von=0 und bis=n

```
1 print(car1[:3])      # = car1[0:3]
2 print(car1[1:])      # = car1[1:5]
3 print(car1[:-1])     # = car1[0:4]
4 print(car1[:])       # = car1 komplett
```

- Mit der Syntax **array[von:bis:schritt]** lässt sich eine **Schrittweite** angeben.

```
1 car1[1:5:2]          # jeder 2.Wert von 1-4 -> [2008,0]
2 car1[::-1]           # alle Werte, falschherum
3                      # -> [4320,0,75,2008,105089]
```


Slicing: Mehrdimensionale Arrays



- Slicing lässt sich auch auf mehrere Dimensionen anwenden.
- Syntax(2D): `arr[zeile_von:zeile_bis , spalte_von:spalte_bis]`

km	year	ps	damaged	price
105.089	2008	75	0	4.320
25.047	2016	95	1	6.740
54.089	2015	275	0	23.320
81.086	2012	80	1	3.890
19.023	2018	90	0	9.870
250.028	2004	120	0	1.570

`cars[2,3]`

km	year	ps	damaged	price
105.089	2008	75	0	4.320
25.047	2016	95	1	6.740
54.089	2015	275	0	23.320
81.086	2012	80	1	3.890
19.023	2018	90	0	9.870
250.028	2004	120	0	1.570

`cars[-1,-1]`

km	year	ps	damaged	price
105.089	2008	75	0	4.320
25.047	2016	95	1	6.740
54.089	2015	275	0	23.320
81.086	2012	80	1	3.890
19.023	2018	90	0	9.870
250.028	2004	120	0	1.570

`cars[1:3,0:2]`

km	year	ps	damaged	price
105.089	2008	75	0	4.320
25.047	2016	95	1	6.740
54.089	2015	275	0	23.320
81.086	2012	80	1	3.890
19.023	2018	90	0	9.870
250.028	2004	120	0	1.570

`cars[1,:]`

km	year	ps	damaged	price
105.089	2008	75	0	4.320
25.047	2016	95	1	6.740
54.089	2015	275	0	23.320
81.086	2012	80	1	3.890
19.023	2018	90	0	9.870
250.028	2004	120	0	1.570

`cars[:, -1]`

km	year	ps	damaged	price
105.089	2008	75	0	4.320
25.047	2016	95	1	6.740
54.089	2015	275	0	23.320
81.086	2012	80	1	3.890
19.023	2018	90	0	9.870
250.028	2004	120	0	1.570

`cars[2]`

Slicing mit Bedingungen



	km	year	ps	damaged	price
car1	105.089	2008	75	0	4.320
car2	25.047	2016	95	1	6.740


- **Boolesche Bedingungen** auf Arrays ergeben Boolesche Arrays

```
1 car1>1000 -> [True,True,False,False,True]
```

- Mit diesen booleschen Arrays können wir wieder **slicen**!

```
1 car1[car1>1000] = 999 # clippt hohe Werte zu 999
```

	km	year	ps	damaged	price
car1	105.089	2008	75	0	4.320



car1 [neu]	999	999	95	1	999
---------------	-----	-----	----	---	-----

Slicing mit Bedingungen



Slicing mit Bedingungen funktioniert auch mehrdimensional!

Beispiel: Clippe hohe PS-Werte

```
1 cars[cars[:,2]>100,2] = 99
```

km	year	ps	damaged	price
105.089	2008	75	0	4.320
25.047	2016	95	1	6.740
54.089	2015	275	0	23.320
81.086	2012	80	1	3.890
19.023	2018	90	0	9.870
250.028	2004	120	0	1.570



km	year	ps	damaged	price
105.089	2008	75	0	4.320
25.047	2016	95	1	6.740
54.089	2015	99	0	23.320
81.086	2012	80	1	3.890
19.023	2018	90	0	9.870
250.028	2004	99	0	1.570

Kennzahlen



Wir können mit numpy auch **statistische Kennzahlen** berechnen:

- **Mittelwert und Standardabweichung** über alle Werte

```
1 np.mean(cars), np.std(cars)    -> 19896.1, 49301.5
```

- **Median, spaltenweise**

```
1 np.median(cars, axis=0)
```

km	year	ps	damaged	price
105.089	2008	75	0	4.320
25.047	2016	95	1	6.740
54.089	2015	275	0	23.320
81.086	2012	80	1	3.890
19.023	2018	90	0	9.870
250.028	2004	120	0	1.570



67.587,5	2013,5	92,5	0	5.530
----------	--------	------	---	-------

Kennzahlen: Korrelationen



Wir können die Korrelationen zwischen Merkmalen berechnen:

```
1 np.corrcoef(cars.T)
```

(Anmerkung: `corrcoef()` verwendet die transponierte Datenmatrix (deshalb `'.T'`))

km	year	ps	damaged	price
105.089	2008	75	0	4.320
25.047	2016	95	1	6.740
54.089	2015	275	0	23.320
81.086	2012	80	1	3.890
19.023	2018	90	0	9.870
250.028	2004	120	0	1.570



	km	year	ps	damaged	price
km	1	-0.94	-0.07	-0.32	-0.49
year	-0.94	1	0.18	0.27	0.56
ps	-0.07	0.18	1	-0.36	0.89
damaged	-0.32	0.27	-0.36	1	-0.29
price	-0.49	0.56	0.89	0.29	1

