All programming languages were created to interact with the computer, i.e., make the computer perform a task.

So if you wanted to make the computer add two or more numbers, you will write your own code (or function) from scratch using a programming language.

If you wanted to do a linear regression of on one quantitative variable to another, you will also have to write your own code from scratch.

Luckily, you do not have to do this from scratch all the time, because other programmers around the world have already written code for scientific, mathematical computations and packaged them into something called a **library**. All we have to do then, is to get (download and install) this libary and use it.

NumPy is an open-source Python library that facilitates efficient numerical operations on large quantities of data.

Numpy stands for ***Numerical Python***.

Why use NumPy?

- Math operations are 50x faster on NumPy's ndarray object than on native Python lists.
- It contains built-in functions that facilitate working with arrays and math, such as functions for linear algebra, array transformations, and matrix math.
- It requires fewer lines of code for most mathematical operations than native Python lists.

> ***Think of NumPy as a separate software/tool for mathematical computation. To use this tool, you will need Python syntax.***
>
> > *To find out more about NumPy you can check its documentation:* [NumPy Documentation (https://numpy.org/doc/stable/)](https://numpy.org/doc/stable/)

Quick start - Installation

Note: This guide will assume that the user will be using Jupyter Notebook to follow along.

If you installed Anaconda, NumPy comes pre-installed and no further installation steps are necessary. Skip to Importing NumPy.

However, if this is not the case you can type (or copy and paste) and run the following command in your Jupyter Notebook:

```
!pip install numpy
```

or

```
!conda install numpy
```

Importing NumPy

To access NumPy and its functions import it in your Python code like this:

```
import numpy as np
```

> Because of the `as np` statement, we are able to call numpy by this name: `np`, which saves typing when coding. We do not have to type out `numpy` in full after running this code. To use numpy we simply type `np`. Like a nick name. Instead of calling your friend, ***Joseph***'s name in full, you decide to call him ***Joe***.

Using NumPy

As earlier mentioned, NumPy is like a separate software/tool. Just as a software like Excel has built-in functions for varying purposes, NumPy also has numerous useful functions. These functions can be broadly *divied* up into a few categories. Hence, Numpy has functions for:

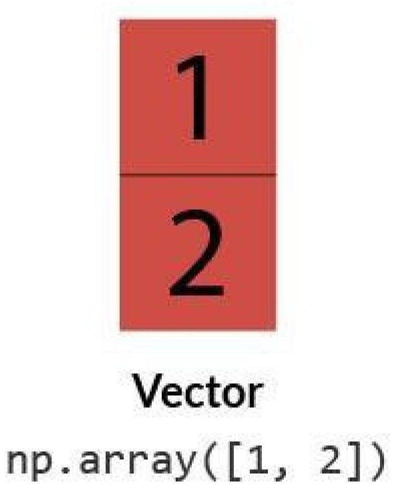
- Array Creation
- Conversions
- Manipulations
- Exploration
- Ordering
- Operations
- Basic Statistics
- Basic Linear Algebra

Creating Arrays

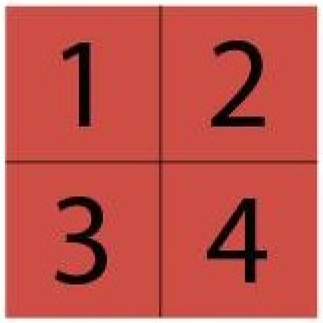
The NumPy array is the basis of the NumPy package.

A NumPy array is an **`n`** -dimensional data structure.

A **`one`** -dimensional NumPy array can be thought of as a **`vector`** .



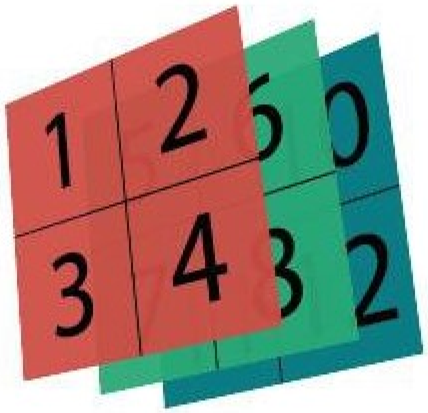
A **`two`** -dimensional array as a **`matrix`** (i.e., a set of vectors)



Matrix

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

A **`three`** -dimensional array as a **`tensor`** (i.e., a set of matrices).



3D Matrix

```
np.array([[[1, 2], [3, 4]],
          [[5, 6], [7, 8]],
          [[9, 10], [11, 12]]])
```

> An array can consist of integers, floating-point numbers, or strings. However, the data type must be ***consistent*** within an array,i.e., all values must be integers or all floats.

`np.array()`

To define an array, we can use the **`np.array()`** function.

Below, we pass a list of two elements, each of which is a list containing two values. The result is

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

In [2]:
```
np.array([[1,2],[3,4]])
```

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

In [3]:
```
list1 = [1,2,3,4]
array1 = np.array(list1)

array1
```

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

In [4]:
```
list2 = [[1,2,3],[4,5,6]]
array2 = np.array(list2)

array2
```

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

`np.arange()`

The **`np.arange()`** function is great for creating vectors very easily. It is often used to create sample data.

Here, we create a vector with values spanning 1 up to (but not including) 5:

In [5]:
```
np.arange(1, 5)
```

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

`np.zeros()`

NumPy has a function that allows us to create an array of zeroes of varying dimensions (shapes).

Here, we create an array of zeros with three rows and one column.

In [6]:
```
np.zeros((3,1))
```

Out[6]:
```
array([[0.],
       [0.],
       [0.]])
```

`np.ones()`

You can also initialize an array with ones instead of zeros:

In [7]:
```
np.ones((3, 1))
```

Out[7]:
```
array([[1.],
       [1.],
       [1.]])
```

`np.full()`

The **`np.full()`** creates an array repeating a fixed value.

Here we create a 2x3 array with the number 5 in each element:

In [8]:
```
np.full((2,3), 5)
```

Out[8]:
```
array([[5, 5, 5],
       [5, 5, 5]])
```

`np.linspace()`

We can create a linearly spaced array with **`np.linspace()`** .

In [9]:
```
arr_1 = np.linspace(0, 20, 5) # (Start, stop, step)

arr_1
```

Out[9]:
```
array([ 0.,  5., 10., 15., 20.])
```

Storing arrays in variables

As you may have thought, we can create and store arrays in variables for later use.

In [10]:
```
var_1 = np.array([[1,2],[3,4]])

var_2 = np.arange(1, 5)

var_3 = np.zeros((3,1))

var_4 = np.ones((3, 1))

var_5 = np.full((2,3), 5)
```

Pseudo-random number generation

Being able to generate pseudo-random numbers is often necessary in data science applications.

Examples include modeling the *uniform* distribution and the *normal (Gaussian)* distribution.

`np.random.rand()`

In [11]:
```
uniform_data = np.random.rand(1000)
```

`np.random.normal()`

In [12]:
```
normal_data = np.random.normal(loc=5, scale=3.0, size=1000)
```

Visualizing normal data using arrays

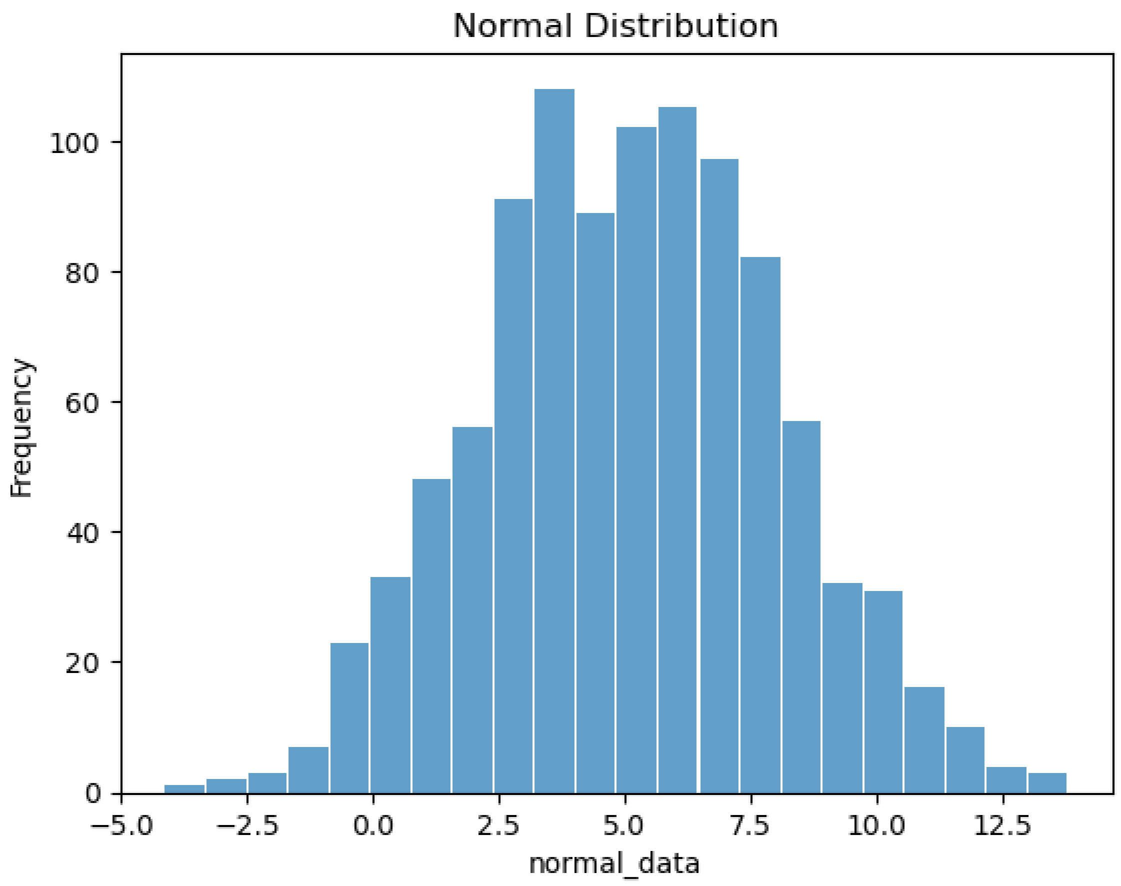
Using `matplotlib` , another popular, widely-used Python data science library for visualization, we can visualize the normal data generated with `np.random.normal()` .

In [13]:
```
import matplotlib.pyplot as plt

plt.hist(x=normal_data, bins='auto', alpha=0.7, rwidth=0.95)

plt.title("Normal Distribution")
plt.ylabel('Frequency')
plt.xlabel('normal_data')

plt.show()
```



Array shape

All arrays have a shape or a dimension which refers to their number of rows and columns

These values are accessible via the **`.shape`** method.

For example, let's get the shape of a vector

In [14]:
```
vector = np.arange(5)
print("Vector shape:", vector.shape)
```

```
Vector shape: (5,)
```

The shape of a matrix

In [15]:
```
matrix = np.ones([3, 2])
print("Matrix shape:", matrix.shape)
```

```
Matrix shape: (3, 2)
```

The shape of a tensor

In [16]:
```
tensor = np.zeros([2, 3, 3])
print("Tensor shape:", tensor.shape)
```

```
Tensor shape: (2, 3, 3)
```

NumPy array operations

Many operations can be performed on NumPy arrays which makes them very helpful for manipulating data:

- Reshaping arrays
- Indexing and Slicing arrays
- Combining arrays
- Splitting arrays
- Numerical operations (min, max, mean, etc)

Reshaping Arrays

Using `.reshape()` , we can reshape an array into any compatible dimensions.

For example, we create an array where each element is incremented from 1 to 9.

In [17]:
```
arr = np.arange(1, 10)
print(arr)
```

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

Then we reshape it to a 3x3 matrix.

In [18]:
```
arr = arr.reshape(3, 3)
print(arr)
```

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

After, we reshape back to the original size.

In [19]:
```
arr = arr.reshape(9)
print(arr)
```

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

Indexing and Slicing Arrays

At some point, it will become necessary to index (or select) subsets of a NumPy array.

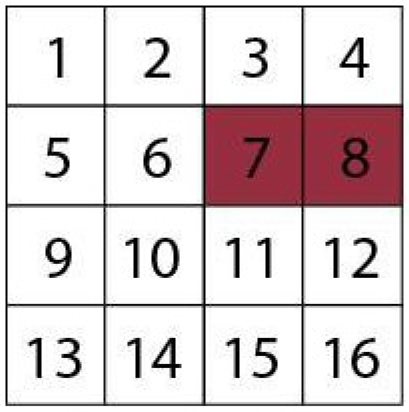
For instance, you might want to plot one column of data or perform a manipulation of that column.

- Commas separate axes of an array.
- Colons mean "through". For example, x[0:4] means the first 5 rows (rows 0 through 4) of x.
- Negative numbers mean "from the end of the array." For example, x[-1] means the last row of x.
- Blanks before or after colons means "the rest of". For example, x[3:] means the rest of the rows in x after row 3. Similarly, x[:3] means all the rows up to row 3. x[:] means all rows of x.

In [20]:
```
arr = np.arange(1, 17) # Ranging from 1 up to 17, but not including 17
arr = arr.reshape(4, 4)
arr
```

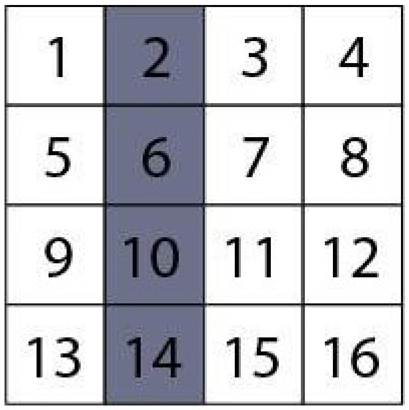
Out[20]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

We would index or slice the data in the following way:



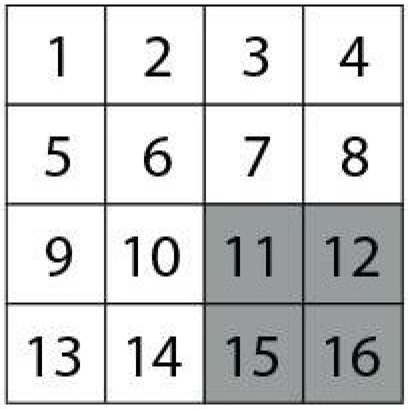
Row one, columns two to four

```
>>> arr[1, 2:4]
array([7, 8])
```



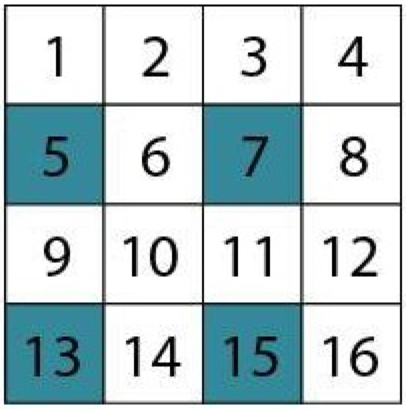
All rows in column one

```
>>> arr[:, 1]
array([2,  6, 10, 14])
```



All rows after row two,
all columns after column two

```
>>> arr[2:, 2:]
array([[11, 12],
       [15, 16]])
```



Every other row after row one,
every other column

```
>>> arr[1::2, ::2]
array([[5, 7],
       [13, 15]])
```

Combining or Concatenating Arrays

NumPy also provides useful functions for concatenating (i.e., joining) arrays.

Let's say we wanted to restrict our attention to the first and the last rows of our array, **`arr`** .

First, we'll define new sub-arrays as follows:

In [21]:
```
array_start = arr[0, :]

array_start
```

Out[21]:
```
array([1, 2, 3, 4])
```

In [22]:
```
array_end = arr[-1, :]

array_end
```

Out[22]:
```
array([13, 14, 15, 16])
```

np.vstack()

To concatenate these arrays we can use **`np.vstack`** , where the ***v*** denotes vertical, or row-wise stacking of the sub-arrays:

In [23]:
```
np.vstack((array_start, array_end))
```

Out[23]:
```
array([[ 1,  2,  3,  4],
       [13, 14, 15, 16]])
```

Above we stacked the first row and last row on top of each other.

`np.hstack()`

The horizontal counterpart of **`np.vstack()`** is **`np.hstack()`** , which combines sub-arrays column-wise.

In [24]:
```
np.hstack((array_start, array_end))
```

Out[24]:
```
array([ 1,  2,  3,  4, 13, 14, 15, 16])
```

`np.concatenate()`

The **`np.concatenate()`** function can also perform the same tasks as both **`np.vstack()`** and **`np.hstack()`** .

The syntax for this function is similar to that of both **`np.vstack()`** and **`np.hstack()`** , with the additional requirement of specifying the axis along which concatenation should be performed.

`axis = 0` ***will produce similar output as*** ***`np.vstack()`***

In [25]:
```
np.concatenate((array_start, array_end), axis = 0)
```

Out[25]:
```
array([ 1,  2,  3,  4, 13, 14, 15, 16])
```

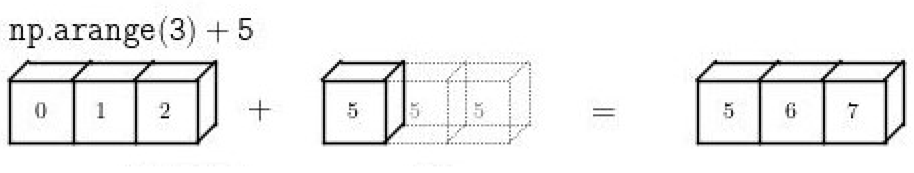
Splitting Arrays

The opposite of concatenating (i.e., joining) arrays is splitting them. To split an array, NumPy provides the following commands:

- **`hsplit`** : splits along the horizontal axis
- **`vsplit`** : splits along the vertical axis
- **`dsplit`** : Splits an array along the 3rd axis (depth)
- **`array_split`** : lets you specify the axis to use in splitting

Numerical Operations on Arrays (Broadcasting)

Broadcasting is a process performed by NumPy that allows mathematical operations to be performed on all elements of an array at the same time.



```
In [26]: my_arr = np.arange(3)

         my_arr

Out[26]: array([0, 1, 2])

In [27]: my_arr + 5

Out[27]: array([5, 6, 7])

In [28]: my_arr - 1

Out[28]: array([-1,  0,  1])

In [29]: my_arr * 2

Out[29]: array([0, 2, 4])

In [30]: my_arr / 2

Out[30]: array([0. , 0.5, 1. ])
```