

2

Numpy

- Numpy ist ein Modul, welches die Grundlage für den Umgang mit Daten liefert
- neuer Datentyp ndarray (n-dimensional array): Vektor, Matrix, ...
- Berechnungen viel effizienter, da im Gegensatz zur Liste nur ein Datentyp



Vektor
(1-dim)

1
2
3
4
5

Matrix
(2-dim)

1	6	11
2	7	12
3	8	13
4	9	14
5	10	15

Würfel
(3-dim)

1	6	11
2	7	12
3	8	13
4	9	14
5	10	15

...

In der Datenanalyse verwenden wir meistens Tabellen, bei denen jede Spalte ein Vektor ist.

Ein Bild kann als 3-dimensionales Array dargestellt werden



=

	128	89	168	219	222
	60	23	200	210	220
	65	34	180	187	43
	128	89	168	219	222
	60	23	200	210	220

3 Ebenen (RGB = rot, grün, blau) mit jeweils $n \times m$ Farbwerten

- Jedes Element eines ndarray ist ein dtype object (data-type object)

```
import numpy as np
a = np.array([1, 3, 5, 7])
print(type(a))
print(a[1])
print(type(a[1]))
print(a.shape)           # Dimension des Array
```

- Zwei Dimensionen aus verschachtelten Listen

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

- Zugriff über eckige Klammern

```
b[1,2]
```

- Der Datentyp der Werte wird automatisch festgelegt oder übergeben

```
b = np.array([[1,2],[3,4],[5,6]], dtype="float64")
```

- Der Speicherverbrauch unterscheidet sich

```
import sys
a = np.arange(1000.0)
b = np.arange(1000)
print(sys.getsizeof(a))
print(sys.getsizeof(b))
```

dtype	Beschreibung
bool_	Standard Boolean
int_	Standard Integer
int8, int16, int32, int64	8-bit Integer (-128 bis 127), 16-/32-/64-Bit Integer
uint8, uint16, uint32, int64	unsigned 8-bit Integer (0 bis 255), 16-/32-/64-bit uint
float16, float32, float64	16-/32-64-Bit float (half-precision, single precision, double precision)
str_	Standard String

- Bei mehreren Datentypen Rückfall auf allgemeinen Datentyp object (möglich, aber Warnung)

```
c = np.array([1, "Hallo", 5.5, [1,23]])
```

- Es können auch eigene, zusammengesetzte Datentypen erzeugt werden. Strings benötigen eine maximale Länge (ansonsten object benutzen).

```
student=np.dtype([('name','U20'), ('alter', 'int_'),  
                  ('punktzahl', 'float_')])  
print(student)
```

```
klasse = np.array([('Klaus', 21, 50.5),('Maria', 18, 75.3)],  
                  dtype=student)  
print(klasse)
```

- **shape** liefert die Form des Arrays, **ndim** die Anzahl Dimensionen
- Mit **arange()** wird eine Zahlenfolge erzeugt (wie range)
- Mit **reshape()** kann ein Array umgeformt werden
- **np.zeros()** erzeugt ein mit Nullen gefülltes Array, **np.ones()** ein mit Einsen gefülltes

- Mit dem sogenannten Slicing lassen sich Teile eines Arrays extrahieren oder bearbeiten.
- Hinter dem Variablennamen werden in eckigen Klammern die einzelnen Dimensionen angesprochen

Syntax	Beschreibung
<code>a[0]</code> , <code>a[0,0]</code> , <code>a[0,0,0]</code>	Das erste Element eines 1-/2-/3-dimensionalen Arrays
<code>a[0,:]</code>	die erste Zeile eines 2D-Arrays
<code>a[:,1]</code>	die zweite Spalte eines 2D-Array
<code>a[0, 0:2]</code>	die ersten beiden Spalten der ersten Zeile (bei 0:2 ist der Index 2 nicht enthalten)
<code>a[-1, :]</code>	die letzte Zeile
<code>a[1:3, 2:]</code>	Zeilen 2 und 3, alle Spalten ab der dritten

- Erzeugung eines booleschen Arrays mit einer Bedingung

```
b = a>3
```

- Filterung mittels Bedingungen ist leicht für Vektoren. Bei höherdimensionalen Arrays wird in einen Vektor umgewandelt

```
a = np.arange(12)  
a[]
```

- Verknüpfung von Bedingungen mit & (und), | (oder), ~ (nicht) und ^ (xor). And, or und not funktionieren hier nicht.
- Die Funktionen **np.any()** und **np.all()** geben einen booleschen Wert zurück

- Operationen erfolgen elementweise

```
a = np.arange(12).reshape(3,4)
```

```
a + 3
```

```
a * 2
```

```
b = np.arange(11,-1,-1).reshape(3,4)
```

```
print(a*b)
```

- Referenz `b = a` vs. deep copy `b = a.copy()`

- Unterschied zwischen **sum(a)** und **a.sum()**
- Die Funktionen **max()** und **min()** liefern Maximum und Minimum, **argmax()** und **argmin()** den zugehörigen Index
- Runden mittels **round()**, **floor()** oder **ceil()**
`a.round(2)`
- Sortieren mit **sort()**
`a.sort()`
- Bedingung mit **where()**
`np.where(a < 5, a, 10*a)`

- Numpy besitzt einen Pseudozufallsgenerator im Untermodul `numpy.random`. Der Aufruf wurde vor Kurzem geändert.

```
from numpy.random import default_rng  
rng = default_rng()  
zufall = rng.standard_normal(10)
```

```
# veraltet  
from numpy import random  
zufall = random.standard_normal(10)
```

- Mit der Standardbibliothek von Python im Modul `random` geht das übrigens auch, aber pro Zahl, was nicht so effizient ist

```
import random  
random.normalvariate(0, 1)
```

- Initialisierung des Zufallsgenerators mit einem Initialwert (Seed)

```
from numpy.random import default_rng  
rng = default_rng(seed = 42)
```
- Erzeugung von verschiedenen Zufallswerten

```
std_normal = rng.standard_normal(10)  
ganze_zahlen = rng.integers(low=1, high=10, size = 4)  
kommazahlen = rng.random(size=10)  
dna = rng.choice(["A","C","G","T"], size = 6, replace=True)  
dna2 = rng.permutation(dna)
```

- Komponentenweise Verrechnung ist viel schneller als eine Schleife

```
%timeit s1 + s2
```

```
%timeit [s1[i] + s2[i] for i in range(n)]
```

```
%timeit np.maximum(s1,s2)
```

```
%timeit [max(s1[i],s2[i]) for i in range(n)]
```

- numpy ist so schnell, weil
 - ein np-Array nur einen Datentyp enthält
 - so viel wie möglich parallel verarbeitet wird
 - kritische Teile in C, C++ und Fortran geschrieben sind