Was ist NumPy?

- Abkürzung für `Numerical Python`
- Bildet die Basis für viele Python-Projekte (Vor allem zum Erzeugen von Arrays)
- Verwaltet multi-dimensionale Arrays und stellt Funktionen dafür bereit
- Kann mit großen Datenmengen umgehen
- Sehr schnell, da wesentliche Teile davon in `C` ausprogrammiert sind
- Geht von den Berechnungsmöglichkeiten her relativ weit (werden wir nicht ausschöpfen)
- Basis-Datenstruktur ist ein `ndarray` (n-dimensionales Array) - oder auch NumPy Array

```
1 # Hat sich so eingebürgert dass als np importiert wird
2 import numpy as np
3 a = np.array([1,2,3]) # So wird ein 1-dimensionales Array angelegt
4 print(a) # Ausgabe des Arrays
5 print(a.shape) # Wie viele Werte auf den einzelnen Dimensionen
6 print(len(a.shape)) # Wie viele Dimensionen
7 print(a.dtype) # Um welchen Datentyp handelt es sich?
```

- Ausführlicheres Tutorial: `https://www.tutorialspoint.com/numpy/index.htm`

Hilfreiche Funktionen für die Erzeugung von (mehrdimensionalen) Arrays

Eindimensionale Arrays:

```
a1 = np.arange(10) # [0 1 2 3 4 5 6 7 8 9]
a11 = np.arange(10, 20, 0.5) # (start, stop, step)
# 5 Werte zwischen 1 und 2 mit gleichem Abstand:  [1. 1.25 1.5 1.75 2.]
a2 = np.linspace(1, 2, 5)
a3 = np.ones(10) # lauter 1en
a4 = np.zeros(10) # lauter 0en
a5 = np.random.rand(10) # Zufallszahlen zwischen [0,1[]
# 5 Ganzahlen zufällig gezogen zwischen [10 und 20[]
a6 = np.random.randint(10,20,5) # Gleichverteilt (alle Zahlen haben die gleiche
   Chance, gezogen zu werden)
```

Mehrdimensionale Arrays: Hier werden einfach weitere Dimensionen angegeben

```
a5 = np.random.rand(3,2,4)
```

Erzeugt...

```
[[[0.9502729  0.47978706 0.80420501 0.69607102]
[0.12001849 0.07467159 0.74332051 0.90530865]]

[[0.7824477  0.29988021 0.39840717 0.02550705]
[0.10465624 0.37901605 0.01341696 0.2704336 ]]

[[0.69887726 0.79655702 0.77091248 0.6599289 ]
[0.97888692 0.11476978 0.86355308 0.44119605]]]
```

Umgang mit Datentypen

Der Datentyp kann bei der Erzeugung gesetzt werden:

```
d1 = np.array([1,2,3], dtype=complex) # [1.+0.j 2.+0.j 3.+0.j]
```

```
d1 = np.array([1,2,3], dtype=str) # ['1' '2' '3']
```

Auch bei bereits erzeugten Arrays kann der Typ geändert werden:

```
d2 = d2.astype(int) #[1 2 3]
# copy=false : in das gleiche Array
# casting='safe' : Es gibt einen Fehler wenn casting nicht funktioniert
ad1 = ad1.astype(str, copy=False, casting='safe')
# die nicht built-in Datentypen:
db = np.array(['1995-10-28 23:55', '2020-01-18 23:01'])
db = db.astype('M') # Datetime
```

Wichtigste verfügbare Datentypen:

- i - integer
- b - boolean
- u - unsigned integer
- f - float
- c - complex float
- m - timedelta
- M - datetime
- O - object
- S - string

Weitere Hilfreiche Funktionen

Ändern der Dimension:

```
1 dr1 = np.zeros(10)
2 dr1 = dr1.reshape(2,5)
3 print(dr1)
```

Erzeugt...

```
[[0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0.]]
```

Einfache Rechenoperationen: Operatoren können auf ganze Arrays angewandt werden

```
1  r1 = np.arange(10) * 10
```

Erzeugt...

```
[0 10 20 30 40 50 60 70 80 90]
```

Zugriff auf einzelne Elemente: So wie in Python gewohnt

```
1 a1 = np.arange(10) #[0 1 2 3 4 5 6 7 8 9]
2 print(a1[1]) # 1
3 print(a1[1:4]) # [1 2 3]
4 print(a1[-1]) #9 Beginnt von hinten
5 print(a1[6:]) # [6 7 8 9]
```

Mehrdimensionaler Zugriff auf einzelne Elemente

Ändern der Dimension:

```
1  am = np.array([[1,2,3],[4,5,6],[7,8,9]])
2  print("Gesamte Matrix")
3  print(am)
4  print("Spalte 2")
5  print(am[:,1]) #Spalte 2
6  print("Zeile 2")
7  print(am[1,:]) #Zeile 2
8
9  print("Einzelne Werte")
10 print(am[[1,2],[0,1]]) #Positionen (1,0), (2,1)), also 4 und 8
```

Erzeugt...

```
Gesamte Matrix
[[1 2 3]
[4 5 6]
[7 8 9]]
Spalte 2
[2 5 8]
Zeile 2
[4 5 6]
Einzelne Werte
[4 8]
```

Abfragen von Array-Werten

Mittels Bedingungen: Diese Variante geht mit "normalen" Python Arrays nicht!

```
1          ab = np.arange(10) #[0 1 2 3 4 5 6 7 8 9]
2          print("ab < 5:")
3          print(ab[ab < 5])
4          print("~ab < 5:") #~ist 'not'
5          print(ab[~(ab < 5)])
```

Erzeugt...

```
ab < 5:
[0 1 2 3 4]
~ab < 5:
[5 6 7 8 9]
```

Fehlende Werte: Die Konstante `np.nan`

```
1          a = np.array([np.nan, 1,2,np.nan,3,4,5])
2          print a[~np.isnan(af)] # Filtert alle nicht fehlenden Werte
```

Komplexere Operationen auf Array-Werte

```
1 am = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

Wenn man nicht die Werte, sondern die Positionen haben möchte (`where`):

```
1 i = np.where(am > 5)
```

Ergibt...

```
(array([1, 2, 2, 2]), array([2, 0, 1, 2]))
```

Das kann wiederum als Index verwendet werden:

```
1 print(am[i]) # [6 7 8 9]
```

Sogar eine Zuweisung an alle diese Positionen ist dann möglich:

```
1 am[i] = 20
```

Ergibt:

```
[[ 1  2  3]
[ 4  5 20]
[20 20 20]]
```

Manipulation von Array-Werten mittels Funktionen

Diese Funktionen werden mit np (dem numpy import alias), nicht dem Objekt, aufgerufen.

Daten für die Beispiele unten:

```
1 am = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
```

Hier die wichtigsten Funktionen:

```
1 # Zeilenweise angehängt. axis=1 Spaltenweise angehängt
2 am = np.append(am,[[13,14,15,16]], axis=0)
3 # Ähnlich dazu insert, hier muss der Index wo eingefügt werden soll
    angegeben werden
```

```
1 #Zerlege die Matrix Spaltenweise in sub-Arrays der Größe 2
2 af2 = np.split(af,2 , 1)
```

```
1 #Lösche die 3. Zeile in af. Die neue Matrix wird zurückgegeben
2 af3 = np.delete(af, 2,0)
```

```
1 #Schmeisse die doppelten hinaus und gib Ergebnis als 1-dim Array zurück
2 af4 = np.unique(af)
```

```
1 #Mache aus einem mehrdimensionalen Array ein Eindimensionales
2 af5 = np.ravel(af4)
```

Weitere Funktionen und Details unter:
https://www.tutorialspoint.com/numpy/numpy_array_manipulation.htm

Kennwerte von Daten

```
1        a1 = np.random.normal(170,10, 1000) #Wird für die Beispiele
            verwendet
```

Lagemaße: Geben das Zentrum von Verteilungen an und dienen der Zusammenfassung von Daten

- `Mittelwert`: Summe / Anzahl
  `a1.mean()`
- `Median`: Wenn man die Werte sortiert ist es der Wert in der Mitte
  `np.median(a1)`
- `10% - Quantil`: links von diesem Wert sind die unteren 10 %
  `np.quantile(a1,0.1)`
- `unteres Quartil`: links von diesem Wert ist das untere Viertel
  `np.quantile(a1,0.25)`
- `Modus`: Der Wert der am öftesten Vorkommt

```
1                  from scipy import stats #dazu braucht man eine
                      weitere Bibliothek
2                  a2 = np.random.randint(10,20,500)
3                  m = stats.mode(a2)
4                  print("Modus", m.mode[0], m.count[0])
```

Streuungsmaße (Dispersionsmaße)

- `Standardabweichung`: Wie stark streuen die einzelnen Werte um den Mittelwert
  `np.std(a1)` 1x Standardabweichung: 68% der Werte 2x sind 95% 3x 99,7%, also z.B. mittleren 95%: `np.mean(a1) - np.std(a1)*2, np.mean(a1) + np.std(a1)*2`
- Spannweite (Range): max - min
  `a1.max() - a1.min()`

Umgang mit fehlenden Werten

- Wenn einzelne Zellen den Wert nan beinhalten, dann wird aus den Berechnungen auch `nan`.
- Möchte man, dass diese Werte ignoriert werden, dann gibt es spezielle Funktionen die mit `nan` beginnen, z.B.: `np.nanmean()`

```
1 print(np.nanmean(col1))
```

- Weitere Methoden, die mit `nan` umgehen können:
  - `np.nanmedian()`
  - `np.nanstd()`
  - `np.nanquantile()`
  - `np.nansum()`
- Es ist allerdings empfohlen, diese Methoden nicht unüberlegt zu verwenden und immer die `nan`'s zu ignorieren, denn oft ist es nicht beabsichtigt dass diese enthalten sind.
- Der Befehl, der für jeden Wert checkt es ein fehlender Wert ist: `np.isnan(col1)`
- So checkt man, ob irgendein fehlender Wert drin ist: `np.isnan(col1).any()`
- Eine Strategie könnte sein, die fehlenden Werte durch einen default-Wert zu ersetzen (das kann z.B. der Mittelwert sein, das kommt auf die Daten an).

```
1 col1[np.isnan(col1)] = -10
```

- Das ersetzen / behandeln der fehlenden Werte (bzw. die Auswahl der Strategie) ist keine triviale Angelegenheit, da gibt es teilweise Diskussionen in der Wissenschaft.

Daten aus Datei in NumPy importieren

- Das ist prinzipiell möglich, sollte aber eigentlich über Pandas gemacht werden. Mehr dazu später.
- Die Funktion dazu ist `genfromtxt`
- Es werden jede Menge Parameter angeboten, die bezgl. Festlegen der Datentypen nicht so funktionieren wie erwartet, deshalb folgende Vorgehensweise:
  1. Importieren des kompletten Datensatzes, mit den automatisch erkannten Datentypen
  2. Konvertieren in die gewünschten Datentypen nach dem Import (umständlich, macht aber weniger Kopfweh)
  3. Jede Spalte für sich aufbereiten und für die Analyse verwenden

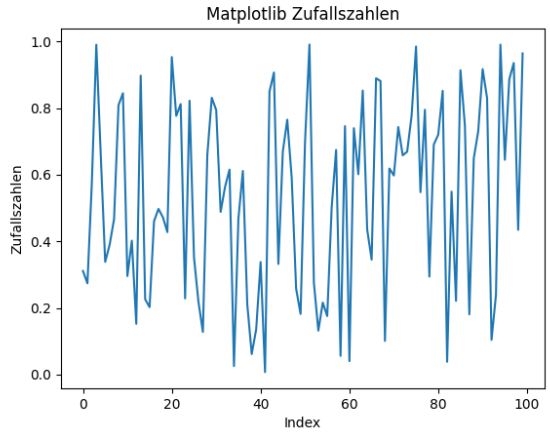
```
1 # Lade den Datensatz mit einem bestimmten Trennzeichen zwischen den
    Spalten, die erste Spalte wird nicht verwendet, da sie
    Spaltenüberschriften beinhaltet
2 d = np.genfromtxt('dataset.csv', delimiter=",", skip_header=1)
3 col1 = d[:,0] # Holen der ersten Spalte. Das ist dann ein ndarray
4 col1 = col1.astype('int') # Übertragen der Spalte in einen bestimmten
    Datentyp
```

Matplotlib - Allgemeines

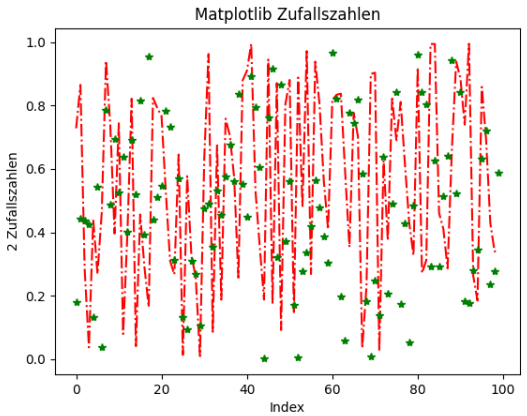
- Ist eine Bibliothek füt die Datenvisualisierung in Python
- Ist eher für die einfache Erstellung von Grafiken gedacht...
- `Plotly` besprechen wir später: kann schönere Grafiken, ist aber komplexer
- installation: `pip install matplotlib`
- import: `from matplotlib import pyplot as plt`
- Ausführliche Doku: `https://matplotlib.org/`

Grundaufbau:

```
1  x = np.arange(100)
2  y = np.random.rand(100)
3  plt.title("Matplotlib Zufallszahlen")
4  plt.xlabel("Index")
5  plt.ylabel("Zufallszahlen")
6  plt.plot(x,y)
7  # Speichern der Datei im aktuellen Arbeitsverzeichnis
8  plt.savefig("m1.png")
9  # Es geht ein viewer auf und zeigt den Plot. Nicht vor savefig
       aufrufen!
10 plt.show()
```



Matplotlib - mehrere Plots



Es besteht die Möglichkeit, mehrere Plots in eine Grafik zu geben:

```
y2 = np.random.rand(100)
plt.title("Matplotlib Zufallszahlen")
plt.xlabel("Index")
plt.ylabel("2 Zufallszahlen")
plt.plot(x,y, "r-.") # durchgägngige Linie in rot
plt.plot(x,y2,"g*") # Sterne in grün
plt.savefig("m2.png")
plt.show()
```

Kürzel für Farben:

- `'b'` Blue
- `'g'` Green
- `'r'` Red
- `'c'` Cyan
- `'m'` Magenta
- `'y'` Yellow
- `'k'` Black
- `'w'` White

Kürzel für Punkte bzw. Linien:

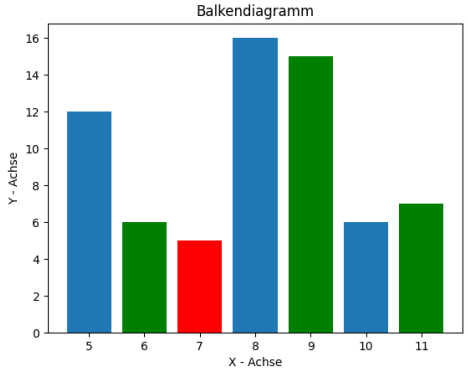
- `'-'` Durchgängige Linie
- `'--'` Strichlierte Linie
- `'-.'` Punt-Strich-Linie
- `':'` Punktierte Linie
- `'o'` Kreis (als Punkt)
- `'*'` Stern (als Punkt)
- `'+'` Plus-Zeichen (als Punkt)
- `'d'` bzw. `'D'` Kleiner bzw. großer Diamant (als Punkt)

Weitere Möglichkeiten unter
`https://www.tutorialspoint.com/numpy/numpy_matplotlib.htm`

Matplotlib - Arten von Plots

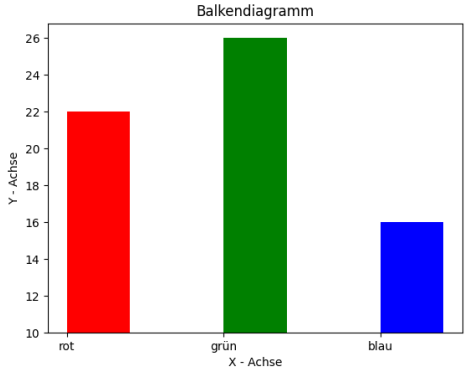
Das sind die wichtigsten zusätzlich angebotenen Plot-Varianten:
Balkendiagramm:

```
x = [5,8,10] # Position der Balken
y = [12,16,6] # Höhe der Balken

# Werden für die grünen Balken verwendet
x2 = [6,9,11]
y2 = [6,15,7]
plt.bar(x, y, align = 'center')
plt.bar(x2, y2, color = 'g', align = 'center')
plt.bar(7, 5, color = 'r', align = 'center') # Geht auch ohne Array
plt.title('Balkendiagramm')
plt.xlabel('X - Achse')
plt.ylabel('Y - Achse')
plt.show()
```



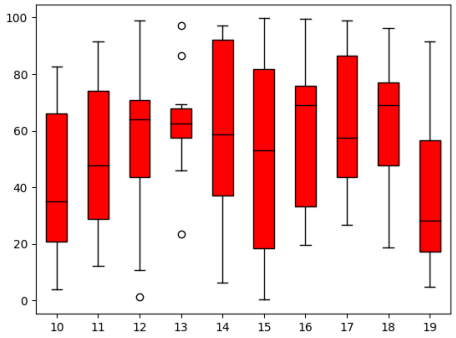
Weitere Parameter:

```
x = [1,2,3] # Position der Balken
y = [12,16,6] # Höhe der Balken

plt.bar(x,y, align = 'edge', width=0.4, bottom=10, color=['r','g','b'],
   tick_label=['rot','grün','blau'])
```



Matplotlib - Arten von Plots

Boxplot:




```
d = np.random.rand(100) * 100
d = d.reshape(10,10)

plt.boxplot(d, vert=True, # Horizontal oder Vertikal
        notch=False,  # Einbiegung beim Median
        patch_artist=True,  # Mit Farbe angefüllt
        labels=np.arange(10,20), # Beschriftung X-Achse
        boxprops=dict(facecolor='r', color='black'),  # Füllfarbe und Rahmenfarbe
        medianprops=dict(color='black')) # Farbe des Medianstrichs

plt.show()
```

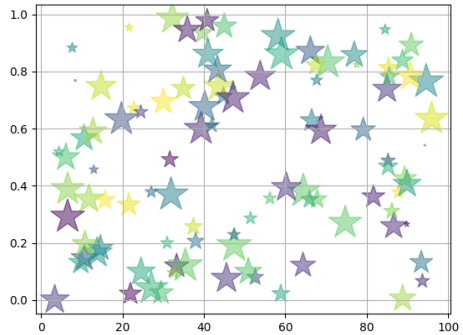
Erklärungen zu den Boxplots:

- Zur Darstellung von Verteilungen von Zahlen
- Mittlere Strich: Median
- Rote Box: Die mittleren 50%
- Die beiden Striche an den Enden: Bereich +- 2* Standardabweichung
- Punkte ausserhalb: Sogenannte Ausreisser

Matplotlib - Arten von Plots

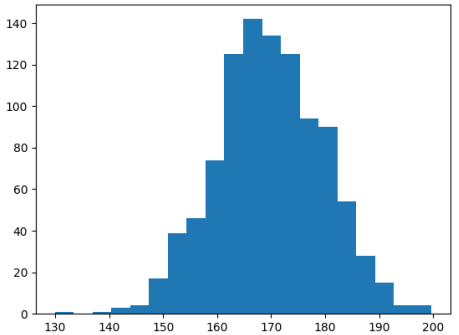
Scatterplot:

```
x = np.random.rand(100) * 100
y = np.random.rand(100)
# Diese Werte werden, wenn gewünscht, als Farben abgebildet
col = np.random.rand(100) * 100
size = np.random.rand(100) * 1000
plt.scatter(x,y,
c=col, # Farbwerte werden hier abgebildet
s=size, # Größe wird hier abgebildet
marker="*",
alpha=0.5) # Transparenz der Punkte
plt.grid(True)
plt.show()
```



Histogramm:

```
#Normalverteilung (Mittelwert, Standardabweichung, Anzahl)
x = np.random.normal(170, 10, 1000)
plt.hist(x, bins=20) #bins: Wie viele Balken
plt.savefig("out/m7.png")
plt.show()
```



Weitere Möglichkeiten unter: `https://matplotlib.org/stable/gallery/index.html`

Kleine Beispielauswertung - Wetterdaten London

- Das Beispiel befindet sich unter: `ex_02_matplotlib_sample_session.py`
- `NumPy` soll eigentlich nur eine Basisbibliothek sein und ist nicht direkt dafür gedacht, Datensätze auszuwerten
- Zu diesem Zweck wird später `Pandas` verwendet werden

```
1  import numpy as np
2  from matplotlib import pyplot as plt
3
4  d = np.genfromtxt('data/london_weather.csv', delimiter=",", skip_header=1 )
5
6  dt =  d[:,0] #Datum mit folgendem Aufbau: 19790103 (3.Jänner 1979)
7  # Aufteilen in Tag, Monat, Jahr
8  day = (dt % 100).astype('i')
9  month = (dt % 10000 / 100).astype('i')
10 year = (dt % 100000000 / 10000).astype('i')
11
12 # Check ob es funktioniert hat
13 print("Jahr:", np.unique(year, return_counts=True))
14 print("Monat", np.unique(month, return_counts=True))
15 print("Tag:", np.unique(day, return_counts=True))
16 print("Jahr MIN MAX" , np.min(year), np.max(year))
17
18 sun = d[:,2] # Sonnenstunden
19 print (sun)
20
21 #PLausibilitätscheck
22 print("Sun MIN MAX" , np.min(sun), np.max(sun))
23 plt.boxplot(sun)
24 plt.show()
25
26 sun1979 = sun[year == 1979] #Holen der Sonnenstunden im Jahr 1979
27 sun2020 = sun[year == 2020]
28 plt.boxplot([sun1979, sun2020]) #Gegenüberstellung der Sonnenstunden
29 plt.show()
```