Week 4 - Basic NumPy

Check the accompanying Jupyter notebook
for interactive examples!

Python is Slow

- Interpreted & dynamically typed
- GIL
- Suboptimal data structures (list)

NumPy is …

… a library for numerical computation

… sometimes faster

- “outsourcing” of slow Python loops to fast C code (*vectorization*)
- static typing
- arrays instead of lists
- in some cases multithreaded*
- used as a building block for other scientific libraries

… sometimes more elegant

- syntax extends even to libraries that don’t build on NumPy

** More on multithreading in NumPy:*
https://stackoverflow.com/a/16618280

Importing Modules

Not all of Python's functionality is available by default.

More specialized tools are organized in **modules** that need to be **imported** before they can be used

```
import time

current_time = time.localtime()

print(current_time.tm_year)      # 2021
```

You can also import **specific functions** from modules if you don't need the rest

```
from time import localtime

current_time = localtime()

print(current_time.tm_year)      # 2021
```

Check the Jupyter notebook for more import syntax!

Reminder: Installing NumPy

NumPy is a third-party module and thus needs to be installed manually!

1. Activate conda environment
2. `pip install numpy`
3. check if it worked by importing `numpy` in the interactive shell / a Jupyter notebook

The `np.ndarray`

Everything in NumPy is built around the `ndarray`.

Arrays, like lists, store values, but with two limitations:

1. fixed data type
2. fixed number of elements (or *shape*)

NumPy arrays can be created in different ways

```
import numpy as np

# from Python sequences

array1 = np.array([1, 2, 3])
array2 = np.array([1, 2, 3], dtype=float)

# using NumPy functions

array3 = np.zeros(3)     # [0., 0., 0.]
array4 = np.ones(3)      # [1., 1., 1.]
array5 = np.arange(3)    # [0, 1, 2]
```

...more ways in the Jupyter notebook!

Multidimensional Arrays

NumPy arrays are specifically suited to be **multidimensional** (i.e. nested arrays).

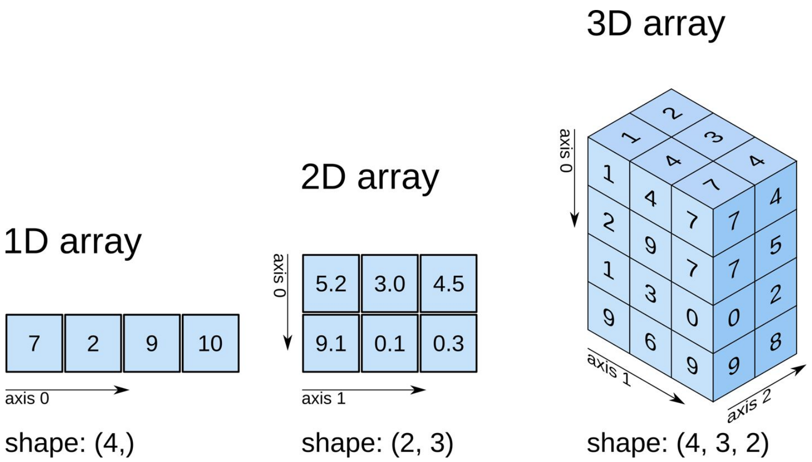


Image: https://predictivehacks.com/tips-about-numpy-arrays/

The **shape** is a tuple that specifies the **size of each dimension**

Some ways to create multidimensional arrays:

```
# from Python nested sequences

arr1 = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])

# many NumPy functions also take a shape
# argument

arr2 = np.zeros(shape=(3, 3))
```

Reshaping Arrays

Arrays can be **reshaped** into different shapes.

The number of dimensions may change, but the total number of elements may not.

One of the given dimension sizes may be **-1** and will be inferred automatically.

`array.flatten()` amounts to `array.reshape(-1)`

```
array = np.arange(9)
# [0, 1, 2, 3, 4, 5, 6, 7, 8]

array = array.reshape(3, 3)
# [[0, 1, 2],
#  [3, 4, 5],
#  [6, 7, 8]]

array = array.reshape(1, 9)
# [[0, 1, 2, 3, 4, 5, 6, 7, 8]]

array = array.reshape(3, -1)
# same as array.reshape(3, 3)

array = array.flatten()    # shape: (9)
```

Basic Indexing

NumPy provides convenient ways to index items in ndarrays:

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

# sub-array
sub_array = array[1]     # [4, 5, 6]

# single element
element = array[0, 2]    # 3

# don't do this:
element = array[0][2]
```

```
# slices work like they do in Python

slice = array[2, 0:2]    # [7, 8]
slice = array[1:, 2]     # [6, 9]
slice = array[2, ::-1]   # [9, 8, 7]

# slices are more general sub-arrays

slice = array[:, 1]      # [2, 5, 8]

# you can also get n-dim slices

slice = array[:-1, :-1]  # [[1, 2],
                         #  [4, 5]]
```

There are more ways of indexing, which will be covered in the next lecture!

Mathematical Operations

Thanks to the magic of dunder methods, NumPy arrays work with mathematical operators

element-wise operations: `+`, `-`, `*`, `/`, `**`

there is also **matrix-multiplication**: @

The work of looping through the array is handled by fast, precompiled C code!

```
# element-wise

arr1 = np.array([1, 2, 3, 0])
arr2 = np.array([2, 2, 0, 0])

result = arr1 + arr2      # [3, 4, 3, 0]
result = arr1 - arr2      # [-1, 0, 3, 0]
result = arr1 * arr2      # [2, 4, 0, 0]
result = arr1 / arr2
# [0.5, 1., inf, nan]

# matrix multiplication

mat1 = np.array([[1, 2], [3, 4]])
mat2 = np.array([[2, 2], [2, 2]])

result = mat1 @ mat2      # [[ 6,  6],
                          #  [14, 14]]
```

Broadcasting

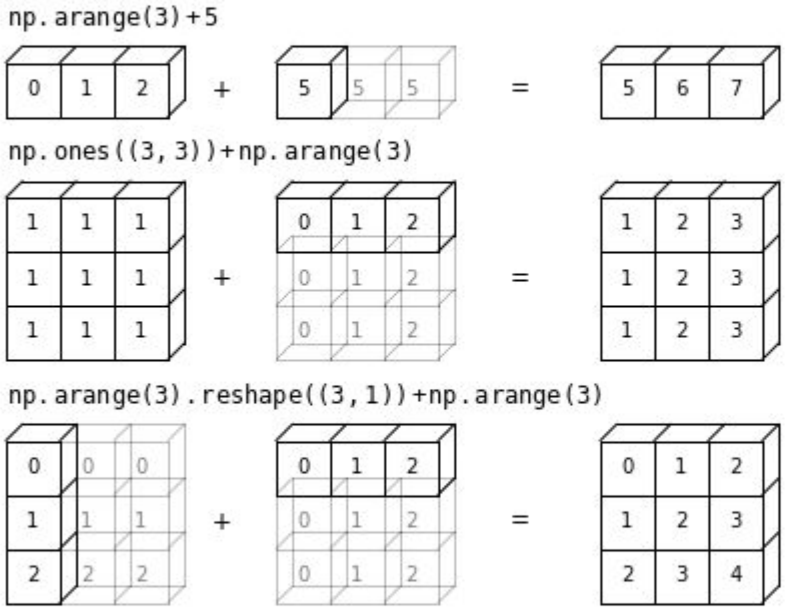
What happens if two operand arrays don't have the exact same shape?

1. Step If the arrays have different numbers of dimensions, the smaller shape is padded with ones on its left side.

2. Step If the number of the dimensions matches, but the size of a dimension does not, dimensions with the size of 1 are expanded.

3. Step If the shapes of the arrays still defer after applying the steps 1 and 2, a broadcasting error is raised.

More about broadcasting:
https://numpy.org/doc/stable/user/basics.broadcasting.html



Source:
https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html

Element-wise Functions

NumPy provides many functions that take entire arrays as inputs and perform element-wise operations.

A special type of element-wise functions are called **ufuncs** (universal functions). If given multiple arguments, they will perform automatic **broadcasting** where required.

A list of all available ufuncs can be found here:
https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs

A complete list of all NumPy functions can be found here:
https://numpy.org/doc/stable/reference/routines.html

Aggregation Functions

Other functions take one or more arrays as input and reduce it to a single number or an array of lower dimensionality.

- `np.sum`
- `np.min` / `np.max`
- `np.mean`
- [...]

They take a special argument which determines along which axis (or axes) to perform the operation.

If left out, it will be performed along all axes.

```
array = np.arange(9).reshape(3, 3)

# sum of each column
col_sum = np.sum(array, axis=0)

# sum of each row
row_sum = np.sum(array, axis=1)

# sum of entire array
arr_sum = np.sum(array, axis=(0, 1))

# or simply
arr_sum = np.sum(array)
```

Comparing Arrays

Comparison operators on NumPy arrays work **element-wise** and return a **boolean array**. Broadcasting is applied.

To check if two arrays are numerically identical, use `np.array_equal(array1, array2)`

`np.allclose(array1, array2)` checks if all values are equal within some error margin

`np.any()` and `np.all()` check if any/all values in an array are True(-ish)

```
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([0, 2, 3, 0])

equal = arr1 == arr2
# [False, True, True, False]

equal = arr1 == 4
# [False, False, False, True]

equal = np.array_equal(arr1, arr2)
# False
```

Random Numbers

NumPy provides a module for generating arrays of pseudo-random numbers with various distributions.

Check the notebook for examples!

https://numpy.org/doc/stable/reference/random/generator.html#simple-random-data