What is Numpy?

- Numpy, or Numerical Python, is a package for performing mathematical and statistical operations **quickly**
- Written in C and FORTRAN, which is more efficient than python and scales better as data size increases.
- Python is slow and not suitable for large amounts of data
- Pandas uses numpy
- *Note that we use alias np. This simply makes referencing them in the code easier, and is best practice.*

The first thing we want to do is import numpy.

In [2]:

```
import numpy as np
```

ndarray

The workhorse of Numpy is the `ndarray` object, which is short for *n-dimensional array*. These objects are:

- Much quicker to operate on than Python collections
- Have a wealth of useful operations ready to be called
- Share many format properties with Mathematical arrays, tensors etc.

Let us first define a Python list containing the ages of 6 people.

In [4]:

```
ages_list = [10, 5, 8, 32, 65, 43]
print(ages_list)
```

```
[10, 5, 8, 32, 65, 43]
```

There are 3 main ways to instantiate a Numpy ndarray object. One of these is to use `np.array(<collection>)`

In [3]:

```
ages = np.array(ages_list)
print(type(ages))
print(ages)
```

```
<class 'numpy.ndarray'>
[10  5  8 32 65 43]
```

The ndarray we have just created has a number of immutable properties:

- Size
- Shape

We can get information about these properties by calling `.size` and `.shape`.

Note: these are not method calls, these are calls to get attributes of the ndarray object of the array

In [8]:

```
print(ages)
print("Size:\t" , ages.size)
print("Shape:\t", ages.shape)
```

```
[10  5  8 32 65 43]
Size:    6
Shape:   (6,)
```

What difference do you see between these two properties?

The other 2 main ways to instantiate an array are by using `.zeros()` and `.empty()`. Zeros will produce an array of, you guessed it, zeros, which will be of the dimensions you specify.

In [8]:

```
zeroArr = np.zeros(5)
print(zeroArr)
```

```
[0. 0. 0. 0. 0.]
```

Whereas empty will produce an array of the dimensions you specify, but containing random, essentially irrelevant values.

Multi-dim

Now let us define a new list containing the weights of these 6 people.

In [6]:

```
weight_list = [32, 18, 26, 60, 55, 65]
```

Now, we define an ndarray containing all fo this information, and again print the size and shape of the array.

In [7]:

```
people = np.array([ages_list, weight_list])

a = np.array([[10, 5, 8, 32, 65, 43],
              [32, 18, 26, 60, 55, 65]
             ])

print("People:\t" , people)
print("Size:\t" , people.size)
print("Shape:\t", people.shape)
```

```
People:  [[10  5  8 32 65 43]
 [32 18 26 60 55 65]]
Size:    12
Shape:   (2, 6)
```

What do you notice to be different between this array and the previous one?

What happens if we try to reassign the shape attribute of the array?

In [14]:

```
people = people.reshape(12,1)
print("People:\t" , people)
print("Size:\t" , people.size)
print("Shape:\t", people.shape)
```

```
People:  [[10]
 [ 5]
 [ 8]
 [32]
 [65]
 [43]
 [32]
 [18]
 [26]
 [60]
 [55]
 [65]]
Size:    12
Shape:   (12, 1)
```

Note: The new shape must be the same "size" as the old shape

Exercise

- Generate a 1D numpy array with the values [7, 9, 65, 33, 85, 99]
- Generate a matrix (2D numpy array) of the values:

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 4 \\ 2 & 3 & 0 \\ 0 & 5 & 1 \end{pmatrix}$$

- Change the dimensions of this array to another permitted shape

Array Generation

Instead of defining an array manually, we can ask numpy to do it for us.

The `np.arange()` method creates a range of numbers with user defined steps between each.

In [13]:

```
five_times_table = np.arange(0, 55, 5)
five_times_table
```

Out[13]:

```
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45, 50])
```

The `np.linspace()` method will produce a range of evenly spaced values, starting, ending, and taking as many steps as you specify.

In [6]:

```
five_spaced = np.linspace(0,50,11)
print(five_spaced)
```

```
[ 0.  5. 10. 15. 20. 25. 30. 35. 40. 45. 50.]
```

The `.repeat()` method will repeat an object you pas a specified number of times.

In [28]:

```
twoArr = np.repeat(2, 10)
print(twoArr)
```

```
[2 2 2 2 2 2 2 2 2 2]
```

The `np.eye()` functions will create an identity matrix/array for us.

In [12]:

```
identity_matrix = np.eye(6)
print(identity_matrix)
```

```
[[1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1.]]
```

Operations

There are many, many operations which we can perform on arrays. Below, we demonstrate a few.

What is happening in each line?

In [14]:

```
five_times_table
```

Out[14]:

```
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45, 50])
```

In [15]:

```
print("1:", 2 * five_times_table)
print("2:", 10 + five_times_table)
print("3:", five_times_table - 1)
print("4:", five_times_table/5)
print("5:", five_times_table **2)
print("6:", five_times_table < 20)
```

```
1: [  0  10  20  30  40  50  60  70  80  90 100]
2: [10 15 20 25 30 35 40 45 50 55 60]
3: [-1  4  9 14 19 24 29 34 39 44 49]
4: [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
5: [   0   25  100  225  400  625  900 1225 1600 2025 2500]
6: [ True  True  True  True False False False False False False Fals
e]
```

Speed Test

If we compare the speed at which we can do these operations compared to core python, we eill notice a substantial difference.

In [27]:

```
fives_list = list(range(0,5001,5))
fives_list
```

Out[27]:

```
[0,
 5,
 10,
 15,
 20,
 25,
 30,
 35,
 40,
 45,
 50,
 55,
 60,
 65,
 70,
 75,
 80,
 85,
 90,
 95,
 100,
 105,
 110,
 115,
 120,
 125,
 130,
 135,
 140,
 145,
 150,
 155,
 160,
 165,
 170,
 175,
 180,
 185,
 190,
 195,
 200,
 205,
 210,
 215,
 220,
 225,
 230,
 235,
 240,
 245,
 250,
 255,
 260,
 265,
 270,
 275,
 280,
 285,
 290,
```

295,
300,
305,
310,
315,
320,
325,
330,
335,
340,
345,
350,
355,
360,
365,
370,
375,
380,
385,
390,
395,
400,
405,
410,
415,
420,
425,
430,
435,
440,
445,
450,
455,
460,
465,
470,
475,
480,
485,
490,
495,
500,
505,
510,
515,
520,
525,
530,
535,
540,
545,
550,
555,
560,
565,
570,
575,
580,
585,
590,
595,

600,
605,
610,
615,
620,
625,
630,
635,
640,
645,
650,
655,
660,
665,
670,
675,
680,
685,
690,
695,
700,
705,
710,
715,
720,
725,
730,
735,
740,
745,
750,
755,
760,
765,
770,
775,
780,
785,
790,
795,
800,
805,
810,
815,
820,
825,
830,
835,
840,
845,
850,
855,
860,
865,
870,
875,
880,
885,
890,
895,
900,

905,
910,
915,
920,
925,
930,
935,
940,
945,
950,
955,
960,
965,
970,
975,
980,
985,
990,
995,
1000,
1005,
1010,
1015,
1020,
1025,
1030,
1035,
1040,
1045,
1050,
1055,
1060,
1065,
1070,
1075,
1080,
1085,
1090,
1095,
1100,
1105,
1110,
1115,
1120,
1125,
1130,
1135,
1140,
1145,
1150,
1155,
1160,
1165,
1170,
1175,
1180,
1185,
1190,
1195,
1200,
1205,

1210,
1215,
1220,
1225,
1230,
1235,
1240,
1245,
1250,
1255,
1260,
1265,
1270,
1275,
1280,
1285,
1290,
1295,
1300,
1305,
1310,
1315,
1320,
1325,
1330,
1335,
1340,
1345,
1350,
1355,
1360,
1365,
1370,
1375,
1380,
1385,
1390,
1395,
1400,
1405,
1410,
1415,
1420,
1425,
1430,
1435,
1440,
1445,
1450,
1455,
1460,
1465,
1470,
1475,
1480,
1485,
1490,
1495,
1500,
1505,
1510,

1515,
1520,
1525,
1530,
1535,
1540,
1545,
1550,
1555,
1560,
1565,
1570,
1575,
1580,
1585,
1590,
1595,
1600,
1605,
1610,
1615,
1620,
1625,
1630,
1635,
1640,
1645,
1650,
1655,
1660,
1665,
1670,
1675,
1680,
1685,
1690,
1695,
1700,
1705,
1710,
1715,
1720,
1725,
1730,
1735,
1740,
1745,
1750,
1755,
1760,
1765,
1770,
1775,
1780,
1785,
1790,
1795,
1800,
1805,
1810,
1815,

1820,
1825,
1830,
1835,
1840,
1845,
1850,
1855,
1860,
1865,
1870,
1875,
1880,
1885,
1890,
1895,
1900,
1905,
1910,
1915,
1920,
1925,
1930,
1935,
1940,
1945,
1950,
1955,
1960,
1965,
1970,
1975,
1980,
1985,
1990,
1995,
2000,
2005,
2010,
2015,
2020,
2025,
2030,
2035,
2040,
2045,
2050,
2055,
2060,
2065,
2070,
2075,
2080,
2085,
2090,
2095,
2100,
2105,
2110,
2115,
2120,

2125,
2130,
2135,
2140,
2145,
2150,
2155,
2160,
2165,
2170,
2175,
2180,
2185,
2190,
2195,
2200,
2205,
2210,
2215,
2220,
2225,
2230,
2235,
2240,
2245,
2250,
2255,
2260,
2265,
2270,
2275,
2280,
2285,
2290,
2295,
2300,
2305,
2310,
2315,
2320,
2325,
2330,
2335,
2340,
2345,
2350,
2355,
2360,
2365,
2370,
2375,
2380,
2385,
2390,
2395,
2400,
2405,
2410,
2415,
2420,
2425,

2430,
2435,
2440,
2445,
2450,
2455,
2460,
2465,
2470,
2475,
2480,
2485,
2490,
2495,
2500,
2505,
2510,
2515,
2520,
2525,
2530,
2535,
2540,
2545,
2550,
2555,
2560,
2565,
2570,
2575,
2580,
2585,
2590,
2595,
2600,
2605,
2610,
2615,
2620,
2625,
2630,
2635,
2640,
2645,
2650,
2655,
2660,
2665,
2670,
2675,
2680,
2685,
2690,
2695,
2700,
2705,
2710,
2715,
2720,
2725,
2730,

2735,
2740,
2745,
2750,
2755,
2760,
2765,
2770,
2775,
2780,
2785,
2790,
2795,
2800,
2805,
2810,
2815,
2820,
2825,
2830,
2835,
2840,
2845,
2850,
2855,
2860,
2865,
2870,
2875,
2880,
2885,
2890,
2895,
2900,
2905,
2910,
2915,
2920,
2925,
2930,
2935,
2940,
2945,
2950,
2955,
2960,
2965,
2970,
2975,
2980,
2985,
2990,
2995,
3000,
3005,
3010,
3015,
3020,
3025,
3030,
3035,

3040,
3045,
3050,
3055,
3060,
3065,
3070,
3075,
3080,
3085,
3090,
3095,
3100,
3105,
3110,
3115,
3120,
3125,
3130,
3135,
3140,
3145,
3150,
3155,
3160,
3165,
3170,
3175,
3180,
3185,
3190,
3195,
3200,
3205,
3210,
3215,
3220,
3225,
3230,
3235,
3240,
3245,
3250,
3255,
3260,
3265,
3270,
3275,
3280,
3285,
3290,
3295,
3300,
3305,
3310,
3315,
3320,
3325,
3330,
3335,
3340,

3345,
3350,
3355,
3360,
3365,
3370,
3375,
3380,
3385,
3390,
3395,
3400,
3405,
3410,
3415,
3420,
3425,
3430,
3435,
3440,
3445,
3450,
3455,
3460,
3465,
3470,
3475,
3480,
3485,
3490,
3495,
3500,
3505,
3510,
3515,
3520,
3525,
3530,
3535,
3540,
3545,
3550,
3555,
3560,
3565,
3570,
3575,
3580,
3585,
3590,
3595,
3600,
3605,
3610,
3615,
3620,
3625,
3630,
3635,
3640,
3645,

3650,
3655,
3660,
3665,
3670,
3675,
3680,
3685,
3690,
3695,
3700,
3705,
3710,
3715,
3720,
3725,
3730,
3735,
3740,
3745,
3750,
3755,
3760,
3765,
3770,
3775,
3780,
3785,
3790,
3795,
3800,
3805,
3810,
3815,
3820,
3825,
3830,
3835,
3840,
3845,
3850,
3855,
3860,
3865,
3870,
3875,
3880,
3885,
3890,
3895,
3900,
3905,
3910,
3915,
3920,
3925,
3930,
3935,
3940,
3945,
3950,

3955,
3960,
3965,
3970,
3975,
3980,
3985,
3990,
3995,
4000,
4005,
4010,
4015,
4020,
4025,
4030,
4035,
4040,
4045,
4050,
4055,
4060,
4065,
4070,
4075,
4080,
4085,
4090,
4095,
4100,
4105,
4110,
4115,
4120,
4125,
4130,
4135,
4140,
4145,
4150,
4155,
4160,
4165,
4170,
4175,
4180,
4185,
4190,
4195,
4200,
4205,
4210,
4215,
4220,
4225,
4230,
4235,
4240,
4245,
4250,
4255,

4260,
4265,
4270,
4275,
4280,
4285,
4290,
4295,
4300,
4305,
4310,
4315,
4320,
4325,
4330,
4335,
4340,
4345,
4350,
4355,
4360,
4365,
4370,
4375,
4380,
4385,
4390,
4395,
4400,
4405,
4410,
4415,
4420,
4425,
4430,
4435,
4440,
4445,
4450,
4455,
4460,
4465,
4470,
4475,
4480,
4485,
4490,
4495,
4500,
4505,
4510,
4515,
4520,
4525,
4530,
4535,
4540,
4545,
4550,
4555,
4560,

4565,
4570,
4575,
4580,
4585,
4590,
4595,
4600,
4605,
4610,
4615,
4620,
4625,
4630,
4635,
4640,
4645,
4650,
4655,
4660,
4665,
4670,
4675,
4680,
4685,
4690,
4695,
4700,
4705,
4710,
4715,
4720,
4725,
4730,
4735,
4740,
4745,
4750,
4755,
4760,
4765,
4770,
4775,
4780,
4785,
4790,
4795,
4800,
4805,
4810,
4815,
4820,
4825,
4830,
4835,
4840,
4845,
4850,
4855,
4860,
4865,

```
 4870,
 4875,
 4880,
 4885,
 4890,
 4895,
 4900,
 4905,
 4910,
 4915,
 4920,
 4925,
 4930,
 4935,
 4940,
 4945,
 4950,
 4955,
 4960,
 4965,
 4970,
 4975,
 4980,
 4985,
 4990,
 4995,
 ...]
```

In [28]:

```
five_times_table_lge = np.arange(0,5001,5)
five_times_table_lge
```

Out[28]:

```
array([   0,    5,   10, ..., 4990, 4995, 5000])
```

In [29]:

```
%timeit five_times_table_lge + 5
```

```
1.03 µs ± 5.71 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops
each)
```

In [30]:

```
%timeit [e + 5 for e in fives_list]
```

```
46.9 µs ± 609 ns per loop (mean ± std. dev. of 7 runs, 10000 loops ea
ch)
```

Boolean string operations can also be performed on ndarrays.

In [40]:

```
words = np.array(["ten", "nine", "eight", "seven", "six"])

print(np.isin(words, 'e'))

print("e" in words)
["e" in word for word in words]
```

```
[False False False False False]
False
```

Out[40]:

```
[True, True, True, True, False]
```

Transpose

What does it mean to transpose an array/vector?

Numpy allows you to perform transpose operations with ease, and comes in very useful when matrix multiplication is required.

We simply use `<ndarray>.T` .

In [15]:

```
people.shape = (2, 6)
print(people, "\n")
print(people.T)
```

```
[[10  5  8 32 65 43]
 [32 18 26 60 55 65]]

[[10 32]
 [ 5 18]
 [ 8 26]
 [32 60]
 [65 55]
 [43 65]]
```

Data Types

As previously mentioned, ndarrays can only have one data type. If we want to obtain or change this, we use the `.dtype` attribute.

In [16]:

```
people.dtype
```

Out[16]:

```
dtype('int64')
```

What is the data type of the below ndarray?

In [17]:

```
ages_with_strings = np.array([10, 5, 8, '32', '65', '43'])
ages_with_strings
```

Out[17]:

```
array(['10', '5', '8', '32', '65', '43'], dtype='<U21')
```

What is the dtype of this array?

In [18]:

```
ages_with_strings = np.array([10, 5, 8, '32', '65', '43'], dtype='int32')
ages_with_strings
```

Out[18]:

```
array([10,  5,  8, 32, 65, 43], dtype=int32)
```

What do you think has happened here?

In [19]:

```
ages_with_strings = np.array([10, 5, 8, '32', '65', '43'])
print(ages_with_strings)
```

```
['10' '5' '8' '32' '65' '43']
```

In [20]:

```
ages_with_strings.dtype = 'int32'
print(ages_with_strings)
```

```
[49 48  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 53
0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 56  0  0  0
0  0
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 51 50  0  0  0  0  0
0  0
  0  0  0  0  0  0  0  0  0  0  0  0 54 53  0  0  0  0  0  0  0  0
0  0
  0  0  0  0  0  0  0  0  0 52 51  0  0  0  0  0  0  0  0  0  0  0
0  0
  0  0  0  0  0  0]
```

In [21]:

```
ages_with_strings.size
```

Out[21]:

```
126
```

In [22]:

```
ages_with_strings.size/21
```

Out[22]:

```
6.0
```

In [23]:

```
np.array([10, 5, 8, '32', '65', '43']).size
```

Out[23]:

```
6
```

The correct way to have changed the data type of the ndarray would have been to use the `.astype()` method, demonstrated below.

In [24]:

```
ages_with_strings = np.array([10, 5, 8, '32', '65', '43'])
print(ages_with_strings)
print(ages_with_strings.astype('int32'))
```

```
['10' '5' '8' '32' '65' '43']
[10  5  8 32 65 43]
```

Exercise

- #### Create an array of string numbers, but use dtype to make it an array of floats.
- #### Transpose the matrix, printing the new size and shape.
- #### Use the .astype() method to convert the array to boolean.

Q: What is the difference between .astype() and dtype?

- .astype() - a method to convert the type of an existing array to another
- dtype - an attribute of the array which can be specified on instantiation, or edited later, however it is not the recommended way to change type.

Array Slicing Operations

As before, we can use squeare brackets and indices to access individual values, and the colon operator to slice the array.

In [29]:

```
five_times_table
```

Out[29]:

```
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45, 50])
```

In [30]:

```
five_times_table[0]
```

Out[30]:

```
0
```

In [31]:

```
five_times_table[-1]
```

Out[31]:

```
50
```

In [32]:

```
five_times_table[:4]
```

Out[32]:

```
array([ 0,  5, 10, 15])
```

In [33]:

```
five_times_table[4:]
```

Out[33]:

```
array([20, 25, 30, 35, 40, 45, 50])
```

We can also slice an n-dim ndarray., specifying the slice operation accross each axis.

In [34]:

```
print(people)
people[:3, :3]
```

```
[[10  5  8 32 65 43]
 [32 18 26 60 55 65]]
```

Out[34]:

```
array([[10,  5,  8],
       [32, 18, 26]])
```

Exercise

- Create a numpy array with 50 zeros
- Create a np array of 2 repeated 20 times
- Create a numpy array from 0 to 2 π in steps of 0.1

For one of the arrays generated:

- Get the first five values
- Get the last 3 values
- Get the 4th value to the 7th value

_Solution----------

In [35]:

```
seq = np.arange(0, 2*np.pi, 0.1)
print(seq)
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4 1.5 1.6
1.7
 1.8 1.9 2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.  3.1 3.2 3.3 3.4
3.5
 3.6 3.7 3.8 3.9 4.  4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.  5.1 5.2
5.3
 5.4 5.5 5.6 5.7 5.8 5.9 6.  6.1 6.2]
```

In [36]:

```
firstFive = fives[:5]
print(firstFive)
```

```
---------------------------------------------------------------------
-------
NameError                                 Traceback (most recent cal
l last)
<ipython-input-36-21f158d4a6bc> in <module>
----> 1 firstFive = fives[:5]
      2 print(firstFive)

NameError: name 'fives' is not defined
```

In [ ]:

```
lastThree = fives[-3:]
print(lastThree)
```

In [ ]:

```
selection = fives[4:9]
print(selection)
```

We can reverse an array by using `.flip()` or by using the `::` operator.

In [37]:

```
reverse_five_times_table = np.flip(five_times_table)
reverse_five_times_table
```

Out[37]:

```
array([50, 45, 40, 35, 30, 25, 20, 15, 10,  5,  0])
```

In [38]:

```
reverse_five_times_table = five_times_table[-1::-1]
print(reverse_five_times_table)
five_times_table
```

```
[50 45 40 35 30 25 20 15 10  5  0]
```

Out[38]:

```
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45, 50])
```

We can also use the `::` operator to select steps of the original array.

In [39]:

```
five_times_table[0::3] #Every 3rd element starting from 0
```

Out[39]:

```
array([ 0, 15, 30, 45])
```

Exercise

Take one of the arrays you defined and

- #### Reverse it
- #### Only keep every 4th element.
- #### Get every 2nd element, starting from the last and moving backwards.

Stats

Numpy allows us to compute various statistical properties of an array by using simple method calls.

What does the below method do?

In [44]:

```
np.array([1.65432, 5.98765]).round(2)
```

Out[44]:

```
array([1.65, 5.99])
```

In [46]:

```
nums = np.arange(0, 4, 0.2555)
```

Exercise

- Compute min, max, sum, mean, median, variance, and standard deviation of the above array, all to to 2 decimal places.

In [47]:

```
print("min = ", np.min(nums).round(2))
print("max = ", np.max(nums).round(2))
print("sum = ", np.sum(nums).round(2))
print("mean = ", np.mean(nums).round(2))
print("median = ", np.median(nums).round(2))
print("var = ", np.var(nums).round(2))
print("std = ", np.std(nums).round(2))
```

```
min =  0.0
max =  3.83
sum =  30.66
mean =  1.92
median =  1.92
var =  1.39
std =  1.18
```

Random

With `np.random` , we can generate a number of types of dataset, and create training data.

The below code simulates a fair coin toss.

In [63]:

```
flip = np.random.choice([0,1], 10)
flip
```

Out[63]:

```
array([1, 1, 1, 1, 1, 1, 1, 0, 1, 1])
```

In [46]:

```
np.random.rand(10,20,9)
```

Out[46]:

```
array([[[0.56965633, 0.43786449, 0.32686642, ..., 0.38985598,
         0.44218475, 0.52122425],
        [0.2458722 , 0.85972478, 0.09142689, ..., 0.95382345,
         0.98523035, 0.27611836],
        [0.53185301, 0.16903017, 0.05014549, ..., 0.3206893 ,
         0.64742251, 0.38963055],
        ...,
        [0.02193858, 0.75205133, 0.78913665, ..., 0.35906987,
         0.43943691, 0.29535861],
        [0.70239244, 0.88379607, 0.04613278, ..., 0.55238248,
         0.059907  , 0.27837216],
        [0.38040203, 0.80778392, 0.72716391, ..., 0.57795181,
         0.79543993, 0.7469111 ]],

       [[0.48784697, 0.9731576 , 0.53867866, ..., 0.66648103,
         0.86033606, 0.81663827],
        [0.80743728, 0.84272635, 0.86629897, ..., 0.87839853,
         0.74166232, 0.61783922],
        [0.49713947, 0.17212248, 0.21084869, ..., 0.91832743,
         0.52915891, 0.88963326],
        ...,
        [0.03107481, 0.1206452 , 0.48917524, ..., 0.92654952,
         0.03784272, 0.47964962],
        [0.33691513, 0.58975299, 0.32728182, ..., 0.02028508,
         0.90037774, 0.07793587],
        [0.78030843, 0.87770165, 0.07694953, ..., 0.98996348,
         0.78465144, 0.66524646]],

       [[0.1824377 , 0.13828449, 0.47466649, ..., 0.19126558,
         0.40374537, 0.27107151],
        [0.86560956, 0.51179951, 0.49056991, ..., 0.20737973,
         0.2168636 , 0.04724461],
        [0.11029802, 0.12848733, 0.57617064, ..., 0.79896648,
         0.48150717, 0.20814662],
        ...,
        [0.44386522, 0.26930424, 0.86918533, ..., 0.82112955,
         0.62530818, 0.48121986],
        [0.40262981, 0.5544588 , 0.51624514, ..., 0.30722067,
         0.29765281, 0.30048486],
        [0.04239027, 0.37654263, 0.90880896, ..., 0.0634595 ,
         0.31818081, 0.97522414]],

       ...,

       [[0.46047448, 0.72193374, 0.86308918, ..., 0.19252748,
         0.48832444, 0.21529991],
        [0.2007859 , 0.21382435, 0.40356511, ..., 0.04508325,
         0.91245445, 0.40602681],
        [0.2971383 , 0.09228396, 0.51792191, ..., 0.88007127,
         0.97490349, 0.89521536],
        ...,
        [0.01596619, 0.40231244, 0.89366801, ..., 0.51322254,
         0.17389239, 0.9004429 ],
        [0.0973504 , 0.56642194, 0.9281889 , ..., 0.15249537,
         0.73506992, 0.89542452],
        [0.24708041, 0.62602737, 0.17297692, ..., 0.85586734,
         0.94048883, 0.78968573]],

       [[0.94972929, 0.62115069, 0.97492165, ..., 0.71551252,
```

```
         0.62052194, 0.97309311],
        [0.17463346, 0.57159244, 0.90765261, ..., 0.38361668,
         0.63314946, 0.03244099],
        [0.99494044, 0.20837834, 0.54529705, ..., 0.13974728,
         0.56235978, 0.39343011],
        ...,
        [0.72429005, 0.54853468, 0.11028382, ..., 0.37628346,
         0.74197689, 0.90172052],
        [0.97318587, 0.09522832, 0.22217468, ..., 0.73139621,
         0.84709923, 0.31423138],
        [0.66809375, 0.33178021, 0.33793697, ..., 0.99906372,
         0.58293636, 0.80479623]],

       [[0.29690175, 0.82942721, 0.92938406, ..., 0.62893332,
         0.45028359, 0.57304334],
        [0.72604183, 0.21460279, 0.84492842, ..., 0.13757703,
         0.01941592, 0.19401543],
        [0.69399724, 0.63747152, 0.12650449, ..., 0.88486362,
         0.22165696, 0.49532484],
        ...,
        [0.50915288, 0.22669814, 0.44627994, ..., 0.32549298,
         0.77751499, 0.15101588],
        [0.05698444, 0.85792681, 0.8457406 , ..., 0.17358866,
         0.69746282, 0.74985247],
        [0.89076095, 0.84905039, 0.9851012 , ..., 0.76730005,
         0.30120341, 0.19182771]]])
```

We can produce 1000 datapoints of a normally distributed data set by using `np.random.normal()`

In [66]:

```
mu, sigma = 0, 0.1 # mean and standard deviation
s = np.random.normal(mu, sigma, 1000)
```

Exercise

- Simulate a six-sided dice using numpy.random.choice(), generate a list of values you would obtain from 10 throws.
- Simulate a two-sided coin toss that is NOT fair: it is twice as likely to have head than tails.
- Challenge: produce a Poisson distribution with 100 data points.

Frompyfunct

Create a function to find if the string contains the substring

`* Hint frompyfunction()`

In [ ]:

```
def contains(string, substring):
    return substring in string
```

- Frompyfunc(func name, number of inputs, number of outputs)
- Returns a numpy universal function
  - Universal function - works with nd arrays
  - Allows us to perform element wise operations

In [ ]:

```
#without frompyfunc it won't work
contains(words, 'e')
```

In [ ]:

```
contains = np.frompyfunc(contains, 2, 1)
```

In [ ]:

```
contains(words, "e")
```