



Python NumPy

Programmierkurs 2 Data Science WS24/25

Leonard Traeger
M. Sc. Information Systems
leonard.traeger@fh-dortmund.de



Learning Goals Python NumPy

- **Explain** the advantages and disadvantages of Python's dynamic referencing.
- **Describe** NumPy's array datatype casting alternatives and **point out** suitable uniform datatypes given sequence examples e.g. [Ford, BMW, 42].
- Create one- and multi-dimensional NumPy arrays and **apply** Indexing, Boolean Masking, any and all Comparison, Slicing, and Reshaping.
- **Explain** the concept of NumPy's Vectorization via arithmetic Ufuncs and demonstrate why it is more „pythonic“ and give reasons why it is more efficient than without.
- **Summarize** what SciPy's library offers and **give an example** for a mathematical function you could use in your data science project instead of deploying it yourself.
- **Outline** and **explain** common data summary metrics and **apply** these on numeric-typed NumPy arrays.
- **Demonstrate** how same-length NumPy arrays in relationship can be conveniently arranged so values can be accessed either by index or by name.

Variable and Datatypes (Recap Python I)

Assignment uses dynamic referencing.

- The type/class is determined from the value, not declared.
- Type/class information belongs to the data, not the name bound to that data.

```
x = 10000
```

- x is not just a “raw” integer.
- x is a pointer to a compound C structure, which contains several values.
- Dynamic referencing in Python is more flexible but also more time and space consuming than compared to raw C.

Price of Python Flexibility

- Python's array object provides efficient storage of array-based data.
 - We can even create heterogeneous lists, tuples, sets, and dictionaries:

```
l_list = ["Zuckerberg", "Musk", 42]
t_tuple = ('Zuckerberg', 42, 'Gates', 'Bezos')
s_set = {'Zuckerberg', 42, 'Gates', 'Bezos'}
d_dictionary = {'person': 2, 'cat': 4, 'spider': 8}
```

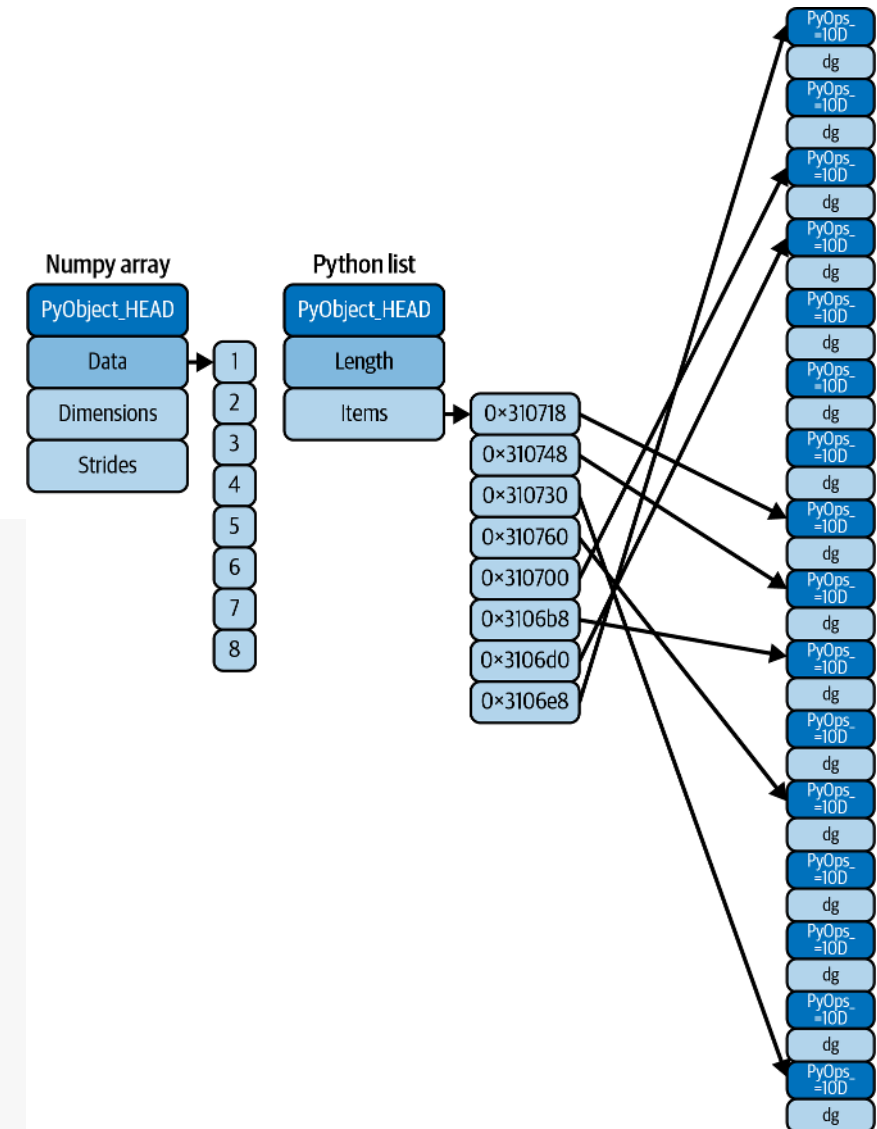
```
[type(item) for item in l_list]
#Prints "[str, str, int]"
```

- Flexibility comes at a **cost**: to allow these flexible types, **each item** in the list must contain its **own type, reference count**, and other information.

Containers and NumPy

- **Special case** that all **variables** are of the **same type**, much of information is redundant
- Motivation: **data** in a **fixed-type array** can be much more efficient to store.

```
np_arr = np.array(range(8))
print(np_arr)           #Prints "[0 1 2 3 4 5 6 7]"
print(np_arr.dtype)     #Prints "int64"
np_arr2 = np.array(range(8), dtype=np.float32)
print(np_arr2)          #Prints "[0 1 2 3 4 5 6 7]"
print(np_arr2.dtype)    #Prints "float32"
np_arr3 = np.asarray(l_list2)
print(np_arr3)          #Prints "[0 1 2 3 4 5 6 7]"
print(np_arr3.dtype)    #Prints "int64"
np_arr4 = np.asarray(["Zuckerberg", "Musk", 42])
print(np_arr4)          #Prints "['Zuckerberg' 'Musk' '42']"
print(np_arr4.dtype)    #Prints "<U21"
```



NumPy and Datatypes

- ... tries to guess a datatype when you create an array.
- ... but includes an optional argument to explicitly specify the datatype.
 - `np.array([1,2,3], dtype=np.float32)`

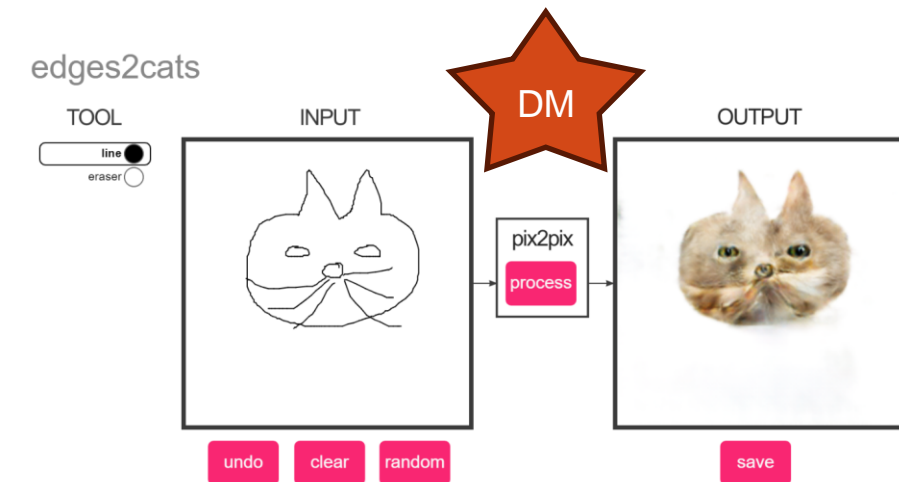
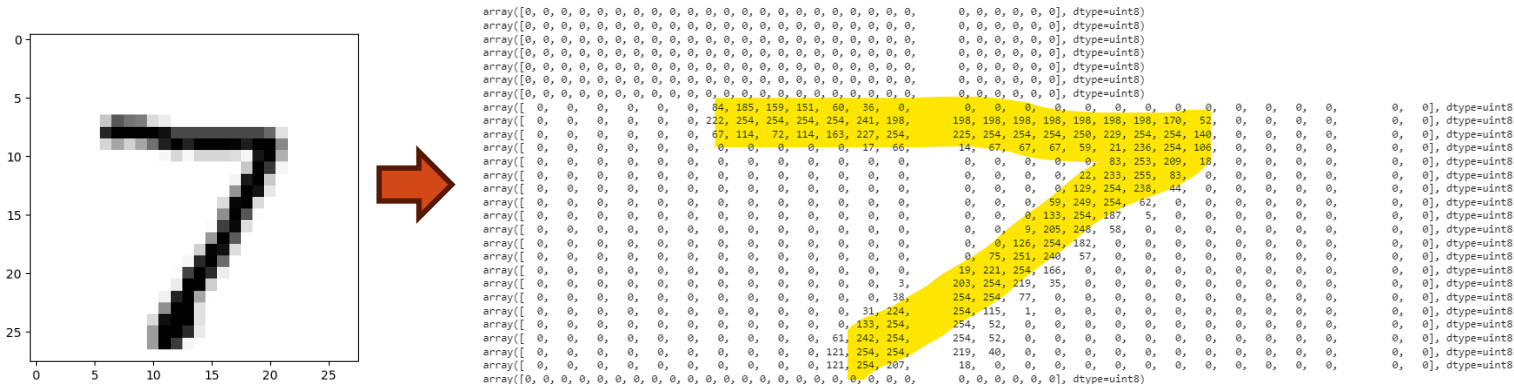
Table 12-1. NumPy data types

Character	Description	Example
'b'	Byte	<code>np.dtype('b')</code>
'i'	Signed integer	<code>np.dtype('i4') == np.int32</code>
'u'	Unsigned integer	<code>np.dtype('u1') == np.uint8</code>
'f'	Floating point	<code>np.dtype('f8') == np.float64</code>
'c'	Complex floating point	<code>np.dtype('c16') == np.complex128</code>
'S', 'a'	String	<code>np.dtype('S5')</code>
'U'	Unicode string	<code>np.dtype('U') == np.str_</code>
'V'	Raw data (void)	<code>np.dtype('V') == np.void</code>

VanderPlas, J., "Python Data Science Handbook", O'Reilly, 2017

NumPy short for "Numerical Python"

- Stores data **efficiently** in a **uniform fixed-type array**.
- **NumPy** arrays can be **multidimensional**.
- **NumPy** adds **efficient manipulation** and **operations** on that data.
- Core library for scientific computing.



<https://affinelayer.com/pixsrv/index.html>

- Images, sound, text,... can be converted into numerical representations.

NumPy Arrays

- Initialize NumPy arrays from Python lists.
- **One-dimensional index:** i'th value (counting from zero) can be accessed by specifying the desired index in square brackets

```
a = np.array([1, 2, 3])    # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"
```

```
b = np.array([[1,2,3],[4,5,6]]) # Create a rank 2 array
print(b.shape)                  # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0]) # Prints "1 2 4"
```

b	X = 0	X = 1	X = 2
Y = 0	1	2	3
Y = 1	4	5	6

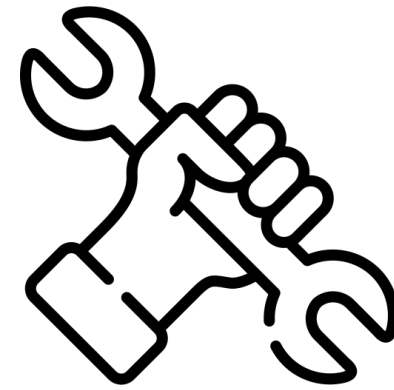
NumPy Arrays (cont.)

- **Multi-dimensional index:** items can be accessed using a comma-separated (row,column).

b	X = 0	X = 1	X = 2
Y = 0	1	2	3
Y = 1	4	5	6

- To index from the end of the array, you can use negative indices:
`print(b[0,-1]) : '3'`
- Values can also be modified using any of the preceding index notation.

Training #1



How does the NumPy array look like after the modifications?

b	X = 0	X = 1	X = 2
Y = 0	1	2	3
Y = 1	4	5	6

```
b[0, 2] = 9
b[1, -2] = 42
print(b)
```

NumPy Boolean Masking

- Select the elements of an array that satisfy some condition.

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

bool_idx = (b > 2)    # Find the elements of a that are bigger than 2;
                      # this returns a numpy array of Booleans of the same
                      # shape as a, where each slot of bool_idx tells
                      # whether that element of a is > 2.

print(bool_idx)       # Prints "[[False False  True]
                      #           [ True  True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(b[bool_idx])    # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(b[b > 2])
```

NumPy Comparison

Operator	Equivalent ufunc	Operator	Equivalent ufunc
==	np.equal	!=	np.not_equal
<	np.less	<=	np.less_equal
>	np.greater	>=	np.greater_equal

- Checking whether any or all the values are True of an NumPy array

```
c = np.array([[9, 4, 0, 3], [8, 6, 3, 1], [3, 7, 4, 0]])

print(np.count_nonzero(c))           # Prints "10"
print(np.any(c > 8))                 # Prints "True"
print(np.all(c < 10))                # Prints "True"
print(np.any(c == 2))                 # Prints "False"
print(np.any(c == 2) or np.all(c >= 0)) # Prints "True"

# np.all and np.any can be used along particular axes
print(np.any(c > 7, axis=1))
```

NumPy Slicing

- Accessing subarrays of **one-dimensional** array `x`, use: `x[start:stop:step]`
- Default values: `start=0`, `stop=<size of dimension>`, `step=1`

```
a = np.array([1, 2, 3, 4, 5, 6, 7, 8])
print(a[:3])           #Prints first three elements "[1 2 3]"
print(a[3:])           #Prints # elements after index 3 "[4 5 6 7 8]"
print(a[1:4])          #Prints middle subarray "[2 3 4]"
print(a[::2])          #Prints every second element "[1 3 5 7]"
print(a[::-1])         #In case step value is negative
                      #defaults for start and stop are swapped
                      #Prints reverse "[8 7 6 5 4 3 2 1]"
```


NumPy Slicing (cont.)

veg[0]				
veg[1]				
veg[2]				

- **Multi-dimensional slices** work the same by multiple slices **separated by commas**.

```
b = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#   [ 5  6  7  8]
#   [ 9 10 11 12]]

print(b[:2, :3])      # first two rows & three columns
# [[1 2 3]
#   [5 6 7]]

print(b[:3, ::2])     # three rows, every second column
# [[ 1  3]
#   [ 5  7]
#   [ 9 11]]

print(b[::-1, ::-1])
#
#
#
```

NumPy Reshape

- E.g. transform a one dimensional numpy array into a multi-dimensional shape or vice versa.
- **Requires the size of the initial array to match the size of the reshaped array.**

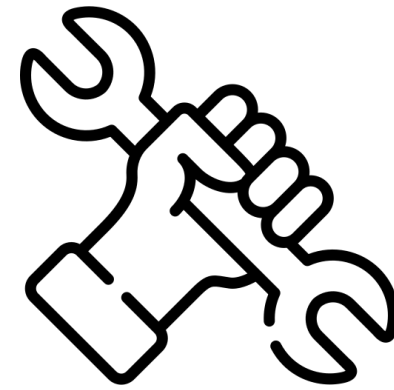
```
grid = np.arange(1, 10)
print(grid.shape)    #Print "9,"
print(grid)          #Print "[1, 2, 3, 4, 5, 6, 7, 8, 9]"

grid1 = np.arange(1, 10).reshape(3, 3)
print(grid1.shape)   #Print "3,3"
print(grid1)         #Print "[[1 2 3]
                        #      [4 5 6]
                        #      [7 8 9]]"

grid2 = grid1.reshape(9,) #reverse reshape
print(grid2)            #Print "[1, 2, 3, 4, 5, 6, 7, 8, 9]"

grid3 = grid2.reshape(9,1) #column vector via reshape
print(grid3)

grid_fail = np.arange(1, 11).reshape(3, 4)
                # "cannot reshape array of size 10 into shape (3,3)"
```



Training #2

Cascadingly program the following steps:

1. Generate a numpy array `x` with the sequence `[1,2,...,100]`.
2. Reshape the numpy array to an numpy array with 10 rows `x1`.
3. Slice the sixth row as a view `x2` and
4. Loop over each item and add 100.
 - Use a `for` loop and `enumerate(x2)`
5. Print `x`

```
[ 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 50 151 152 153 154
155 156 157 158 159 160 61 62 63 64 65 66 67 68 69 70 71 72
 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
 91 92 93 94 95 96 97 98 99 100]
```

NumPy Subarrays as No-Copy Views

- NumPy array **slices** are returned as **views**.
- **Modifying subarray view changes original array.**

```
a = np.array([1, 2, 3, 4, 5, 6, 7, 8])
b = a[1:4]
b2 = a[1:4].copy()
print(b)          #Prints middle subarray "[2 3 4]"
b[0] = 42
b2[-1] = 1000
print(b)          #Prints middle subarray "[42 3 4]"
print(b2)         #Prints middle subarray "[ 2 3 1000]"
print(a)          #Prints middle subarray "[ 1 42 3 4 5 6 7 8]"
```

- Use `.copy()` to copy the underlying data buffer to avoid this behavior.

Similar case in
pandas

NumPy Ufuncs

- So far we have covered the basics of **NumPy**, but its actual **power** comes from **functions** which **optimize** (faster) the **computation**.
- Key to make repeated calculations on array elements much more efficient:
 - **Vectorization**: functions that encapsulate loops!
- A **function or operator** written on an **entire sequence at once** is said to be vectorized.
- The vectorized approach is designed to **push the loop into** the **compiled layer** that **underlies NumPy**, leading to much faster execution.
- Examples:
 - Looping over arrays to index on each element
 - Element-wise division
 - ...

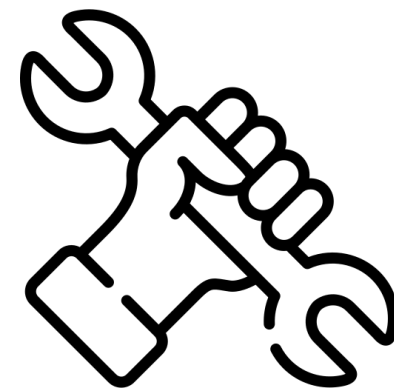
NumPy Ufuncs (arithmetic)

- Vectorization pushes loops into the compiled layer of NumPy arrays.
- Includes standard addition, subtraction, multiplication, and division.

```
x = np.array([0, 1, 2, 3])
print(x + 5) #Prints "[5 6 7 8]"
print(x - 5) #Prints "[-5 -4 -3 -2]"
print(x * 2) #Prints "[0 2 4 6]"
print(x / 2) #Prints "[0. 0.5 1. 1.5]"
print(x ** 2)
```

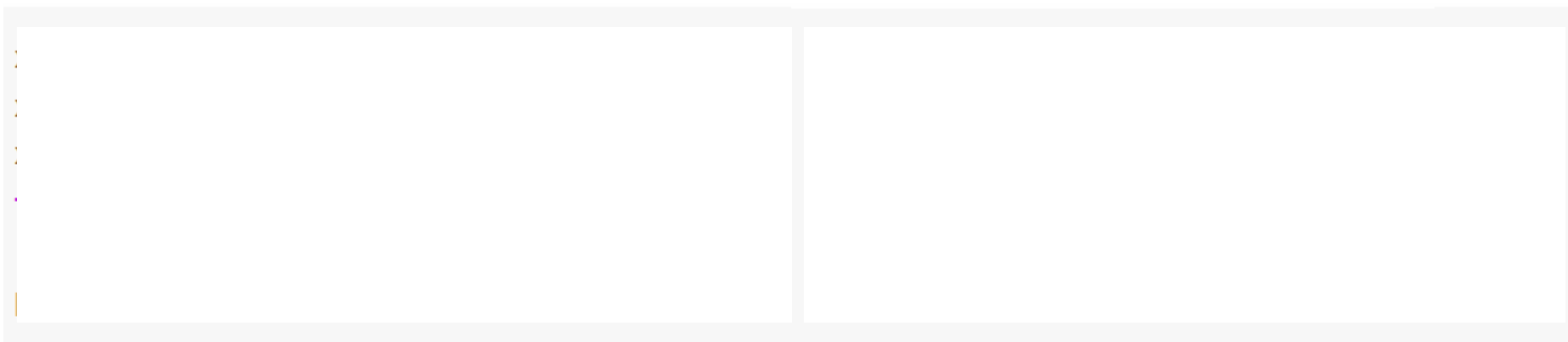
Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., $1 + 1 = 2$)
-	np.subtract	Subtraction (e.g., $3 - 2 = 1$)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., $2 * 3 = 6$)
/	np.divide	Division (e.g., $3 / 2 = 1.5$)
//	np.floor_divide	Floor division (e.g., $3 // 2 = 1$)
**	np.power	Exponentiation (e.g., $2 ** 3 = 8$)
%	np.mod	Modulus/remainder (e.g., $9 \% 4 = 1$)

VanderPlas, J., "Python Data Science Handbook", O'Reilly, 2017



NumPy Ufuncs (cont.)

Which version is more efficient and looks more “pythonic” ?



```
[ 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 50 151 152 153 154
155 156 157 158 159 160 61 62 63 64 65 66 67 68 69 70 71 72
 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
 91 92 93 94 95 96 97 98 99 100]
```

NumPy Ufuncs (cont.)

```
#trigonometric functions
theta = np.linspace(0, np.pi, 3)
print(theta)
print(np.sin(theta))
print(np.cos(theta))
print(np.tan(theta))
```

```
#inverse trigonometric functions
x = np.array([-1, 0, 1])
print(np.arcsin(x))
print(np.arccos(x))
print(np.arctan(x))
```

```
#exponents
x = np.array([1, 2, 3])
print(np.exp(x))      #e^x
print(np.exp2(x))     #2^x
print(np.power(3., x)) #3^x
```

```
#logarithms
print(np.log(x))      #ln(x)
print(np.log2(x))     #log2(x)
print(np.log10(x))    #log10(x)
```

- ...and many more such as comparison operations, conversions from radians to degrees, rounding and remainders.

SciPy Ufuncs

<https://docs.scipy.org/doc/scipy/tutorial/index.html#user-guide>

“**Extends NumPy** providing additional tools for **array computing** and provides specialized data structures, such as sparse matrices and k-dimensional trees.”

- If you want to compute some **mathematical function** on your data ...it is **very likely** that it is **implemented** in **SciPy**.
- E.g., special functions, integration, optimization, signal processing including filtering with convolution (images) and spectral analysis, eigenvalue problems, graph routines, **spatial data structures and algorithms**, statistics (probability density function, survival function, discrete and continuous distributions, sampling and Monte Carlo Methods), multidimensional image processing, ...and **many many more**.

Know that there is a library, but
memorizing functions is impossible...

Let's cover the **Euclidian distance** as one example ☺

Euclidean distance



- Formula to find the distance between two points on a plane.
- *What we intuitively measure when someone would ask us to take a distance measure.*

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2}$$

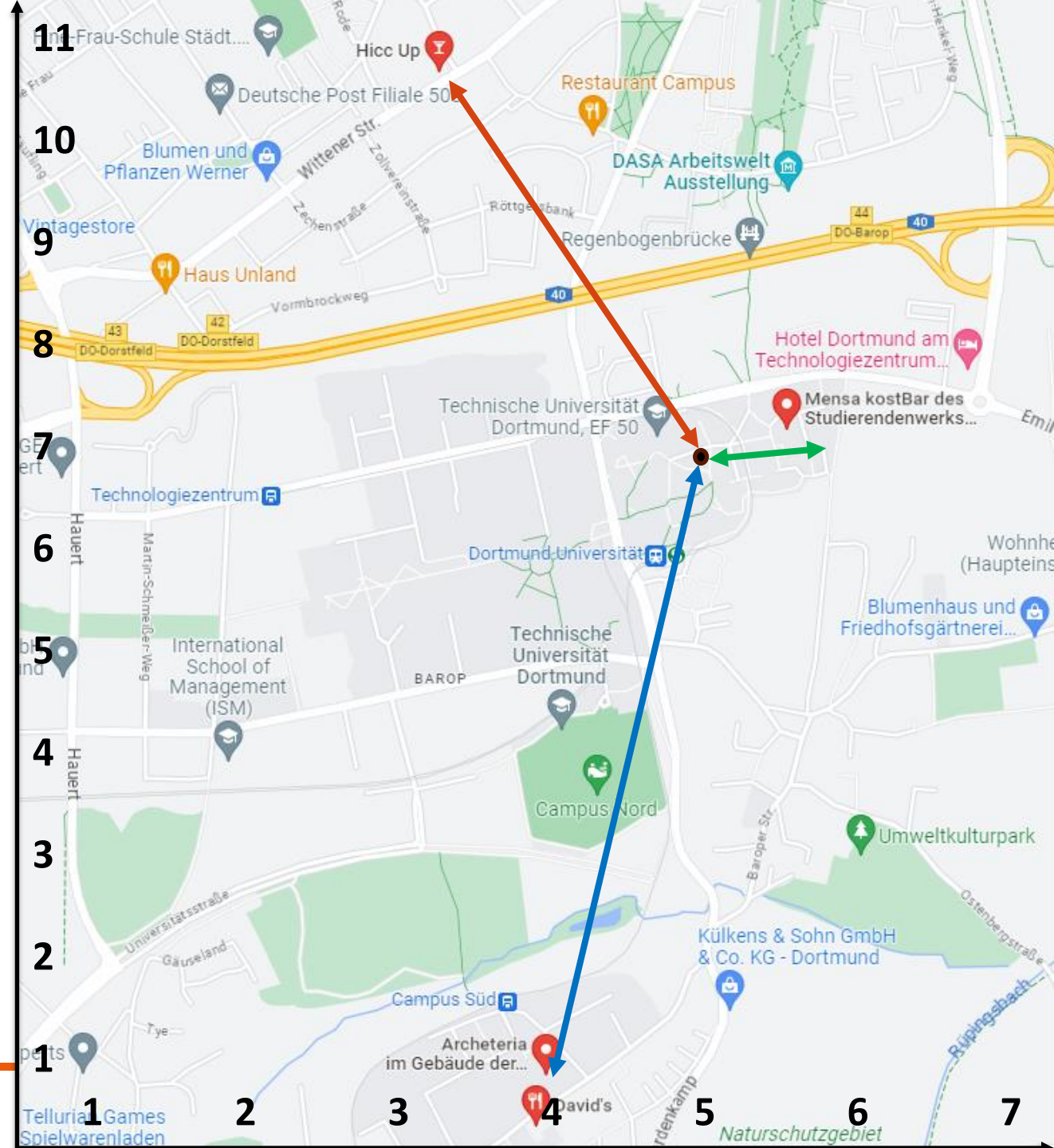
- **d** is a **function** with input of
- two same length **vectors p, q** (NumPy arrays)
- **Σ** (sigma) denotes a **sum loop** over **index i** (length of p or q vector)
- Index value of vector **qi** or **pi** is accessible (e.g., when $i = 1$, then $q[1]$ and $p[1]$)

Kaltgetränk

Standorte

- FH Dortmund = [5,7]
- Mensa = [6,7]
- Hicc Up = [3,11]
- David's = [4,1]

```
fhd = np.array([5,7])  
mensa = np.array([6,7])  
hiccup = np.array([3,11])  
davids = np.array([4,1])
```

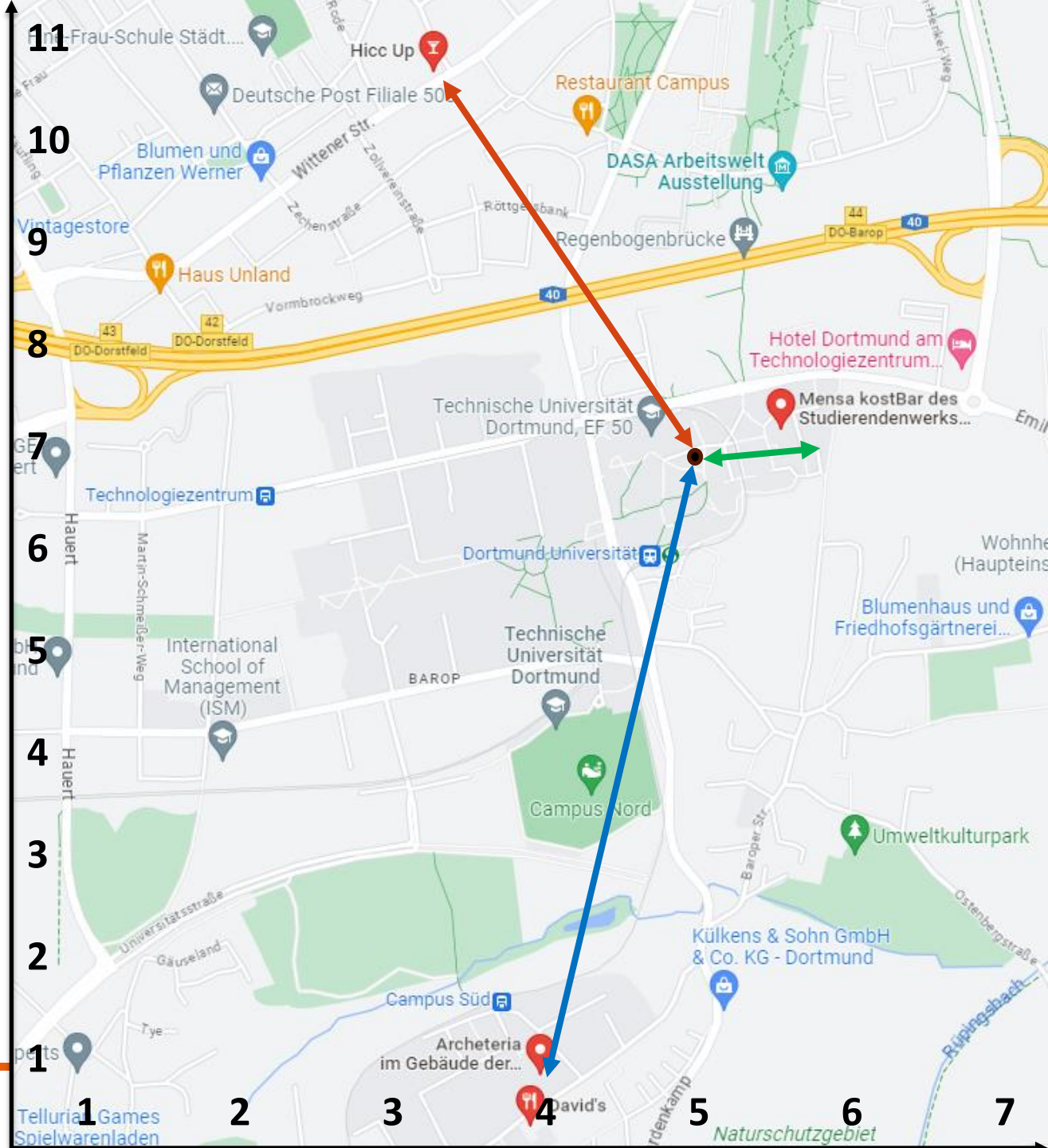


Kaltgetränk (cont.)

```
fhd = np.array([5,7])  
mensa = np.array([6,7])  
hiccup = np.array([3,11])  
davids = np.array([4,1])
```

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2}$$

```
def euclidean_distance(p, q):  
    if(len(p) == len(q)):  
        sum = 0  
        for i in range(len(p)):  
            sum = sum + (q[i]-p[i])**2  
        return np.sqrt(sum)
```

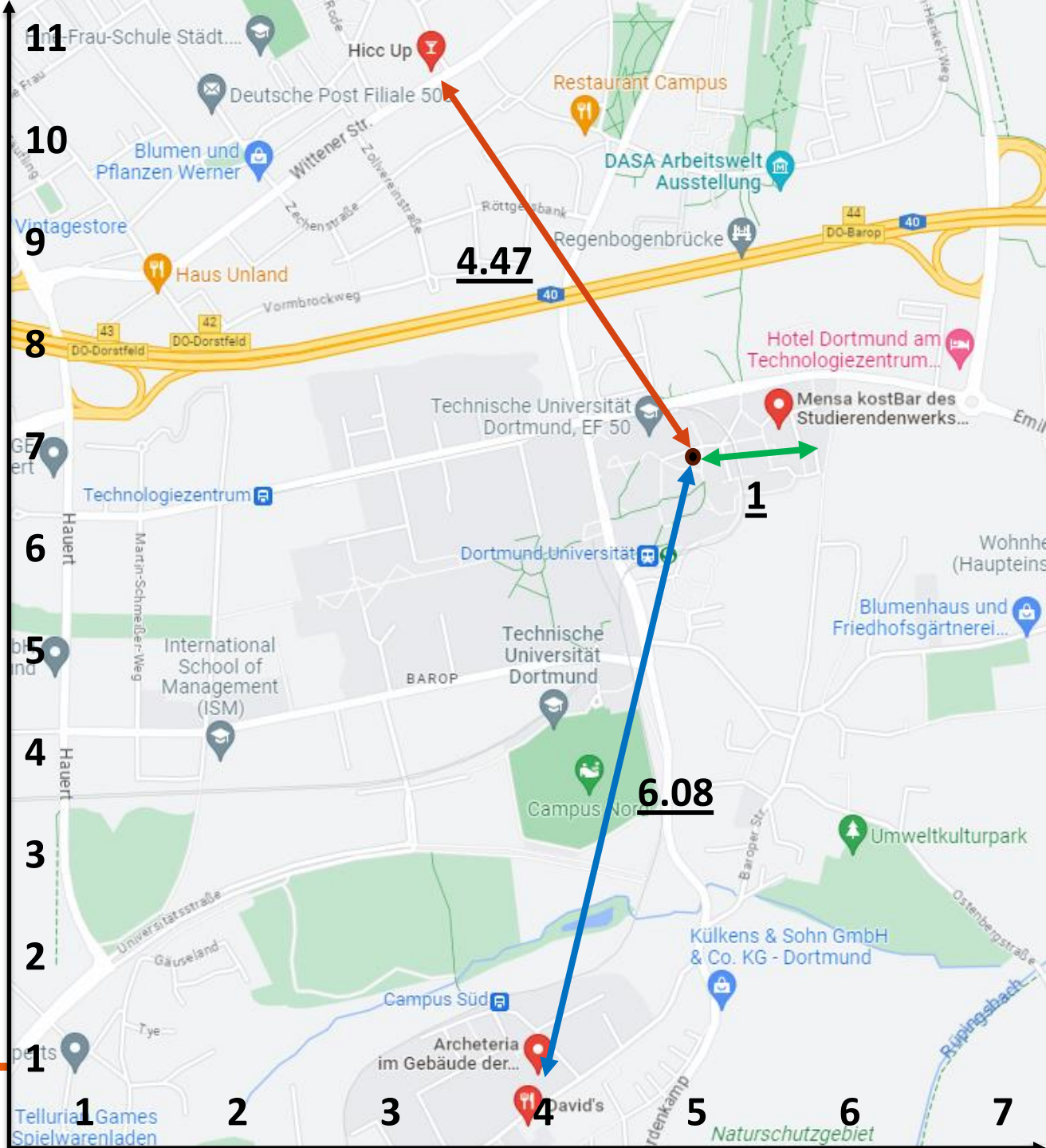


Kaltgetränk (cont.)

```
fhd = np.array([5,7])
mensa = np.array([6,7])
hiccup = np.array([3,11])
davids = np.array([4,1])
```

```
def euclidean_distance(p, q):
    if(len(p) == len(q)):
        sum = 0
        for i in range(len(p)):
            sum = sum + (q[i]-p[i])**2
        return np.sqrt(sum)
```

```
print(euclidean_distance(fhd, mensa)) #1.0
print(euclidean_distance(fhd, hiccup)) #4.47
print(euclidean_distance(fhd, davids)) #6.08
```



Kaltgetränk (cont.)

```
fhd = np.array([5,7])
mensa = np.array([6,7])
hiccup = np.array([3,11])
davids = np.array([4,1])
```



Installing User Guide [API reference](#) Building from source Development Release notes



scipy
scipy.cluster
scipy.constants
scipy.datasets
scipy.fft
scipy.fftpack
scipy.integrate

scipy.spatial.distance.euclidean

`scipy.spatial.distance.euclidean(u, v, w=None)`
Computes the Euclidean distance between two 1-D arrays.

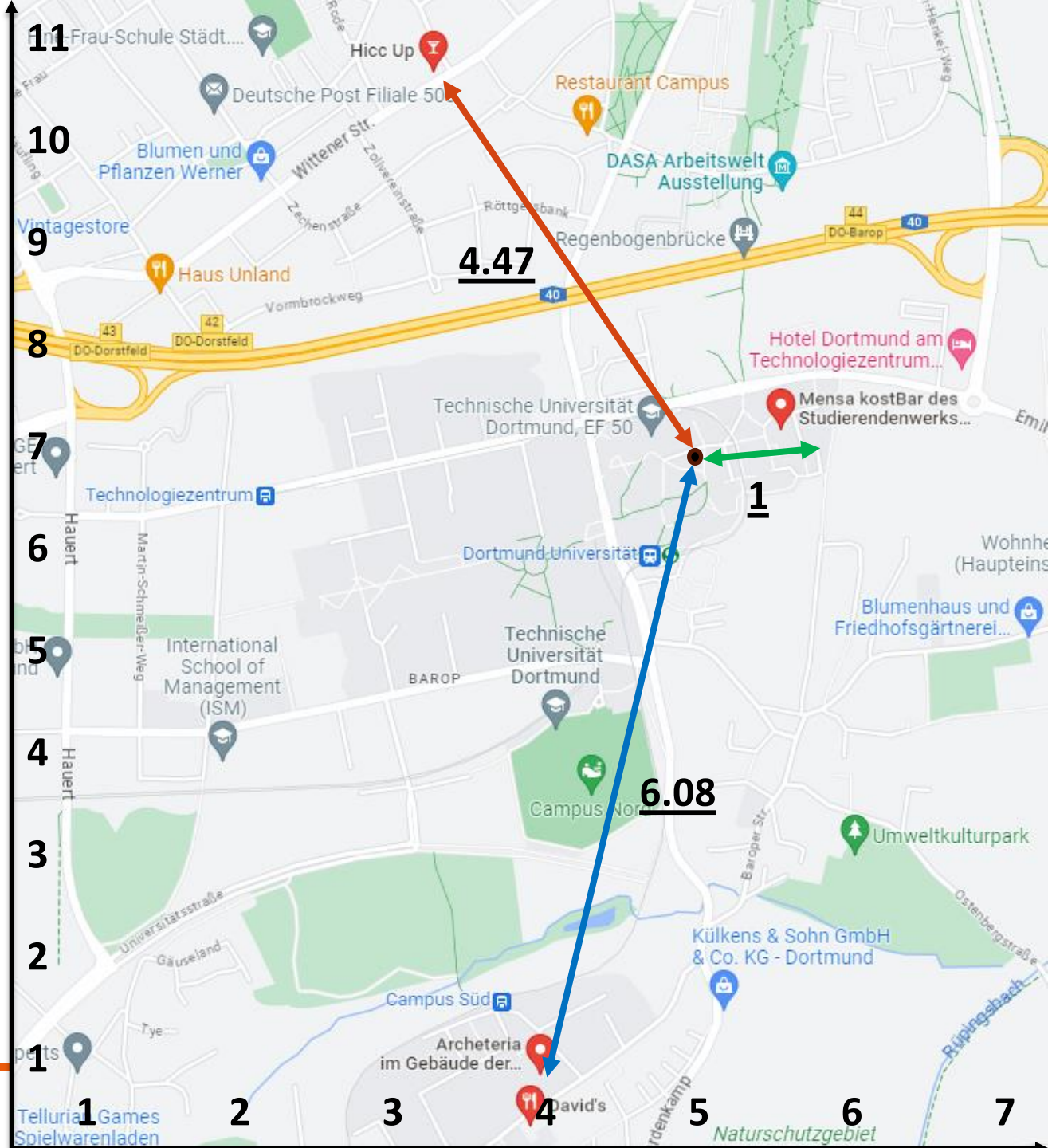
[\[source\]](#)

The Euclidean distance between 1-D arrays u and v , is defined as

$$\|u - v\|_2 = \left(\sum (w_i |u_i - v_i|^2) \right)^{1/2}$$

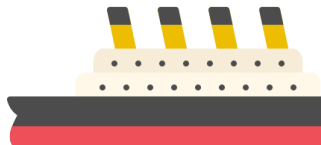
```
from scipy.spatial import distance
```

```
print(distance.euclidean(fhd, mensa)) #1.0
print(distance.euclidean(fhd, hiccup)) #4.47
print(distance.euclidean(fhd, davids)) #6.08
```



NumPy Aggregations

- **First step** in **exploring any dataset** is often to compute various **summary statistics**.
- Descriptive statistics summarize the **central tendency, dispersion** and **shape of a dataset's distribution** (excluding NaN values).
- Common summary statistics:
 - Mean
 - Median
 - Min
 - Max
 - Sum
 - Standard deviation
 - Quantiles



	Age	Fare
count	714.000000	891.000000
mean	29.699118	32.204208
std	14.526497	49.693429
min	0.420000	0.000000
25%	20.125000	7.910400
50%	28.000000	14.454200
75%	38.000000	31.000000
max	80.000000	512.329200

NumPy has fast built-in aggregation functions!

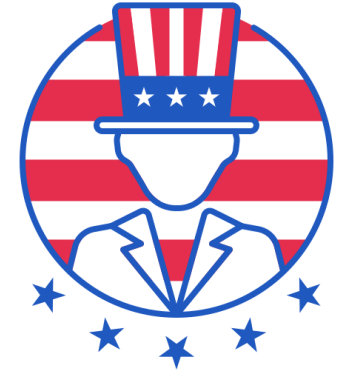
NumPy Aggregations (cont.)

- **Aggregation operation reduce the entire array to a single summarizing value!**

Function name	NaN-safe version	Description
<code>np.sum</code>	<code>np.nansum</code>	Compute sum of elements
<code>np.prod</code>	<code>np.nanprod</code>	Compute product of elements
<code>np.mean</code>	<code>np.nanmean</code>	Compute mean of elements
<code>np.std</code>	<code>np.nanstd</code>	Compute standard deviation
<code>np.var</code>	<code>np.nanvar</code>	Compute variance
<code>np.min</code>	<code>np.nanmin</code>	Find minimum value
<code>np.max</code>	<code>np.nanmax</code>	Find maximum value
<code>np.argmin</code>	<code>np.nanargmin</code>	Find index of minimum value
<code>np.argmax</code>	<code>np.nanargmax</code>	Find index of maximum value
<code>np.median</code>	<code>np.nanmedian</code>	Compute median of elements
<code>np.percentile</code>	<code>np.nanpercentile</code>	Compute rank-based statistics of elements
<code>np.any</code>	N/A	Evaluate whether any elements are true
<code>np.all</code>	N/A	Evaluate whether all elements are true

Most of these aggregations are nicely integrated in the „box-plot“ diagram!

VanderPlas, J., "Python Data Science Handbook", O'Reilly, 2017



NumPy Aggregations (cont.)

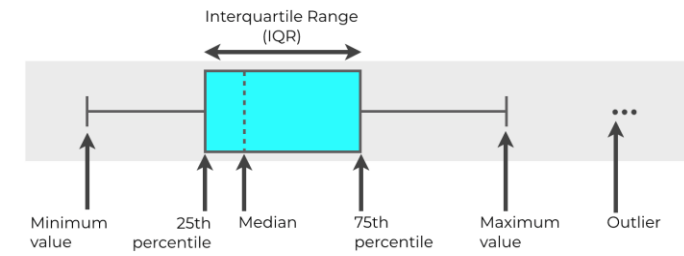
What is the **data summary** of heights of US presidents?

```
usp_height = np.array([189, 170, 189, 163, 183, 171, 185, 168, 173, 183,  
                       173, 173, 175, 178, 183, 193, 178, 173, 174, 183,  
                       183, 168, 170, 178, 182, 180, 183, 178, 182, 188,  
                       175, 179, 183, 193, 182, 183, 177, 185, 188, 188,  
                       182, 185, 191, 182])
```

```
print("Mean height: ", usp_height.mean())      #Prints "180.04.."
print("Standard deviation:", usp_height.std())  #Prints "6.983.."
print("Minimum height: ", usp_height.min())     #Prints "163"
print("Maximum height: ", usp_height.max())     #Prints "193"
print("25th: ", np.percentile(usp_height, 25)) #Prints "174.75"
print("Median: ", np.median(usp_height))        #Prints "182.0"
print("75th: ", np.percentile(usp_height, 75)) #Prints "183.5"
```

Also available in
`Pandas.DataFrame.describe`

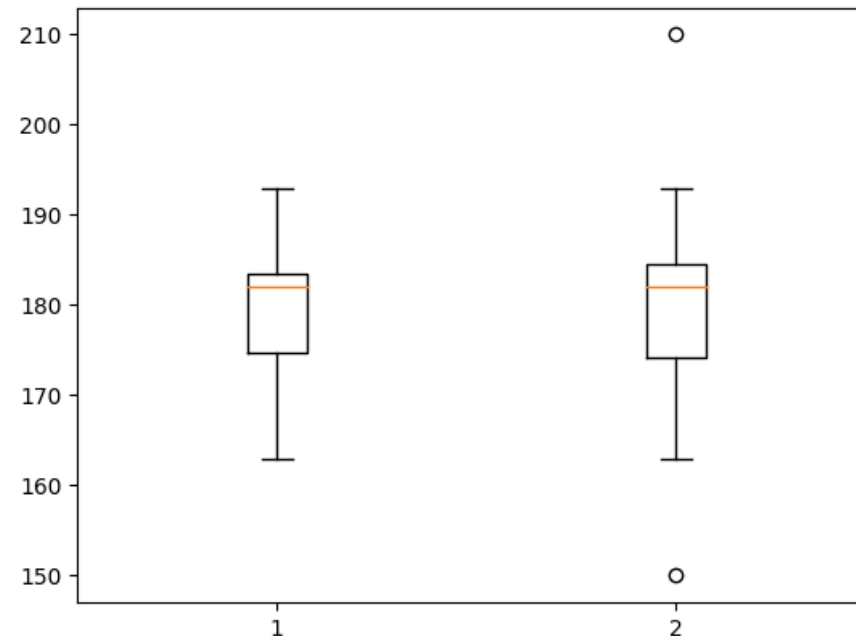
Preview: Boxplot



<https://www.sharpsightlabs.com/blog/ggplot-boxplot/>

```
#average heights of US presidents
usp_height = np.array([189, 170, 189, 163, 183, 171, 185, 168, 173, 183,
                      173, 173, 175, 178, 183, 193, 178, 173, 174, 183,
                      183, 168, 170, 178, 182, 180, 183, 178, 182, 188,
                      175, 179, 183, 193, 182, 183, 177, 185, 188, 188,
                      182, 185, 191, 182])
```

```
#average heights of US presidents
usp_height_2 = np.array([189, 170, 189, 163, 183, 171, 185, 168, 173, 183,
                        173, 173, 175, 178, 183, 193, 178, 173, 174, 183,
                        183, 168, 170, 178, 182, 180, 183, 178, 182, 188,
                        175, 179, 183, 193, 182, 183, 177, 185, 188, 188,
                        182, 185, 191, 182,
                        150, 210]) #some outliers
```



Discussion about whether outliers should remain in DataFrame and subsequent analysis. Documentation!!!

Python Sorting

- In Algorithms 1 you probably have had dreams or nightmares about *insertion sorts*, *selection sorts*, *merge sorts*, *quick sorts*, *bubble sorts*, and many, many more...
- **All algorithms accomplish the same** (in different efficiency depending on the case): sorting the values in a list or array...

```
print(sorted(usp_height))    # returns a sorted copy
usp_height.sort()           # acts in-place and returns None
print(usp_height)
#Both print "[163, 168, 168, 170, 170, 171, 173, 173, 173, 173, 174, 175,
#           175, 177, 178, 178, 178, 178, 179, 180, 182, 182, 182, 182,
#           182, 183, 183, 183, 183, 183, 183, 183, 183, 185, 185, 185,
#           188, 188, 188, 189, 189, 191, 193, 193]"

print(sorted('FH Dortmund')) # Python can also sort a string
#Prints "[' ', 'D', 'F', 'H', 'd', 'm', 'n', 'o', 'r', 't', 'u']"
```

NumPy Sorting 😊

- `np.sort` function is analogous to Python's built-in `sorted` function, but vectorized.



```
print(np.sort(usp_height)) # returns a sorted copy
usp_height.sort()          # acts in-place and returns None
print(usp_height)
```

- In multi-dimensional arrays, you can define the sorting axis (column = 0, row = 1):

```
md_array = np.array([[3,5,4],[2,0,1]])
print(np.sort(md_array, axis=0))
# Prints "[[2 0 1]
#           [3 5 4]]"
print(np.sort(md_array, axis=1))
# Prints "[[3 4 5]
#           [0 1 2]]"
```

NumPy Structured Arrays

- Imagine several categories of data on people e.g., presidents.
- Store these in three **separate arrays** is clumsy and **technically unrelated**.

```
name = ['Biden', 'Trump', 'Obama']  
age = [78, 70, 47]  
height = [1.83, 1.90, 1.87]
```

- Let's create an empty structured array using a compound data type specification:

```
presidents = np.zeros(3, dtype={'names':('name', 'age', 'height'),  
                                'formats':('U10', 'i4', 'f8')})  
print(presidents.dtype)  
# Prints "[('name', '<U10'), ('age', '<i4'), ('height', '<f8')]"
```

index	name	age	height
0	Biden	78	1.83
1	Trump	70	1.9
2	Obama	47	1.87


NumPy Structured Arrays (cont.)

```
name = ['Biden', 'Trump', 'Obama']
age = [78, 70, 47]
height = [1.83, 1.90, 1.87]

presidents = np.zeros(3, dtype={'names':('name', 'age', 'height'),
                                'formats':('U10', 'i4', 'f8')})

print(presidents.dtype)
# Prints "[('name', '<U10'), ('age', '<i4'), ('height', '<f8')]"
```

We have
constructed a
relational table!



- Fill the array with our lists of values.

```
presidents['name'] = name
presidents['age'] = age
presidents['height'] = height
print(presidents)
# Prints "[('Biden', 78, 1.83) ('Trump', 70, 1.9 ) ('Obama', 47, 1.87)]"
```

- Data is now conveniently arranged in one structured array.

NumPy Structured Arrays (cont.)

- **Handy thing** with structured arrays: **refer to values** either **by index** or **by name**.

```
print(presidents['name'])
# Prints "array(['Biden', 'Trump', 'Obama'], dtype='<U10')"
```

```
print(presidents[0])
# Prints "('Biden', 78, 183.)"
```

```
print(presidents[-1]['name'])
# Prints "Obama"
```



```
print(presidents[presidents['height'] > 185])
# Prints "[('Trump', 70, 190.) ('Obama', 47, 187.)]"
```

```
print(presidents[presidents['height'] > 185]['name'])
# Prints "['Trump' 'Obama']"
```

- ...and use Boolean Masking.

This is how Pandas was born
on top of NumPy 😊

Training #3

name	#games	age
Roman Bürki	200	31
Marwin Hitz	60	34
Gregor Kobel	30	23

1. **Structured Array:** create a structured array based on the data of goalkeepers.
2. **Sorting:** Output the structured array sorted by age in descending order. *Hint: `[::-1]`*
3. **Boolean Masking:** Output all rows with goalkeepers who played more than 50 games.
4. **Aggregation:** Output the mean age of the goalkeepers above.

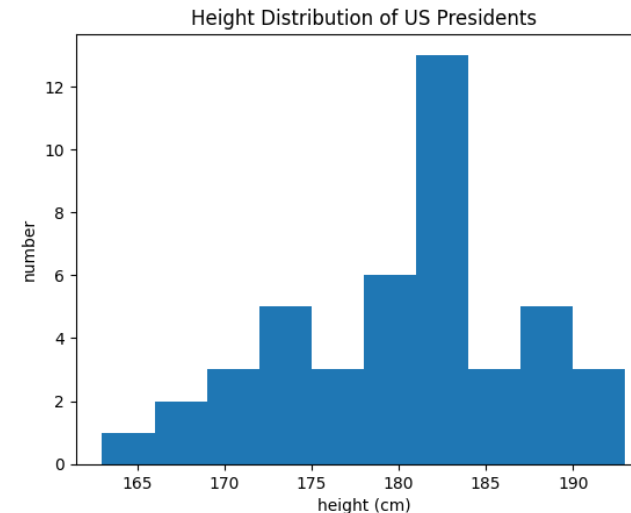
Takeaways

- NumPy is the backbone of scientific Python.
- NumPy offers vectorized and optimized implementations of many mathematical functions, a flexible array class with expressive slicing, helpful scalars, and much more.
- NumPy is a big topic, learn a little at a time.
- NumPy arrays are handy to map onto binary data formats in C, Fortran, or another language.
 - Otherwise: Pandas package is a much better day-to-day choice for structured data.

Outlook

- In the upcoming weeks 6 + 7 we will dive deep into Python Pandas.
- This is the core library you are going to use for wrangling data during your project.
- In week 9 we will dive deep into Python Matplotlib to plot data:
e.g. heights of US presidents.

index	name	age	height	terms
0	Biden	78	1.83	NaN
1	Trump	70	1.9	1
2	Obama	47	1.87	3



See you again in **two weeks** in class!

Questions?