NumPy Cheatsheet

William Darko

December 2020

Contents

1 NumPy Basics

NumPy is stands for Numerical Python, and is a open source python Library containing multidimensional arrays and matrix data structures.

1.1 Installing NumPy library

If python is already installed, NumPy can be installed using:

```
conda install numpy
```

or

```
pip install numpy
```

1.2 Importing NumPy

Import NumPy into your python programme using:

```
import numpy as np
```

1.3 Arrays

Arrays are a central data-structure to the NumPy library. Some properties of arrays in the NumPy array are:

- **dtype**: type of the elements in the array (given all elements in the array are of the same type, dtype will be the same value)
- **rank**: number of dimensions of the array
- **shape**: a tuple of non-negative integers providing the size of the array along each dimensions
- **ndarray**: is the NumPy n-dimensional array and used to represent both matrices and vectors
- **vector**: a 1-dimensional array
- **matrix**: a 2-dimensional array
- **tensor**: often a 3-dimensional array
- **axes**: in NumPy, dimensions are referred to as axes

1.3.1 Creating Basic Arrays

To create NumPy arrays, we can use one of the following:

```
np.array(), np.zeros(), np.ones(), np.empty(), np.arange(),
np.linspace(), dtype
```

- **np.array()**: Create a NumPy array by simply passing in a python list as an argument

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
```

- **np.zeros()**: Create a NumPy array filled with zeros by passing in the length of the array as an argument

```
>>> np.zeros(2)
array([0., 0.])
```

- **np.ones()**: Create a NumPy array filled with ones by passing in the length of the array as an argument

```
>>> np.ones(3)
array([1., 1., 1.])
```

- **np.empty()**: Initialise a NumPy array populated with random values depending on the state of memory; the reason to use this over np.zeros, or np,ones, is speed.

```
>>> np.empty(4)
array([ 3.14, 42., 68 ])  # may vary
```

- **np.arange(int n)**: Create NumPy array populated with integers from 0, to n-1

```
>>> np.arange(4)
array([0, 1, 2, 3])
```

- **np.arange(first, last, step size)**: Create a NumPy array populated with numbers evenly spaced, starting at the first number, bounded by the last number

```
>>> np.arange(2, 9, 2)
array([2, 4, 6, 8])
```

- **np.linspace(first, last, num)**: Create a NumPy array with linearly spaced numbers starting from the first number, ending at the last number, of length num

```
>>> np.linspace(0, 10, num=5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

- **dtype** keyword: The default data type for NumPy arrays is **np.float64** We can explicitly specify the data type we'd like to work with:

```
>>> x = np.ones(2, dtype=np.int64)
>>> x
array([1, 1])
```

1.3.2 Manipulating and sorting arrays

The functions of focus: **np.sort()**, and **np.concatenate()**

- **np.sort()**: return a sorted copy of of same shape and type as an array by specifying the arguments:

  a: ***array-like*** - the array to sort

  axis: ***int or None, optional*** - The axis along which to sort; if *None* is passed, the array is flattened (made 1 dimensional) before sorting. Default is -1, which sorts along last axis.

  kind: ***(quicksort, mergesort, heapsort, stable) optional*** - choice of sorting algorithm to use. Default is *quicksort*.

  order: ***string or list of strings, optional*** - when the array is an array with fields defined, for instance:

```
>>> dtype = [('name', 'S10'), ('height', float), ('age', int)]
>>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38),
('Galahad', 1.7, 38)]
```

  the *order* argument specifies which fields to compare first, or in other words the order in which to compare fields during sorting.

- **ndarray.sort()**: Sort an array in place; similar to **np.sort()** except this sorts an array in place and takes all the same arguments except for an array

- **np.concatenate()**: Where a and b are initialised arrays

```
>>> np.concatenate(a,b)
```

combines the elements of both a, and b into a single array. More generally, np.concatenate() may take arguments of:

a1, a2, ...: sequence of arrays to concatenate which must have the same shape, except in the dimension corresponding to axis (fist axis by default)

axis: optional; a integer representing the axis along which the arrays will be concatenated. If *None* is passed, default of 0th axis is used, and arrays are flattened before being concatenated

out: optional; an ndarray as a destination to place the result. Must be of correct shape, matching what function will return.

dtype: str or dtype; if provided the destination array will have this data type

casting: one of the following (no, equiv, safe, same kind, unsafe)

1.3.3 Knowing shape and size of array

Using ndarray.ndim, ndarray.size, ndarray.shape

- **ndarray.ndim**: tells you the number of axes or dimensions of the array
- **ndarray.size**: tells you the total number of elemets of the array. This is the product of the elements of the arrays shape
- **ndarray.shape**: displays a tuple of integers that indicate the number of elements along each axis, or dimension of the array.

```
>>> array_example = np.array([[[0, 1, 2, 3],
...                            [4, 5, 6, 7]],
...
...                           [[0, 1, 2, 3],
...                            [4, 5, 6, 7]],
...
...                           [[0 ,1 ,2, 3],
...                            [4, 5, 6, 7]]])

>>> array_example.ndim
3
>>> array_example.size
24  // which is the product of 3 * 2 * 4
>>> array_example.shape
(3, 2, 4)
```