NumPy

NumPy (or Numpy) is a Linear Algebra Library for Python, the reason it is so important for Data Science with Python is that almost all of the libraries in the PyData Ecosystem rely on NumPy as one of their main building blocks.

Numpy is also incredibly fast, as it has bindings to C libraries. For more info on why you would want to use Arrays instead of lists, check out this great StackOverflow post.

We will only learn the basics of NumPy, to get started we need to install it!

Installation Instructions

It is highly recommended you install Python using the Anaconda distribution to make sure all underlying dependencies (such as Linear Algebra libraries) all sync up with the use of a conda install. If you have Anaconda, install NumPy by going to your terminal or command prompt and typing:

```
conda install numpy
```

If you do not have Anaconda and can not install it, please refer to Numpy's official documentation on various installation instructions.

Using NumPy

Once you've installed NumPy you can import it as a library:

In [1]:
```
import numpy as np
```

Numpy has many built-in functions and capabilities. We won't cover them all but instead we will focus on some of the most important aspects of Numpy: vectors,arrays,matrices, and number generation. Let's start by discussing arrays.

Numpy Arrays

NumPy arrays are the main way we will use Numpy throughout the course. Numpy arrays essentially come in two flavors: vectors and matrices. Vectors are strictly 1-d arrays and matrices are 2-d (but you should note a matrix can still have only one row or one column).

Let's begin our introduction by exploring how to create NumPy arrays.

Creating NumPy Arrays

From a Python List

We can create an array by directly converting a list or list of lists:

In [2]:
```
my_list = [1,2,3]
my_list
```

Out[2]:
```
[1, 2, 3]
```

In [3]:
```
np.array(my_list)
```

Out[3]:
```
array([1, 2, 3])
```

In [4]:
```
my_matrix = [[1,2,3],[4,5,6],[7,8,9]]
my_matrix
```

Out[4]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [5]:
```
np.array(my_matrix)
```

Out[5]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Built-in Methods

There are lots of built-in ways to generate Arrays

arange

Return evenly spaced values within a given interval.

In [6]:
```
np.arange(0,10)
```

Out[6]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [7]:
```
np.arange(0,11,2)
```

Out[7]:
```
array([ 0,  2,  4,  6,  8, 10])
```

zeros and ones

Generate arrays of zeros or ones

In [8]:
```
np.zeros(3)
```

Out[8]:
```
array([0., 0., 0.])
```

In [9]:
```
np.zeros((5,5))
```

Out[9]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

In [10]:
```
np.ones(3)
```

Out[10]:
```
array([1., 1., 1.])
```

In [11]:
```
np.ones((3,3))
```

Out[11]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

linspace

Return evenly spaced numbers over a specified interval.

In [12]:
```
np.linspace(0,10,3)
```

Out[12]:
```
array([ 0.,  5., 10.])
```

In [13]:
```
np.linspace(0,10,50)
```

Out[13]:
```
array([ 0.        ,  0.20408163,  0.40816327,  0.6122449 ,  0.81632653,
        1.02040816,  1.2244898 ,  1.42857143,  1.63265306,  1.83673469,
        2.04081633,  2.24489796,  2.44897959,  2.65306122,  2.85714286,
        3.06122449,  3.26530612,  3.46938776,  3.67346939,  3.87755102,
        4.08163265,  4.28571429,  4.48979592,  4.69387755,  4.89795918,
        5.10204082,  5.30612245,  5.51020408,  5.71428571,  5.91836735,
        6.12244898,  6.32653061,  6.53061224,  6.73469388,  6.93877551,
        7.14285714,  7.34693878,  7.55102041,  7.75510204,  7.95918367,
        8.16326531,  8.36734694,  8.57142857,  8.7755102 ,  8.97959184,
        9.18367347,  9.3877551 ,  9.59183673,  9.79591837, 10.        ])
```

eye

Creates an identity matrix

In [14]:
```
np.eye(4)
```

Out[14]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

Random

Numpy also has lots of ways to create random number arrays:

rand

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

In [15]:
```
np.random.rand(2)
```

Out[15]:
```
array([0.41703316, 0.23015319])
```

In [16]:
```
np.random.rand(5,5)
```

Out[16]:
```
array([[0.944401  , 0.43269675, 0.76506694, 0.86290281, 0.46198623],
       [0.93574928, 0.06125027, 0.65208648, 0.52099736, 0.48209749],
       [0.99060435, 0.20807724, 0.8986957 , 0.49952628, 0.95325773],
       [0.84222212, 0.65062228, 0.93486537, 0.36647365, 0.46241027],
       [0.88234517, 0.03213418, 0.84524108, 0.95110549, 0.22896302]])
```

randn

Return a sample (or samples) from the "standard normal" distribution. Unlike rand which is uniform:

In [17]:
```
np.random.randn(2)
```

Out[17]:
```
array([-0.12072836, -1.08787606])
```

In [18]:
```
np.random.randn(5,5)
```

Out[18]:
```
array([[ 0.44231303,  0.03709603, -2.29347366, -2.84076473, -1.34917457],
       [-0.22527599, -2.5676545 ,  0.58403109, -0.01472647,  0.09591978],
       [ 0.48574173, -0.35242961,  2.46995345,  0.74812217, -0.30482086],
       [-1.82090993, -0.08618301,  0.4164998 , -0.27828388,  0.21044519],
       [ 1.40321755,  0.3557016 , -0.66915756, -0.57427655, -0.11895588]])
```

randint

Return random integers from `low` (inclusive) to `high` (exclusive).

In [19]:
```
np.random.randint(1,100)
```

Out[19]:
```
11
```

In [20]:
```
np.random.randint(1,100,10)
```

Out[20]:
```
array([74, 32,  8, 56, 75, 16,  5, 71, 76, 51])
```

Array Attributes and Methods

Let's discuss some useful attributes and methods or an array:

In [21]:
```
arr = np.arange(25)
ranarr = np.random.randint(0,50,10)
```

In [22]:
```
arr
```

Out[22]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

In [23]:
```
ranarr
```

Out[23]:
```
array([ 2, 27, 44, 47, 12, 41,  5,  3, 34, 31])
```

Reshape

Returns an array containing the same data with a new shape.

In [24]:
```
arr.reshape(5,5)
```

Out[24]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24]])
```

max,min,argmax,argmin

These are useful methods for finding max or min values. Or to find their index locations using argmin or argmax

In [25]:
```
ranarr
```

Out[25]:
```
array([ 2, 27, 44, 47, 12, 41,  5,  3, 34, 31])
```

In [26]:
```
ranarr.max()
```

Out[26]:
```
47
```

In [27]:
```
ranarr.argmax()
```

Out[27]:
```
3
```

In [28]:
```
ranarr.min()
```

Out[28]:
```
2
```

In [29]:
```
ranarr.argmin()
```

Out[29]:
```
0
```

Shape

Shape is an attribute that arrays have (not a method):

In [30]:
```
# Vector
arr.shape
```

Out[30]:
```
(25,)
```

In [31]:
```
# Notice the two sets of brackets
arr.reshape(1,25)
```

Out[31]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
        16, 17, 18, 19, 20, 21, 22, 23, 24]])
```

In [32]:
```
arr.reshape(1,25).shape
```

Out[32]:
```
(1, 25)
```

In [33]:
```
arr.reshape(25,1)
```

Out[33]:
```
array([[ 0],
       [ 1],
       [ 2],
       [ 3],
       [ 4],
       [ 5],
       [ 6],
       [ 7],
       [ 8],
       [ 9],
       [10],
       [11],
       [12],
       [13],
       [14],
       [15],
       [16],
       [17],
       [18],
       [19],
       [20],
       [21],
       [22],
       [23],
       [24]])
```

In [34]:
```
arr.reshape(25,1).shape
```

Out[34]:
```
(25, 1)
```

dtype

You can also grab the data type of the object in the array:

In [35]:
```
arr.dtype
```

Out[35]:
```
dtype('int32')
```

Great Job!

NumPy Indexing and Selection

In this lecture we will discuss how to select elements or groups of elements from an array.

In [1]:
```
import numpy as np
```

In [2]:
```
#Creating sample array
arr = np.arange(0,11)
```

In [3]:
```
#Show
arr
```

Out[3]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Bracket Indexing and Selection

The simplest way to pick one or some elements of an array looks very similar to python lists:

In [4]:
```
#Get a value at an index
arr[8]
```

Out[4]:
```
8
```

In [5]:
```
#Get values in a range
arr[1:5]
```

Out[5]:
```
array([1, 2, 3, 4])
```

In [6]:
```
#Get values in a range
arr[0:5]
```

Out[6]:
```
array([0, 1, 2, 3, 4])
```

Broadcasting

Numpy arrays differ from a normal Python list because of their ability to broadcast:

In [7]:
```
#Setting a value with index range (Broadcasting)
arr[0:5]=100

#Show
arr
```

Out[7]:
```
array([100, 100, 100, 100, 100,   5,   6,   7,   8,   9,  10])
```

In [8]:
```
# Reset array, we'll see why I had to reset in  a moment
arr = np.arange(0,11)

#Show
arr
```

Out[8]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [9]:
```
#Important notes on Slices
slice_of_arr = arr[0:6]

#Show slice
slice_of_arr
```

Out[9]:
```
array([0, 1, 2, 3, 4, 5])
```

In [10]:
```
#Change Slice
slice_of_arr[:]=99

#Show Slice again
slice_of_arr
```

Out[10]:
```
array([99, 99, 99, 99, 99, 99])
```

Now note the changes also occur in our original array!

In [11]:
```
arr
```

Out[11]:
```
array([99, 99, 99, 99, 99, 99,  6,  7,  8,  9, 10])
```

Data is not copied, it's a view of the original array! This avoids memory problems!

In [12]:
```
#To get a copy, need to be explicit
arr_copy = arr.copy()

arr_copy
```

Out[12]:
```
array([99, 99, 99, 99, 99, 99,  6,  7,  8,  9, 10])
```

Indexing a 2D array (matrices)

The general format is **arr_2d[row][col]** or **arr_2d[row,col]**. I recommend usually using the comma notation for clarity.

In [13]:
```
arr_2d = np.array(([5,10,15],[20,25,30],[35,40,45]))

#Show
arr_2d
```

Out[13]:
```
array([[ 5, 10, 15],
       [20, 25, 30],
       [35, 40, 45]])
```

In [14]:
```
#Indexing row
arr_2d[1]
```

Out[14]:
```
array([20, 25, 30])
```

In [15]:
```
# Format is arr_2d[row][col] or arr_2d[row,col]

# Getting individual element value
arr_2d[1][0]
```

Out[15]:
```
20
```

In [16]:
```
# Getting individual element value
arr_2d[1,0]
```

Out[16]:
```
20
```

In [17]:
```
# 2D array slicing

#Shape (2,2) from top right corner
arr_2d[:2,1:]
```

Out[17]:
```
array([[10, 15],
       [25, 30]])
```

In [18]:
```
#Shape bottom row
arr_2d[2]
```

Out[18]:
```
array([35, 40, 45])
```

In [19]:
```
#Shape bottom row
arr_2d[2,:]
```

Out[19]:
```
array([35, 40, 45])
```

Fancy Indexing

Fancy indexing allows you to select entire rows or columns out of order,to show this, let's quickly build out a numpy array:

In [20]:
```
#Set up matrix
arr2d = np.zeros((10,10))
```

In [21]:
```
#Length of array
arr_length = arr2d.shape[1]
```

In [22]:
```
#Set up array

for i in range(arr_length):
    arr2d[i] = i

arr2d
```

Out[22]:
```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [3., 3., 3., 3., 3., 3., 3., 3., 3., 3.],
       [4., 4., 4., 4., 4., 4., 4., 4., 4., 4.],
       [5., 5., 5., 5., 5., 5., 5., 5., 5., 5.],
       [6., 6., 6., 6., 6., 6., 6., 6., 6., 6.],
       [7., 7., 7., 7., 7., 7., 7., 7., 7., 7.],
       [8., 8., 8., 8., 8., 8., 8., 8., 8., 8.],
       [9., 9., 9., 9., 9., 9., 9., 9., 9., 9.]])
```

Fancy indexing allows the following

In [23]:
```
arr2d[[2,4,6,8]]
```

Out[23]:
```
array([[2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [4., 4., 4., 4., 4., 4., 4., 4., 4., 4.],
       [6., 6., 6., 6., 6., 6., 6., 6., 6., 6.],
       [8., 8., 8., 8., 8., 8., 8., 8., 8., 8.]])
```

In [24]:
```
#Allows in any order
arr2d[[6,4,2,7]]
```

Out[24]:
```
array([[6., 6., 6., 6., 6., 6., 6., 6., 6., 6.],
       [4., 4., 4., 4., 4., 4., 4., 4., 4., 4.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [7., 7., 7., 7., 7., 7., 7., 7., 7., 7.]])
```

More Indexing Help

Indexing a 2d matrix can be a bit confusing at first, especially when you start to add in step size. Try google image searching NumPy indexing to fins useful images, like this one:

Selection

Let's briefly go over how to use brackets for selection based off of comparison operators.

In [25]:
```
arr = np.arange(1,11)
arr
```

Out[25]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [26]:
```
arr > 4
```

Out[26]:
```
array([False, False, False, False,  True,  True,  True,  True,  True,
        True])
```

In [27]:
```
bool_arr = arr>4
```

In [28]:
```
bool_arr
```

Out[28]:
```
array([False, False, False, False,  True,  True,  True,  True,  True,
        True])
```

In [29]:
```
arr[bool_arr]
```

Out[29]:
```
array([ 5,  6,  7,  8,  9, 10])
```

In [30]:
```
arr[arr>2]
```

Out[30]:
```
array([ 3,  4,  5,  6,  7,  8,  9, 10])
```

In [31]:
```
x = 2
arr[arr>x]
```

Out[31]:
```
array([ 3,  4,  5,  6,  7,  8,  9, 10])
```

Great Job!

NumPy Operations

Arithmetic

You can easily perform array with array arithmetic, or scalar with array arithmetic. Let's see some examples:

In [1]:
```python
import numpy as np
arr = np.arange(0,10)
```

In [2]:
```python
arr + arr
```

Out[2]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [3]:
```python
arr * arr
```

Out[3]:
```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

In [4]:
```python
arr - arr
```

Out[4]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [5]:
```python
# Warning on division by zero, but not an error!
# Just replaced with nan
arr/arr
```

```
<ipython-input-5-2f119c028196>:3: RuntimeWarning: invalid value encountered in true_divide
  arr/arr
```

Out[5]:
```
array([nan,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

In [6]:
```python
# Also warning, but not an error instead infinity
1/arr
```

```
<ipython-input-6-c81c9b6169c9>:2: RuntimeWarning: divide by zero encountered in true_divide
  1/arr
```

Out[6]:
```
array([       inf, 1.        , 0.5       , 0.33333333, 0.25      ,
       0.2       , 0.16666667, 0.14285714, 0.125     , 0.11111111])
```

In [7]:
```python
arr**3
```

Out[7]:
```
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729], dtype=int32)
```

Universal Array Functions

Numpy comes with many universal array functions, which are essentially just mathematical operations you can use to perform the operation across the array. Let's show some common ones:

In [8]:
```python
#Taking Square Roots
np.sqrt(arr)
```

Out[8]:
```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])
```

In [9]:
```python
#Calcualting exponential (e^)
np.exp(arr)
```

Out[9]:
```
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03])
```

In [10]:
```python
np.max(arr) #same as arr.max()
```

Out[10]:
```
9
```

In [11]:
```python
np.sin(arr)
```

Out[11]:
```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

In [12]:
```python
np.log(arr)
```

```
<ipython-input-12-a67b4ae04e95>:1: RuntimeWarning: divide by zero encountered in log
  np.log(arr)
```

Out[12]:
```
array([      -inf, 0.        , 0.69314718, 1.09861229, 1.38629436,
       1.60943791, 1.79175947, 1.94591015, 2.07944154, 2.19722458])
```

Great Job!

That's all we need to know for now!

NumPy Exercises - Solutions

Now that we've learned about NumPy let's test your knowledge. We'll start off with a few simple tasks and then you'll be asked some more complicated questions.

Import NumPy as np

In [1]:
```
import numpy as np
```

Create an array of 10 zeros

In [2]:
```
np.zeros(10)
```
Out[2]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Create an array of 10 ones

In [3]:
```
np.ones(10)
```
Out[3]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Create an array of 10 fives

In [4]:
```
np.ones(10) * 5
```
Out[4]:
```
array([5., 5., 5., 5., 5., 5., 5., 5., 5., 5.])
```

Create an array of the integers from 10 to 50

In [5]:
```
np.arange(10,51)
```
Out[5]:
```
array([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])
```

Create an array of all the even integers from 10 to 50

In [6]:
```
np.arange(10,51,2)
```
Out[6]:
```
array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
       44, 46, 48, 50])
```

Create a 3x3 matrix with values ranging from 0 to 8

In [7]:
```
np.arange(9).reshape(3,3)
```
Out[7]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Create a 3x3 identity matrix

In [8]:
```
np.eye(3)
```
Out[8]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Use NumPy to generate a random number between 0 and 1

In [9]:
```
np.random.rand(1)
```
Out[9]:
```
array([0.12771286])
```

Use NumPy to generate an array of 25 random numbers sampled from a standard normal distribution

In [10]:
```
np.random.randn(25)
```
Out[10]:
```
array([ 1.22972068,  1.22536641, -0.53446356, -1.90021679, -1.55455224,
       -2.72950162, -2.68256111,  1.61182643,  0.75374281,  1.26196692,
        0.98206952,  0.35046025, -0.72146298,  0.60142452, -1.06659483,
       -0.27106718,  0.41578358, -1.07121204, -1.30055237, -1.04576771,
        0.8512641 ,  0.75419044, -0.1231919 , -0.57852013,  1.72719382])
```

Create the following matrix:

In [11]:
```
np.arange(1,101).reshape(10,10) / 100
```
Out[11]:
```
array([[0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1 ],
       [0.11, 0.12, 0.13, 0.14, 0.15, 0.16, 0.17, 0.18, 0.19, 0.2 ],
       [0.21, 0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.3 ],
       [0.31, 0.32, 0.33, 0.34, 0.35, 0.36, 0.37, 0.38, 0.39, 0.4 ],
       [0.41, 0.42, 0.43, 0.44, 0.45, 0.46, 0.47, 0.48, 0.49, 0.5 ],
       [0.51, 0.52, 0.53, 0.54, 0.55, 0.56, 0.57, 0.58, 0.59, 0.6 ],
       [0.61, 0.62, 0.63, 0.64, 0.65, 0.66, 0.67, 0.68, 0.69, 0.7 ],
       [0.71, 0.72, 0.73, 0.74, 0.75, 0.76, 0.77, 0.78, 0.79, 0.8 ],
       [0.81, 0.82, 0.83, 0.84, 0.85, 0.86, 0.87, 0.88, 0.89, 0.9 ],
       [0.91, 0.92, 0.93, 0.94, 0.95, 0.96, 0.97, 0.98, 0.99, 1.  ]])
```

Create an array of 20 linearly spaced points between 0 and 1:

In [12]:
```
np.linspace(0,1,20)
```
Out[12]:
```
array([0.        , 0.05263158, 0.10526316, 0.15789474, 0.21052632,
       0.26315789, 0.31578947, 0.36842105, 0.42105263, 0.47368421,
       0.52631579, 0.57894737, 0.63157895, 0.68421053, 0.73684211,
       0.78947368, 0.84210526, 0.89473684, 0.94736842, 1.        ])
```

Numpy Indexing and Selection

Now you will be given a few matrices, and be asked to replicate the resulting matrix outputs:

In [13]:
```
mat = np.arange(1,26).reshape(5,5)
mat
```
Out[13]:
```
array([[ 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]])
```

In [14]:
```
# WRITE CODE HERE THAT REPRODUCES THE OUTPUT OF THE CELL BELOW
# BE CAREFUL NOT TO RUN THE CELL BELOW, OTHERWISE YOU WON'T
# BE ABLE TO SEE THE OUTPUT ANY MORE
```

In [15]:
```
mat[2:,1:]
```
Out[15]:
```
array([[12, 13, 14, 15],
       [17, 18, 19, 20],
       [22, 23, 24, 25]])
```

In [16]:
```
# WRITE CODE HERE THAT REPRODUCES THE OUTPUT OF THE CELL BELOW
# BE CAREFUL NOT TO RUN THE CELL BELOW, OTHERWISE YOU WON'T
# BE ABLE TO SEE THE OUTPUT ANY MORE
```

In [17]:
```
mat[3,4]
```
Out[17]:
```
20
```

In [18]:
```
# WRITE CODE HERE THAT REPRODUCES THE OUTPUT OF THE CELL BELOW
# BE CAREFUL NOT TO RUN THE CELL BELOW, OTHERWISE YOU WON'T
# BE ABLE TO SEE THE OUTPUT ANY MORE
```

In [19]:
```
mat[:3,1:2]
```
Out[19]:
```
array([[ 2],
       [ 7],
       [12]])
```

In [20]:
```
# WRITE CODE HERE THAT REPRODUCES THE OUTPUT OF THE CELL BELOW
# BE CAREFUL NOT TO RUN THE CELL BELOW, OTHERWISE YOU WON'T
# BE ABLE TO SEE THE OUTPUT ANY MORE
```

In [21]:
```
mat[4,:]
```
Out[21]:
```
array([21, 22, 23, 24, 25])
```

In [22]:
```
# WRITE CODE HERE THAT REPRODUCES THE OUTPUT OF THE CELL BELOW
# BE CAREFUL NOT TO RUN THE CELL BELOW, OTHERWISE YOU WON'T
# BE ABLE TO SEE THE OUTPUT ANY MORE
```

In [23]:
```
mat[3:5,:]
```
Out[23]:
```
array([[16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25]])
```

Now do the following

Get the sum of all the values in mat

In [24]:
```
mat.sum()
```
Out[24]:
```
325
```

Get the standard deviation of the values in mat

In [25]:
```
mat.std()
```
Out[25]:
```
7.211102550927978
```

Get the sum of all the columns in mat

In [26]:
```
mat.sum(axis=0)
```
Out[26]:
```
array([55, 60, 65, 70, 75])
```

Great Job!