# Introduction

In this notebook we'll learn how to use NumPy to work with numerical data.

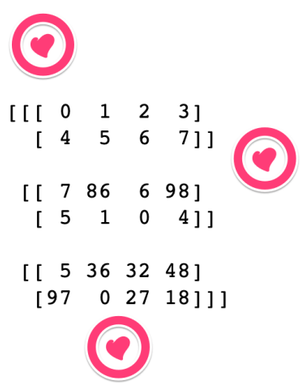

## Import Statements

```
import numpy as np

import matplotlib.pyplot as plt
from scipy import misc # contains an image of a racoon!
from PIL import Image # for reading image files
```

## Understanding NumPy's ndarray

NumPy's most amazing feature is the **powerful** ndarray.




### 1-Dimensional Arrays (Vectors)

```
my_array = np.array([1.1, 9.2, 8.4, 4.2])
```

The shape shows us that it has only one dimension

```
# shows rows and columns
my_array.shape
```

```
(4,)
```

```
# accesign elements like a list
my_array[2]
```

```
8.4
```

```
# shows dimensions of an array
my_array.ndim
```

```
1
```

### 2-Dimensional Arrays (Matrices)

```
array_2d = np.array([[1, 2, 3, 9],
                     [5, 6, 7, 8]])
```

```
print(f"array_2d has {array_2d.ndim} dimensions")
print(f"Its shape is {array_2d.shape}")
print(f"It has {array_2d.shape[0]} rows and {array_2d.shape[1]} columns")
print(array_2d)
```

```
array_2d has 2 dimensions
Its shape is (2, 4)
It has 2 rows and 4 columns
[[1 2 3 9]
 [5 6 7 8]]
```

```
# accesing single element
array_2d[1,2]
```

```
7
```

```
# accesing the whole row
array_2d[0,:]
```

```
array([1, 2, 3, 9])
```

### N-Dimensional Arrays (Tensors)

- How many dimensions does the array below have?
- What is its shape (i.e., how many elements are along each axis)?
- Try to access the value `18` in the last line of code.
- Try to retrieve a 1 dimensional vector with the values `[97, 0, 27, 18]`
- Try to retrieve a (3,2) matrix with the values `[[ 0, 4], [ 7, 5], [ 5, 97]]`

*Hint*: You can use the `:` operator just as with Python Lists.

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

                          [[7, 86, 6, 98],
                           [5, 1, 0, 4]],

                           [[5, 36, 32, 48],
                            [97, 0, 27, 18]]])

# Note all the square brackets!
```

```
# number of dimensions
mystery_array.ndim
```

```
3
```

```
# the shape of the matrix / tensor
mystery_array.shape
```

```
(3, 2, 4)
```

```
# accesing the value 18
mystery_array[2,1,3]
```

```
18
```

```
# retrieve a 1 dimensional vector with the values `[97, 0, 27, 18]`
mystery_array[2,1,:]
```

```
array([97,  0, 27, 18])
```

```
# retrieve a (3,2) matrix with the values `[[ 0,  4], [ 7,  5], [ 5, 97]]`
mystery_array[:,:,0]
```

```
array([[ 0,  4],
       [ 7,  5],
       [ 5, 97]])
```

```
mystery_array[:,:,0].shape
```

```
(3, 2)
```

## NumPy Mini-Challenges

Using `.arange()` to createa a vector `a` with values ranging from 10 to 29. You should get this:

`print(a)`

`[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29]`

```
a = np.arange(10,30)
print(a)
```

```
[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29]
```

Using Python slicing techniques on `a` to:

- Create an array containing only the last 3 values of `a`
- Create a subset with only the 4th, 5th, and 6th values
- Create a subset of `a` containing all the values except for the first 12 (i.e., `[22, 23, 24, 25, 26, 27, 28, 29]`)
- Create a subset that only contains the even numbers (i.e, every second number)

```
print(a[-3:])
print(a[3:6])
print(a[13:])
print(a[::2])
```

```
[27 28 29]
[13 14 15]
[23 24 25 26 27 28 29]
[10 12 14 16 18 20 22 24 26 28]
```

Reversing the order of the values in `a`, so that the first element comes last:

`[29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10]`

If you need a hint, you can check out this part of the NumPy beginner's guide

```
np.flip(a)
# a[::-1]
```

```
array([29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13,
       12, 11, 10])
```

Printing out all the indices of the non-zero elements in this array: [6,0,9,0,0,5,0]

```
b = np.array([6,0,9,0,0,5,0])
b_nozero_indicies = np.nonzero(b)
print(b_nozero_indicies)
```

```
(array([0, 2, 5], dtype=int64),)
```

Using NumPy to generate a 3x3x3 array with random numbers

Hint: Use the `.random()` function

```
z = np.random.random((3,3,3))
print(z)
```

```
[[[0.81653655 0.00431033 0.03870669]
  [0.4955495  0.73472645 0.34327774]
  [0.3126213  0.83261482 0.71309667]]

 [[0.60650752 0.61642721 0.71795893]
  [0.48211282 0.09405044 0.64937934]
  [0.30946544 0.72003725 0.75114457]]

 [[0.50897045 0.05038408 0.9993145 ]
  [0.51498646 0.87179288 0.59454365]
  [0.62430254 0.10839642 0.99908368]]]
```

Using `.linspace()` to create a vector `x` of size 9 with values spaced out evenly between 0 to 100 (both included).

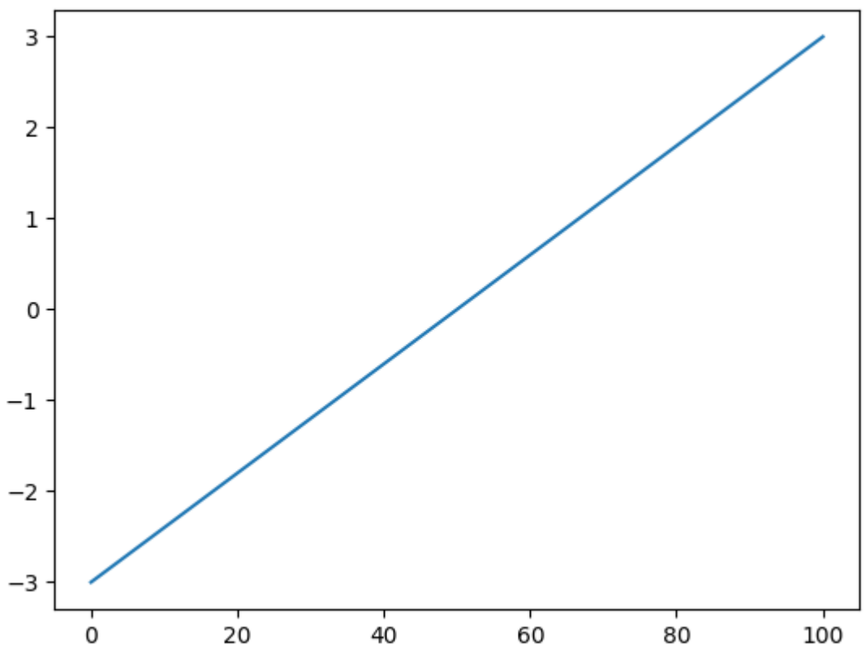
```
x = np.linspace(0,100, num=9)
print(x)
x.shape
```

```
[  0.   12.5  25.   37.5  50.   62.5  75.   87.5 100. ]
(9,)
```

Using `.linspace()` to create another vector `y` of size 9 with values between -3 to 3 (both included). Then plot `x` and `y` on a line chart using Matplotlib.

```
y = np.linspace(start=-3,stop=3,num=9)
print(y)
plt.plot(x,y)
plt.show()
```

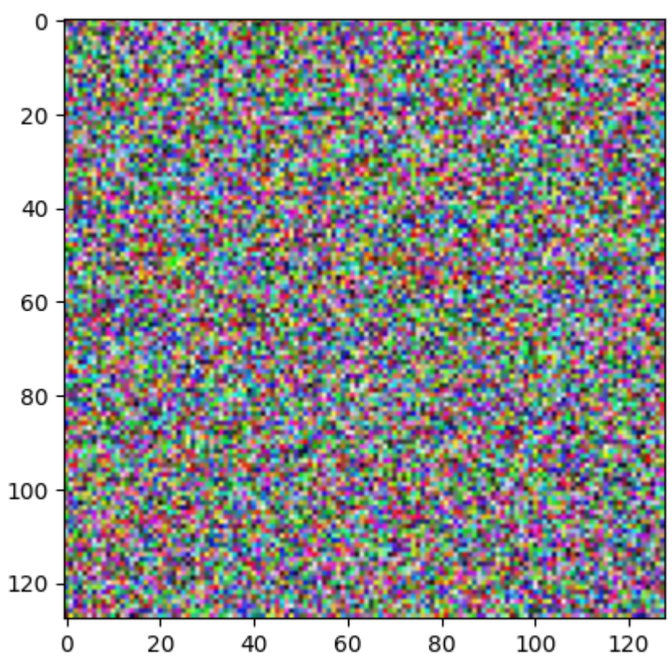
```
[-3.   -2.25 -1.5  -0.75  0.    0.75  1.5   2.25  3.  ]
```




Using NumPy to generate an array called `noise` with shape 128x128x3 that has random values. Then use Matplotlib's `.imshow()` to display the array as an image.

```
noise = np.random.random((128,128,3))
print(noise.shape)
plt.imshow(noise)
```

```
(128, 128, 3)
<matplotlib.image.AxesImage at 0x285d77d24d0>
```




## Linear Algebra with Vectors

```
v1 = np.array([4, 5, 2, 7])
v2 = np.array([2, 1, 3, 3])
v1 + v2
```

```
array([ 6,  6,  5, 10])
```

```
# Python Lists vs ndarrays
list1 = [4, 5, 2, 7]
list2 = [2, 1, 3, 3]
list1 + list2
```

```
[4, 5, 2, 7, 2, 1, 3, 3]
```

```
v1*v2
```

```
array([ 8,  5,  6, 21])
```

## Broadcasting and Scalars

```
array_2d = np.array([[1, 2, 3, 4],
                     [5, 6, 7, 8]])
```

```
# NumPy will make the shape of the smaller array - our scalar -
# compatible with the larger array
# which is called brodcasting
array_2d + 10
```

```
array([[11, 12, 13, 14],
       [15, 16, 17, 18]])
```

```
array_2d * 10
```

```
array([[10, 20, 30, 40],
       [50, 60, 70, 80]])
```

## Matrix Multiplication with @ and .matmul()

**Wikipedia's Example**

$$\underset{4\times 2 \text{ matrix}}{\begin{bmatrix} a_{11} & a_{12} \\ \cdot & \cdot \\ a_{31} & a_{32} \\ \cdot & \cdot \end{bmatrix}} \underset{2\times 3 \text{ matrix}}{\begin{bmatrix} \cdot & b_{12} & b_{13} \\ \cdot & b_{22} & b_{23} \end{bmatrix}} = \underset{4\times 3 \text{ matrix}}{\begin{bmatrix} \cdot & c_{12} & c_{13} \\ \cdot & \cdot & \cdot \\ \cdot & c_{32} & c_{33} \\ \cdot & \cdot & \cdot \end{bmatrix}}$$

$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$c_{33} = a_{31}b_{13} + a_{32}b_{23}$$

```
a1 = np.array([[1, 3],
               [0, 1],
               [6, 2],
               [9, 7]])

b1 = np.array([[4, 1, 3],
               [5, 8, 5]])

print(f'{a1.shape}: a has {a1.shape[0]} rows and {a1.shape[1]} columns.')
print(f'{b1.shape}: b has {b1.shape[0]} rows and {b1.shape[1]} columns.')
print('Dimensions of result: (4x2)*(2x3)=(4x3)')
```

```
(4, 2): a has 4 rows and 2 columns.
(2, 3): b has 2 rows and 3 columns.
Dimensions of result: (4x2)*(2x3)=(4x3)
```

Let's multiply `a1` with `b1`. Then use the `.matmul()` function or the `@` operator to check your work.

```
# c1 = a1 @ b1
c1 = np.matmul(a1,b1)
print(c1.shape)
print(c1)
```

```
(4, 3)
[[19 25 18]
 [ 5  8  5]
 [34 22 28]
 [71 65 62]]
```

## Manipulating Images as ndarrays

```
img_racoon = misc.face()
plt.imshow(img_racoon)
```

```
<matplotlib.image.AxesImage at 0x285d782a210>
```


Checking what is the data type of `img`, and, what is the shape of `img` and how many dimensions does it have. And What is the resolution of the image.

```
print(type(img_racoon))
```

```
<class 'numpy.ndarray'>
```

```
# resolution of the image is 768 on 1024
img_racoon.shape
```

```
(768, 1024, 3)
```

```
img_racoon.ndim
```

```
3
```

Converting the image to black and white. The values in our `img` range from 0 to 255.

- Divide all the values by 255 to convert them to sRGB, where all the values are between 0 and 1.
- Next, multiply the sRGB array by the `grey_vals` to convert the image to grey scale.
- Finally use Matplotlib's `.imshow()` together with the colormap parameter set to gray `cmap=gray` to look at the results.

```
grey_vals = np.array([0.2126, 0.7152, 0.0722])
```

```
img_racoon_sRGB = img_racoon / 255
# img_racoon_gery_scale = np.matmul(img_racoon_sRGB, grey_vals)
# same as above
img_racoon_gery_scale = img_racoon_sRGB @ grey_vals
```

```
plt.imshow(img_racoon_gery_scale, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x285dc208f10>
```


Manipulating the images by doing some operations on the underlying ndarrays.

flipping the grayscale image upside down

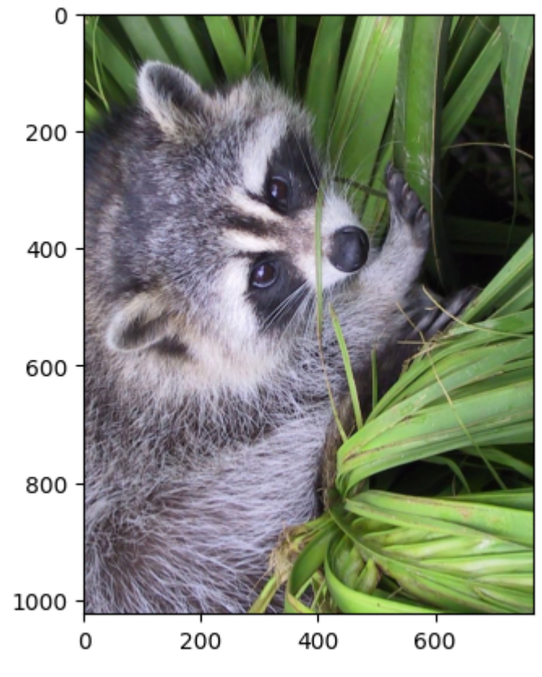
```
plt.imshow(np.flip(img_racoon_gery_scale), cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x285d77e41d0>
```


Rotating the colour image

```
plt.imshow(np.rot90(img_racoon))
```

```
<matplotlib.image.AxesImage at 0x285dd043990>
```




Inverting (i.e., solarize) the colour image. To do this we need to convert all the pixels to their "opposite" value, so black (0) becomes white (255).

```
solar_img = 255 - img_racoon
plt.imshow(solar_img)
```

```
<matplotlib.image.AxesImage at 0x285dc0c6350>
```


## Use your Own Image!

### Use PIL to open

```
file_name = 'data/macarons.jpg'
my_img = Image.open(file_name)
my_img_array = np.array(my_img)
```

```
my_img_array.ndim
```

```
3
```

```
my_img_array.shape
```

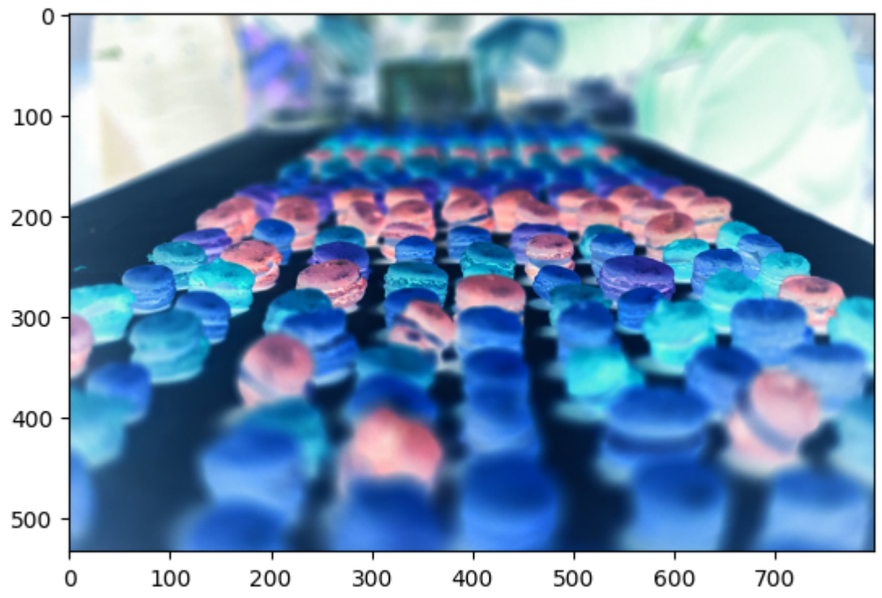
```
(533, 799, 3)
```

```
plt.imshow(my_img_array)
```

```
<matplotlib.image.AxesImage at 0x285dd089990>
```


```
plt.imshow(255-my_img_array)
```

```
<matplotlib.image.AxesImage at 0x285dd0e85d0>
```



Colab paid products - Cancel contracts here