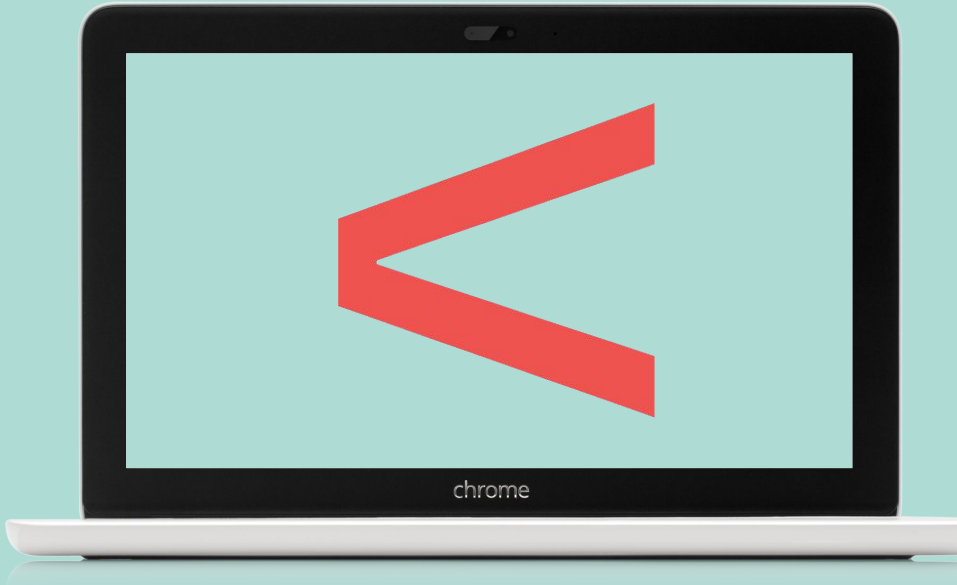
Python libraries: NumPy



What is NumPy?

NumPy is the fundamental library of python, used to perform scientific computing on **numerical data**.

NumPy provides **NumPy arrays**, a data structure that stores values of the same type that are indexed and allow to **perform calculations** across them. Think of a NumPy array as rows and columns of table.

What can NumPy do?

🔎 Inspecting and describing your array

🧮 Performing mathematical operations: statistics, arithmetics, comparison and aggregate functions

🔢 Generating random values

👥 Copying and sorting arrays

✂️ Subsetting, slicing and indexing

💠 Manipulating arrays

📊 Visualizing (matplotlib and seaborn)

🔮 Among other things

Subsetting and slicing with Python

String:

“h	e	l	l	o		w	o	r	l	d	!”
0	1	2	3	4	5	6	7	8	9	10	11
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Subsetting: selecting a single element

first_letter = hello[0]

“h”

Slicing: selecting or several elements

first_word = hello[0 : 5]

“hello”

Notice how we select one element up to the one we want

Subsetting and slicing with Python

List:

[“apple”,	“orange”,	“melon”,	“pear”,	“apple”,	“lemon”,	“grape”]
0	1	2	3	4	5	6

Subsetting: selecting a single element

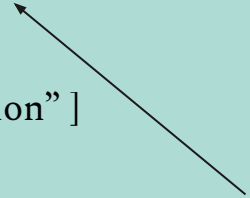
third_fruit = fruit[2]

“melon”

Slicing: selecting several elements

several_fruit = hello[0 : 3]

[“apple”, “orange”, “melon”]

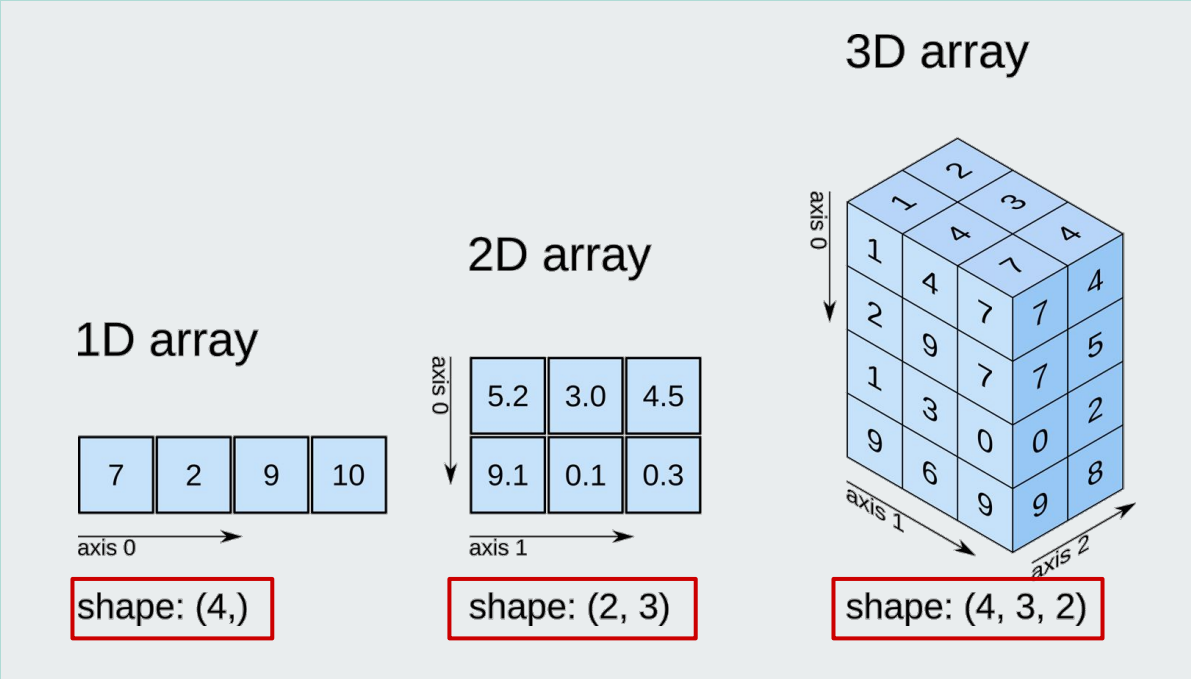


Notice how we select one element up to the one we want

What are the differences between Python lists and NumPy arrays?

Python list	NumPy array
Lists cannot directly handle math operations	You can perform calculations over entire arrays
List don’t need to be declared	Arrays need to be declared with np.array( ) to be created
Takes up a lot of memory space and execution time is slower	Consumes less memory and is faster at performing operations
Only have one dimension	Can have N-dimensions
Allows several data types	Not optimal for different data types

Dimensions in arrays - *n* dimensions



How to create an array from scratch?

1D array:

onedim_arr = np.array([1, 2, 3])

Result: array([1, 2, 3])

2D array:

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

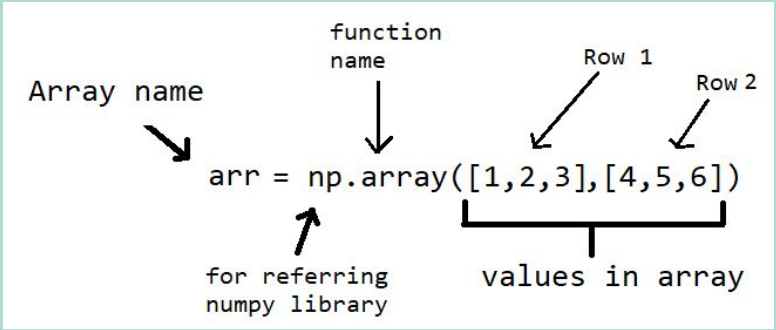
Result: array([[1, 2, 3],
[4, 5, 6]])

3D array:

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

Result: array([[[1, 2, 3],
[4, 5, 6]],

[[7, 8, 9],
[10, 11, 12]]])



How to create an array from a table?

Fruit	Apple	Grape	Banana	Melon	Lemon	Orange
Weight	10	2	3.5	4	1	6
Price	1.25	2.30	4.0	1.80	2.25	3.45
Discount	10	5	15	25	20	30

```
array( [ [  10,     2,  3.5,    4,     1,    6 ],
       [ 1.25,   2.3,    4,  1.8,  2.25, 3.45 ],
       [  10,     5,   15,   25,    20,   30 ] ] )
```

← Weight row

← Price row

← Discount row

How to know the size, shape and number of dimensions?

- **arr.ndim:** number of axes, or dimensions, of the array.
- **arr.size:** total number of elements of the array. This is the product of the elements of the array's shape.
- **arr.shape:** tuple of integers indicating the number of elements stored along each dimension of the array
  - This function is particularly useful when cleaning the data, as it will help you understand how many rows or columns have been dropped

Subsetting arrays

```
array( [ [   10,      2,    3.5,    4,      1,     6 ],
       [ 1.25,    2.3,      4,   1.8,   2.25,  3.45 ],
       [   10,      5,     15,    25,     20,    30 ]  ] )
```

← Weight row

← Price row

← Discount row

Subsetting: selecting a single element

weight_list = arr[0]

[10, 2, 3.5, 4, 1, 6]

third_weight = arr[0, 2]

[3.5]

Slicing arrays: rows

```
array( [ [    10,      2,    3.5,    4,      1,     6 ],
       [ 1.25,     2.3,     4,    1.8,   2.25,  3.45 ],
       [    10,      5,    15,    25,     20,    30 ]  ] )
```

← Weight row

← Price row

← Discount row

Slicing: selecting or several elements

price_discount = arr[1 :]

```
[ [ 1.25, 2.3,   4, 1.8, 2.25, 3.45 ],
  [    10,    5, 15,  25,    20,    30 ] ]
```

Slicing arrays: columns

```
array( [ [    10,        2,     3.5,     4,       1,      6 ],
         [ 1.25,      2.3,       4,   1.8,    2.25,  3.45 ],
         [    10,        5,      15,    25,      20,     30 ] ] )
```

If columns are not next to each other:
look at the brackets and comma in the syntax

first_third_columns= arr[:, [**0,2**]]

```
[ [ 10.    3.5 ]
  [ 1.25    4. ]
  [10.     15. ] ]
```

If columns are contiguous:
there no brackets and you separate with column

fourth_fifth_columns=arr[:, **3:5**]

```
[ [ 4.      1.  ]
  [ 1.8   2.25]
  [ 25.   20.  ] ]
```

Algebraic operations with operators on single array

twice_arr = arr * 2

```
array( [ [ 20. ,  4. ,   7. ,   8. ,   2. , 12.  ],
         [ 2.5,  4.6,   8. ,  3.6,  4.5,  6.9 ],
         [20. , 10. , 30. , 50. , 40. , 60.  ] ] )
```

Algebraic operations with operators between two arrays

sum_of_arrs = arr + twice_arr

```
array( [ [ 30.  ,  6.  , 10.5 , 12.  ,   3.  ,  18.  ],
         [ 3.75,  6.9,  12.  ,  5.4 ,  6.75, 10.35],
         [ 30.  , 15. ,  45.  , 75.  ,  60.  , 90.  ] ] )
```

Aggregation functions

Find max value:
arr.max()
30.0

Find min value:
arr.min()
1.0

Find average value:
data.mean()
8.141666666666667

Check the documentation for more examples

Sorting and concatenating

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

Sorting:

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

Concatenate:

```
new_arr = np.concatenate((a, b))
array( [4, 2, 1, 3, 5, 6, 7, 8] )
```

Always read the documentation!