1. NumPy

NumPy is short for “Numerical Python”. NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

NumPy also offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.

1.1. Array Processing

NumPy provides an N-dimensional array type, the **ndarray**, which describes a collection of “items” of the same type. We can create a NumPy ndarray object by using the **array()** function. We can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray. For example,

```
import numpy as np
x = np.array( [1, 4, 3])
print(type(x)) #<class 'numpy.ndarray'>
```

The items can be indexed using for example N integers. NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

1.2. Array Dimensions

Zero-dimension (0-D) arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array. For example,

```
arr = np.array(42)
print(arr)
```

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array. These are the most common and basic arrays. For example,

```
arr = np.array([1, 2, 3, 4, 5])
```

An array that has 1-D arrays as its elements is called a 2-D array. These are often used to represent matrix. For example,

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

Similarly, an array that has 2-D arrays as its elements is called 3-D array, an array that has 3-D arrays as its elements is called 4-D array and so on.

1.3. Checking Number of Dimensions

We can use **ndim** attribute to return an integer that tells us how many dimensions the array has. For example,

```
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

1.4. Size and Shape Attributes

The **size** attribute returns the total number of elements in the array. This is the product of the elements of the array's shape. For example,

```
c = np.array([[1,2,3],[4,5,6]])
print(c.size)
```

The **shape** attribute will display a tuple of integers that indicate the number of elements stored along each dimension of the array. For example,

```
c = np.array([[1,2,3],[4,5,6]])
print(c.shape)
```

We can also **reshape** the shape of an array. By reshaping we can add or remove dimensions or change number of elements in each dimension. For example,

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

Flattening array means converting a multidimensional array into a 1D array. We can use reshape(-1) to do this. For example,

```
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
newarr = arr.reshape(-1)
print(newarr)
```

1.5. Array Indexing

You can access an array element by referring to its index number. The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 and so on. For example,

```
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element. We can access higher dimension array in the same way. For example,

```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(arr[0, 1])
```

We can also use negative indexing to access an array from the end. The example below prints second last element of second dimension.

```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print(arr[1, -2])
```

1.6. Array Slicing

Array slicing is the process of taking elements from one given index to another given index. The result includes the start index, but excludes the end index. For example,

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

If we don’t use start index it is considered 0, if we don’t use end index it is considered the length of the array in that dimension and if we don’t consider both indices every element in the array is considered. For example,

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:5])
print(arr[2:])
print(arr[:])
```

We can also define the step during array slicing such as [start:end:step]. The default value for step is 1. For example,

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
```

We can also use minus operator to refer to an index from the end. For example, to Slice from the index 3 from the end to index 1 from the end we write.

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
```

We can also slice two-dimensional array. For example, to slice elements from index 1 to index 4 (not included) from the second element of the array, we write

```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
```

To return index 2 from both elements, we write

```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 2])
```

To slice index 1 to index 4 (not included) from both elements of the two-dimensional array, we write

```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 1:4])
```

1.7. NumPy Data Types

NumPy supports a much greater variety of numerical types than Python does. The table below shows these data types.

Numpy type	**C type**	**Description**
numpy.bool_	bool	Boolean (True or False) stored as a byte
numpy.byte	signed char	Platform-defined
numpy.ubyte	unsigned char	Platform-defined
numpy.short	short	Platform-defined
numpy.ushort	unsigned short	Platform-defined
numpy.intc	int	Platform-defined
numpy.uintc	unsigned int	Platform-defined
numpy.int_	long	Platform-defined
numpy.uint	unsigned long	Platform-defined
numpy.longlong	long long	Platform-defined
numpy.ulonglong	unsigned long long	Platform-defined
numpy.half / numpy.float16		Half precision float: sign

		bit, 5 bits exponent, 10 bits mantissa
numpy.single	float	Platform-defined single precision float: typically sign bit, 8 bits exponent, 23 bits mantissa
numpy.double	double	Platform-defined double precision float: typically sign bit, 11 bits exponent, 52 bits mantissa.
numpy.longdouble	long double	Platform-defined extended-precision float
numpy.csingle	float complex	Complex number, represented by two single-precision floats (real and imaginary components)
numpy.cdouble	double complex	Complex number, represented by two double-precision floats (real and imaginary components).
numpy.clongdouble	long double complex	Complex number, represented by two extended-precision floats (real and imaginary components).

1.8. Creating Arrays with Data Types

When you use numpy.array() function to define a new array, you can also consider the dtype of the elements in the array, which can be specified explicitly. For example,

a = np.array([127, 128, 129], dtype = np.intc)

1.9. I/O with NumPy

The ndarray objects can be saved to and loaded from the disk files. The I/O functions available are:

- The load() and save() functions handle numPy binary files (with npy extension).
- The loadtxt() and savetxt() functions handle normal text files.

NumPy introduces a simple file format for ndarray objects. This .npy file stores data, shape, dtype and other information required to reconstruct the ndarray in a disk file such that the array is correctly retrieved even if the file is on another machine with different architecture.

The save() function stores the input array in a disk file with .npy extension. For example,

a = np.array([1,2,3,4,5])
np.save('outfile',a)

To reconstruct array from .npy file, we use load() function. For example,

```
b = np.load('outfile.npy')
print(b)
```

The storage and retrieval of array data in simple text file format is done with savetxt() and loadtxt() functions. For example,

```
a = np.array([1,2,3,4,5])
np.savetxt('out.txt',a)
b = np.loadtxt('out.txt')
print(b)
```

1.10. NumPy Array Copy and View

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy. For example,

```
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42
print(arr)
print(x)
```

The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view. For example,

```
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42
print(arr)
print(x)
```

Every NumPy array has the attribute **base** that returns ***None*** if the array owns the data. Otherwise, the base attribute refers to the original object. For example,

```
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
y = arr.view()
print(x.base)
print(y.base)
```

1.11. Creating Array from Numerical Range

We can create numpy arrays using some given specified range. The numpy.**arrange()** function creates an array by using the evenly spaced values over the given interval. The syntax of arrange function is:

```
numpy.arrange(start, stop, step, dtype)
```

Here, **start** is the starting of an interval (default value is 0), **stop** represents the value at which the interval ends excluding this value, **step** is the number by which the interval values change, and **dtype** is the data type of the numpy array items. For example,

```
arr = np.arange(0,10,.5,float)
print(arr)
```

We can also use numpy.l**inspace()** function as the arrange function. However, it doesn’t allow us to specify the step size in the syntax. Instead of that, it only returns evenly separated values over a specified period. The system implicitly calculates the step size. The syntax is given below:

```
numpy.linspace(start, stop, num, endpoint, retstep, dtype)
```

Here, **start** represents the starting value of the interval, **stop** represents the stopping value of the interval, **num** is the amount of evenly spaced samples over the interval to be generated (default is 50), **endpoint** (Boolean value) represents the value which indicates that the stopping value is included in the interval, **rettstep** (boolean value) represents the steps and samples between the consecutive numbers and **dtype** represents the data type of the array items. For example,

```
arr = np.linspace(0, 10, 5, True, True, np.single)
print(arr)
```

1.12. Array Broadcasting

The term ***broadcasting*** refers to the ability of NumPy to treat arrays of different shapes during arithmetic operations. Arithmetic operations on arrays are usually done on corresponding elements. If two arrays are of exactly the same shape, then these operations are smoothly performed. For example,

```
a = np.array([1,2,3,4])
b = np.array([5,6,7,8])
c = a * b
print(c)
```

If the dimensions of two arrays are dissimilar, element-to-element operations are not possible. However, operations on arrays of non-similar shapes is still possible in NumPy, because of the broadcasting capability. The smaller array is broadcast to the size of the larger array so that they have compatible shapes. For example,

```
a = np.array([[1,2,3],[4,5,6]])
b = np.array([7,8,9])
c = a * b
print(c)
```

The broadcasting analogy is only conceptual. NumPy is smart enough to use the original values without actually making copies so that broadcasting operations are as memory and computationally efficient as possible.

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when they are equal, or one of them is 1. For example,

array-1: 8 array-2: 5 × 2 × 8 result-shape: 5 × 2 × 8	array-1: 5 × 2 array-2: 5 × 4 × 2 result-shape: INCOMPATIBLE	array-1: 4 × 2 array-2: 5 × 4 × 2 result-shape: 5 × 4 × 2
array-1: 8 × 1 × 3 array-2: 8 × 5 × 3 result-shape: 8 × 5 × 3	array-1: 1 × 3 × 2 array-2: 8 × 2 result-shape: INCOMPATIBLE	array-1: 5 × 1 × 3 × 2 array-2: 9 × 1 × 2 result-shape: 5 × 9 × 3 × 2

1.13. Iterating over Array

Iterating means going through elements one by one. As we deal with arrays in numpy, we can do this using basic for loop of python. For example,

```
arr = np.array([1, 2, 3])
for x in arr:
    print(x)
```

We can also iterate on the elements of higher dimensional arrays using for loop. For example,

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
  print(x)
```

Hence, if we iterate on a n-dimensional array it will go through n-1th dimension one by one.
To return the actual values, the scalars, we have to iterate the arrays in each dimension. For example,

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
    for y in x:
        print(y)
```

We can use nditer() function to iterate through *each scalar value* of an array using just one for loop. For example,

```
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
for x in np.nditer(arr):
    print(x)
```

We can use op_dtypes argument and pass it the expected datatype to change the datatype of elements while iterating. NumPy does not change the data type of the element in-place (where the element is in array) so it needs some other space to perform this action, that extra space is called buffer, and in order to enable it in nditer() we pass flags=['buffered']. For example,

```
arr = np.array([1, 2, 3])
for x in np.nditer(arr, flags=['buffered'], op_dtypes=['S']):
    print(x)
```

We can also iterate arrays with different step size. For example, to iterate through second position of first two elements of 2D array skipping 1 position we write.

```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
for x in np.nditer(arr[0:2, 1::2]):
    print(x)
```

We use ndenumerate() method if we require corresponding index of the element while iterating, for example,

```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for idx, x in np.ndenumerate(arr):
    print(idx, x)
```

1.14. Array Searching

We can use **where()** method to search an array for a certain value. This method returns the indexes that get a match. For example,

```
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)
```

We can also find the indexes of values where the values hold some properties. For example, to search for even values in the array, we write

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr % 2 == 0)
print(x)
```

The **searchsorted()** method returns the index where the specified value would be inserted to maintain the search order. For example,

```
arr = np.array([2, 7, 8, 10])
x = np.searchsorted(arr, 7)
print(x)
```

We can also use ***side = 'right'*** to return the rightmost index during search. For example,

```
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7, side = 'right')
print(x)
```

We can also search more than one value in the array. For example,

```
arr = np.array([1, 2, 3, 5, 7])
x = np.searchsorted(arr, [2, 4, 6])
print(x)
```

1.15. Array Sorting

A variety of sorting related functions are available in NumPy. These sorting functions implement different sorting algorithms. The **sort()** function returns a sorted copy of the input array. It has the following parameters.

```
numpy.sort(a, axis, kind, order)
```

- **a** – Array to be sorted
- **axis** – The axis along which the array is to be sorted. If None, the array is flattened, sorting on the last axis
- **kind** – Default is quicksort
- **order** – If the array contains fields, the order of fields to be sorted

Example 1:

```
import numpy as np
arr = np.array([41, 8, 4, 5, 1, 9])
print(np.sort(arr))
```

Example 2:

```
import numpy as np
arr = np.array([[41, 8, 4],[5, 1, 9]])
```

```
print(np.sort(arr, 0))
```

Example 3:

```
import numpy as np
dt = np.dtype([('name', 'S10'),('age', int)])
a = np.array([("raju",21),("anil",25),("ravi", 17), ("amar",27)], dtype = dt)
print (np.sort(a, order = 'name'))
```

1.16. Counting Functions

The **count()** function returns an array with the number of non-overlapping occurrences of substring in the given range. This function has the following parameters.

numpy.char.count(arr, sub, start, end)

- **arr**: Array like input of str or Unicode.
- **sub**: Substring to search for.
- **start**, **end**: Optional arguments to specify range in which to count.

Example:

```
c = np.array(['aAaAaA', '  aA  ', 'abBABba'])
print(np.char.count(c, 'A'))
print(np.char.count(c, 'aA'))
print(np.char.count(c, 'A', start = 1, end = 4))
```

The **count_nonzero()** function counts the number of non-zero values in the array. This function has the following parameters.

numpy.count_nonzero(arr, axis, keepdims)

- **arr**: It is an array-like input for which we have to count the nonzero array's value.
- **axis**: Axis along which we have to count non-zeros. By default, it is None, meaning that non-zeros will be counted along with a flattened version of arr.
- **keepdims**: It is the boolean value and an optional input. If set to True, the axes counted are left in the result as dimensions with size one.

Example:

```
a = np.array([[0, 1, 7, 0], [3, 0, 2, 19]])
print(np.count_nonzero(a))
print(np.count_nonzero(a, axis = 0))
print(np.count_nonzero(a, axis = 1))
print(np.count_nonzero(a, axis = 1, keepdims = True))
```

1.17. Statistical Functions

The **numpy.amin()** and **numpy.amax()** functions are used to find the minimum and maximum of the array elements along the specified axis respectively. For example,

```
a = np.array([[2,10,20],[80,43,31],[22,43,10]])
print(np.amin(a))
print(np.amax(a))
print(np.amin(a, 0))
print(np.amax(a, 0))
print(np.amin(a, 1))
print(np.amax(a, 1))
```

The **numpy.ptp()** function is used to return the range of values along an axis. For example,

```
a = np.array([[2, 10, 20], [80, 43, 31], [22, 43, 10]])
print(np.ptp(a, 1))
print(np.ptp(a, 0))
```

The **numpy.mean()** function is used to calculate mean of items along an axis. And, the **numpy.meadian()** function is used to calculate median of items along an axis. For example,

```
a = np.array([[1,5,3],[14,5,6],[7,8,10]])
print(np.mean(a, 0))
print(np.mean(a, 1))
print(np.median(a, 0))
print(np.median(a, 1))
```

Similarly, we can use **numpy.std()** function is used to calculate standard deviation along an axis and **numpy.var()** function to calculate variance along an axis.

2. Pandas

Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. It is the most popular library for data analysis. There are two core data structures in pandas: **series** and **data frame**.

2.1. Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The values are labeled with index numbers. First value has index 0, second value has index 1, and so on. These labels can be used to access a specified value. For example,

```
import pandas as pd
a = [11, 7, 10]
myvar = pd.Series(a)
print(myvar)
print(myvar[1])
```

We can use index argument to name own labels. These labels can also be used to access values in the series. For example,

```
a = [1, 7, 2]
myvar = pd.Series(a, index = ["x", "y", "z"])
print(myvar)
print(myvar["y"])
```

Series can also have a name attribute. For example,

```
a = [1, 7, 2]
var1 = pd.Series(a, index = ["x", "y", "z"], name = "something")
print(var1)
```

Series can also be instantiated from dictionary. In this case, the keys of the dictionary become the labels. For example

```
marks = {"Ram": 420, "Shyam": 380, "Hari": 390}
myvar = pd.Series(marks)
print(myvar)
```

To select only some of the items in the dictionary, use the index argument and specify only the items you want to include in the Series. For example,

```
marks = {"Ram": 420, "Shyam": 380, "Hari": 390}
myvar = pd.Series(marks, index = ["Ram", "Shyam", "Sita"])
print(myvar)
```

If data is a scalar value, an index must be provided. The value will be repeated to match the length of index. For example,

```
s = pd.Series(5.0, index=["a", "b", "c", "d", "e"])
print(s)
```

Remember: Series acts very similarly to a *ndarray*, and is a valid argument to most NumPy functions.

2.2. Data Frames

A DataFrame is a 2-dimensional data structure, like a 2-dimensional array, or a table with rows and columns. Series is like a column, a DataFrame is the whole table. For example,

```
data = {
   "calories": [420, 380, 390],
   "duration": [50, 40, 45]
}
df = pd.DataFrame(data)
print(df)
```

Along with the data, we can optionally pass index (row labels) to the data frame. For example,

```
data = {
   "calories": [420, 380, 390],
   "duration": [50, 40, 45]
}
df = pd.DataFrame(data, index = ["a", "b", "c"])
print(df)
```

We can also create a data frame from series. The resulting index will be the union of the indexes of the various Series. For example,

```
d = {
   "one": pd.Series([1.0, 2.0, 3.0], index = ["a", "b", "c"]),
   "two": pd.Series([1.0, 2.0, 3.0, 4.0], index = ["a", "b", "c", "d"]),
}
df = pd.DataFrame(d)
print(df)
df1 = pd.DataFrame(d, index = ["d", "b", "a"])
print(df1)
df2 = pd.DataFrame(d, index = ["d", "b", "a"], columns = ["two", "three"])
print(df2)
```

We can create a data frame from another data frame using **copy()** method. For example,

```
data = {
   "name": ["Ram", "Shyam", "Hari"],
   "calories": [420, 380, 390],
   "duration": [50, 40, 45]
}
df = pd.DataFrame(data)
df1 = df[["name", "duration"]].copy()
print(df1)
```

Pandas use the **loc** attribute to return one or more specified row(s). For example,

```
data = {
   "calories": [420, 380, 390],
   "duration": [50, 40, 45]
}
df = pd.DataFrame(data)
print(df.loc[0])
```

You can also get, set, and delete columns from the data frame. For example,

```
data = {
   "calories": [420, 380, 390],
   "duration": [50, 40, 45]
}
df = pd.DataFrame(data)
print(df["calories"]) # get column
df["age"] = [34, 67, 14] # set column
print(df)
del(df["calories"]) # delete column
print(df)
```

2.3. Data Frame Operations

2.3.1. Head and Tail

To view a small sample of a Series or DataFrame object, use the **head()** and **tail()** methods. The default number of elements to display is five, but you may pass a custom number. For example,

```
print(df.head())
print(df.tail())
```

2.3.2. Attributes and Underlying Data

Pandas objects have a number of attributes enabling you to access the metadata. The **shape** attribute gives the axis dimensions of the object, consistent with ndarray. The **size** attribute returns the total number of elements. The **index** attribute gives index information and **columns** attribute gives column information. For example,

```
data = {
   "calories": [420, 380, 390],
   "duration": [50, 40, 45]
}
```

```
df = pd.DataFrame(data)
print(df)
print(df.shape)
print(df.size)
print(df.index)
print(df.columns)
```

2.3.3. Working with Missing Data

Missing data can occur when no information is provided for one or more items. Missing data is a very big problem in a real-life scenario. Missing Data can also refer to as NA (Not Available) values in pandas. Many datasets simply arrive with missing data, either because it exists and was not collected or it never existed. In Pandas missing data is represented by two value: **None** or **NaN**.

In order to check missing values in Pandas DataFrame, we use a function **isnull()**and **notnull()**. Both function help in checking whether a value is NaN or not. These functions can also be used in Pandas Series in order to find null values in a series. We can also use **isna()** and **notna()** functions to check missing values. For example,

```
data = {'First_Score':[100, 90, np.nan, 95],
        'Second_Score': [30, 45, 56, np.nan],
        'Third_Score':[np.nan, 40, 80, 98]}
df = pd.DataFrame(data)
df.isna()
```

In order to fill null values in a datasets, we use **fillna()**, **replace()** and **interpolate()** function. These functions replace NaN values with some value of their own. The **interpolate()** function is basically used to fill NaN values using various interpolation techniques. For example,

```
data = {'First_Score':[40, 90, np.nan, 95],
        'Second_Score': [30, 45, 56, np.nan],
        'Third_Score':[np.nan, 40, 80, 98]}
df = pd.DataFrame(data)
df.fillna(0)
```

To fill NaN values with pervious value, we use ***method*** argument with value ***"pad"***. For example,

```
df.fillna(method = "pad")
```

To fill NaN values with next value, we use ***method*** argument with value ***"bfill"***. For example,

```
df.fillna(method = "bfill")
```

We can also fill all the null values in a particular column by specifying the column name of the data frame. For example,

```
df["Second_Score"].fillna("No Score", inplace = True)
```

We can also use **interpolate()** function to fill the missing values. Linear method ignores the index and treat the values as equally spaced. For example,

```
df.interpolate(method ='linear', limit_direction ='forward', inplace = True)
```

In order to drop a null values from a data frame, we used **dropna()** function. This function drops rows or columns of the data frame with Null values. To drop rows with at least one NaN value, we write

df.dropna()

To drop rows with all data missing or containing NaN, we write

df.dropna(how = 'all')

To drop columns with at least one NaN value, we write

df.dropna(axis = 1)

To drop columns with all data missing or containing NaN, we write

df.dropna(axis = 1, how = "all")

Often times we want to replace arbitrary values with other values. The **replace()** method provides an efficient yet flexible way to perform such replacements. For example,

df.replace(to_replace = 40, value = -99)

To discover duplicates, we can use the **duplicated()** method. The *duplicated()* method returns a Boolean values for each row. For example,

df.duplicated()

To remove duplicates, use the **drop_duplicates()** method. For example,

df.drop_duplicates(inplace = True)

2.3.4. Indexing, Slicing, and Subsetting

We often want to work with subsets of a data frame. There are different ways to accomplish this including: using labels (column headings), numeric ranges, or specific x, y index locations. We use square brackets [] to select a subset of a data frame. For example,

df[['Second_Score', 'Third_Score']]
df[0:2]

We can select specific ranges of our data in both the row and column directions using either label or integer-based indexing.

- **loc** is primarily label based indexing. Integers may be used but they are interpreted as a label.
- **iloc** is primarily integer-based indexing

To select a subset of rows and columns from the data frame, we can use both *loc* and *iloc*. For example,

df.loc[0:2, 'First_Score':'Second_Score']
df.iloc[0:2, 0:1]

We can also select a specific data value using a row and column location within the data frame using *loc* and *iloc* indexing. For example,

df.loc[0,'Second_Score']
df.iloc[2, 1]

We can also select a subset of our data using criteria. For example,

df[df.Second_Score >35]

2.3.5. Fancy Indexing

Sometimes we need to give a label-based “fancy indexing” to the data frame. The concept of fancy indexing is simple which means, we have to pass an array of indices to access multiple array elements at once. This function takes arrays of row and column labels and returns an array of the values corresponding to each (row, col) pair. For example,

```
row = [0, 2]
col = ['Second_Score', 'Third_Score']
print(df.loc[row, col])
```

2.3.6. Merging and Joining

We use **merge()** method to merge data frames similar to the database join operations. This method uses the common column to merge the data frames. The argument ***how*** in this method defines the database join operation, such as, inner, left, right, and outer. And the argument **on** defines common fields, such as ["Field1", "Field2"] of data frames. For example,

```
df1 = pd.DataFrame({"Common": ["A", "B", "C", "D", "E"],
                "Name": ["John", "Alice", "Emma", "Watson", "Harry"],
                "Age": [18, 19, 20, 21, 15]})
df2 = pd.DataFrame({"Common": ["A", "B", "C", "D", "E"],
                "Sport": ["Cricket", "Football", "Table Tennis", "Badminton", "Chess"],
                "Movie": ["Jumanji", "Black Widow", "End Game", "Mr. Robot", 
"Matrix"]})
df3 = pd.merge(df1, df2, how = "left", on = "Common")
print(df3)
```

Similar to the merge method, we have a **join()** method for joining the data frames. The join method uses the index of the data frame to join data frames.
This function is used for combining the columns of two potentially differently-indexed data frames into a single result data frame. We can also use **how** and **on** arguments in the *join ()* method. For example,

```
df1 = pd.DataFrame({"Name": ["John", "Alice", "Emma", "Watson", "Harry"],
                "Age": [18, 19, 20, 21, 15]},
               index = ["A", "B", "C", "D", "X"])
df2 = pd.DataFrame({"Sport": ["Cricket", "Football", "Table Tennis", "Badminton", 
"Chess"],
                "Movie": ["Jumanji", "Black Widow", "End Game", "Mr. Robot", 
"Matrix"]},
               index = ["A", "B", "C", "D", "E"])
df3 = df1.join(df2)
print(df3)
```

2.3.7. Working with CSV and JSON Data

Pandas **read_csv()** function reads a comma-separated values (csv) file into data frame. For example,

```
df = pd.read_csv('data.csv')
print(df.to_string())
```

If the data frame has many rows, pandas will only return the first 5 rows, and the last 5 rows. To display entire data frame, we use to_string() function of the data frame. After reading csv file into a data frame, we can use all the attributes and methods of the data frame to work with the data.
We can also write data frame into a csv file. To write data frame to csv file, we use **to_csv()** function. For example,

```
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
df = pd.DataFrame(data)
df.to_csv("data.csv", index = False)
pd.read_csv("data.csv")
```

Pandas **read_json()** function reads a JavaScript Object Notation (JSON) file into data frame. For example,

```
df = pd.read_json('data.json')
print(df.to_string())
```

We can also write data frame into a JSON file. To write data frame to JSON file, we use **to_json()** function. For example,

```
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
df = pd.DataFrame(data)
df.to_json("data.json")
pd.read_json("data.json")
```

3. Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Most of the Matplotlib utilities lies under the pyplot module, and are usually imported under the ***plt*** alias.
The **plot()** function is used to draw points (markers) in a diagram. This function takes parameters for specifying points in the diagram. First parameter is an array containing points on the x-axis and second parameter is an array containing the points on the y-axis. For example, to draw a line in a diagram from position (1, 3) to position (8, 10), we write.

```
import matplotlib.pyplot as plt
import numpy as np
xpoints = np.array([1, 8])
ypoints = np.array([3, 10])
plt.plot(xpoints, ypoints)
plt.show()
```

To plot only the markers, you can use shortcut string notation parameter 'o', which means 'rings'. For example,

```
plt.plot(xpoints, ypoints, 'o')
```

To plot as many points as we like, we use same number of points in both x-axis and y-axis. For example,

```
xpoints = np.array([1, 2, 6, 8])
ypoints = np.array([3, 8, 1, 10])
plt.plot(xpoints, ypoints)
```

If we do not specify the points in the x-axis, they will get the default values 0, 1, 2, 3, etc. depending on the length of the y-points. For example,

```
ypoints = np.array([3, 8, 1, 10, 5, 7])
plt.plot(ypoints)
```

3.1. Markers

You can use the **marker** argument to emphasize each point with a specified marker. For example, to mark each point with circle, we write

```
plt.plot(xpoints, ypoints, marker = 'o')
```

Some commonly used markers are:

'o' Circle	'*' Star	'.' Point	',' Pixel
'x' X	'X' X (filled)	'+' Plus	'P' Plus (filled)
's' Square	'D' Diamond	'd' Diamond (thin)	'p' Pentagon
'v' Triangle Down	'^' Triangle Up	'<' Triangle Left	'>' Triangle Right

You can use also use the shortcut string notation parameter to specify the marker. This parameter is also called **fmt**, and is written with the syntax **markerlinecolor**. For example,

```
plt.plot(xpoints, ypoints, 'o--r')
```

The marker value can be anything from the table above. The line value and color value can be one of the following given in the table below.

Line reference	**Color reference**
'-' Solid line	'r' Red
':' Dotted line	'g' Green
'--' Dashed line	'b' Blue
'-.' Dashed/dotted line	'c' Cyan
	'm' Magenta
	'y' Yellow
	'k' Black
	'w' White

We can use the argument **markersize** or the shorter version, **ms** to set the size of the markers. For example,

```
plt.plot(xpoints, ypoints, 'o--r', ms = 20)
```

We can use the argument **markeredgecolor** or the shorter **mec** to set the color of the edge of the markers. For example,

```
plt.plot(xpoints, ypoints, 'o--r', ms = 20, mec = 'b')
```

We can use the argument **markerfacecolor** or the shorter **mfc** to set the color inside the edge of the markers. For example,

```
plt.plot(xpoints, ypoints, 'o--r', ms = 20, mec = 'b', mfc = 'y')
```

Remember: We can also use hexadecimal color values (such as *'#4CAF50'*) and color names (such as *'hotpink'*) to represent different colors.

3.2. Line

We can use the argument **linestyle**, or shorter **ls**, to change the style of the plotted line. The different line styles are:

'solid' or '-' (Default)	'dotted' or ':'	'dashed' or '--'	'dashdot' or '-.'	'None' or ' '

We can use the argument **color** or the shorter **c** to set the color of the line. We can also use the argument **linewidth** or the shorter **lw** to change the width of the line.

We can plot as many lines as you like by simply adding more *plot()* functions. For example,

```
x1 = np.array([0, 1, 2, 3])
y1 = np.array([3, 8, 1, 10])
x2 = np.array([0, 1, 2, 3])
y2 = np.array([6, 2, 7, 11])
plt.plot(x1, y1)
plt.plot(x2, y2)
plt.show()
```

We can also plot many lines by adding the points for the x- and y-axis for each line in the same *plot()* function. For example,

```
plt.plot(x1, y1, x2, y2)
```

3.3. Labels

We can use **xlabel()** and **ylabel()** functions to set a label for the x- and y-axis. And, with **title()** function you can set a title for the plot. We can use the **fontdict** parameter in *xlabel()*, *ylabel()*, and *title()* to set font properties for the title and labels. For example,

```
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])
font1 = {'family':'serif','color':'blue','size':20}
font2 = {'family':'serif','color':'darkred','size':15}
plt.title("Sports Watch Data", fontdict = font1)
plt.xlabel("Average Pulse", fontdict = font2)
plt.ylabel("Calorie Burnage", fontdict = font2)
plt.plot(x, y)
plt.show()
```

The **loc** parameter in *title()* is used to position the title in '**left**', '**right**', and '**center**'. Default value for loc is 'center'. For example,

```
plt.title("Sports Watch Data", fontdict = font1, loc = 'left')
```

3.4. Grid Lines

We can use **grid()** function to add grid lines to the plot. We can use **axis** parameter in the *grid()* function to specify which grid lines to display. Legal values for axis are 'x', 'y', and 'both'. Default value is 'both'. For example,

```
plt.grid()
```

We can also set line properties of the grid using *plt.grid(color = 'color', linestyle = 'linestyle', linewidth = number).*

3.5. Subplot

We can use **subplot()** function to draw multiple plots in one figure. The *subplot()* function takes three arguments that describes the layout of the figure. The layout is organized in rows and columns, which are represented by the first and second argument. The third argument represents the index of the current plot. For example,

```
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(2, 1, 1)
plt.plot(x,y)
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(2, 1, 2)
plt.plot(x,y)
```

We can add a title to each plot with the **title()** function. We can also use **subplots_adjust()** method to change the space between subplots. For example,

```
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(2, 1, 1)
plt.plot(x,y)
plt.title("Plot 1")
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(2, 1, 2)
plt.plot(x,y)
plt.title("Plot 2")
plt.subplots_adjust(left = 0.1,
                    bottom = 0.1,
                    right = 0.9,
                    top = 0.9,
                    wspace = 0.4,
                    hspace = 0.4)
plt.show()
```

We use **suptitle()** function to add a title to the entire figure. For example, *plt.suptitle("This is super title").*

3.6. Scatter Plots

We use **scatter()** function to draw a scatter plot. This function draws one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis. For example,

```
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y)
```

When two plots are plotted, the plots will be plotted with two different colors, by default blue and orange. For example,

```
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y)
x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y)
```

We can also set our own color for each scatter plot with the **color** or the **c** argument. For example,

```
plt.scatter(x, y, color = 'hotpink')
```

We can even set a specific color for each dot by using an array of colors as value for the **c** argument. For example,

```
x = np.array([5,7,8,7,2,17])
y = np.array([99,86,87,88,111,86])
cols = np.array(["red","green","blue","yellow","pink","black"])
plt.scatter(x, y, c = cols)
```

We can change the size of the dots with the **s** argument. We can adjust the transparency of the dots with the **alpha** argument. For example,

```
x = np.array([5,7,8,7,2,17])
y = np.array([99,86,87,88,111,86])
sizes = np.array([20,50,100,200,500,1000])
plt.scatter(x, y, s = sizes, alpha = 0.3)
```

3.7. Bar Graphs

Bar graphs are the pictorial representation of data (generally grouped), in the form of vertical or horizontal rectangular bars, where the length of bars are proportional to the measure of data. To display bar graphs, we use **bar()** function. We can also use **color** argument to color bars. For example,

```
x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])
plt.bar(x,y, color = "blue")
```

If you want the bars to be displayed horizontally instead of vertically, use the **barh()** function. For example,

```
plt.barh(x, y)
```

The *bar()* function takes the argument **width** to set the width of the bars. The default width value is 0.8. For horizontal bars, we use **height** instead of *width*. For example,
plt.bar(x,y, color = "blue", width = 0.4)

3.8. Histogram

It is a graph showing the number of observations within each given interval. We can use **hist()** function to produce a histogram. For example,

```
import matplotlib.pyplot as plt
x = [1,1,2,3,3,5,7,8,9,10,
    10,11,11,13,13,15,16,17,18,18,
    18,19,20,21,21,23,24,24,25,25,
    25,25,26,26,26,27,27,27,27,27,
    29,30,30,31,33,34,34,34,35,36,
    36,37,37,38,38,39,40,41,41,42,
    43,44,45,45,46,47,48,48,49,50,
    51,52,53,54,55,55,56,57,58,60,
    61,63,64,65,66,68,70,71,72,74,
    75,77,81,83,84,87,89,90,90,91
  ]
plt.hist(x, bins = 5)
plt.show()
```

3.9. Pie Charts

A pie chart is a type of graph that represents the data in the circular graph. The slices of pie show the relative size of the data. We use the **pie()** function to draw pie charts. For example,

```
import matplotlib.pyplot as plt
x = [35, 25, 25, 15]
plt.pie(x)
plt.show()
```

By default, the plotting of the first wedge starts from the x-axis and move counterclockwise.

We use **label** parameter to add labels to the pie chart. The label parameter must be an array with one label for each wedge. For example,

```
x = [35, 25, 25, 15]
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(x, labels = mylabels)
```

We can change the start angle by specifying a **startangle** parameter. The startangle parameter is defined with an angle in degrees, default angle is 0 (x-axis). For example,

```
plt.pie(x, labels = mylabels, startangle = 90)
```

If we want the wedges stand out, we use explode parameter. The explode parameter must be an array with one value for each wedge. Each value represents how far from the center each wedge is displayed. For example,

```
x = [35, 25, 25, 15]
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
myexplode = [0.2, 0, 0, 0]
plt.pie(x, labels = mylabels, startangle = 90, explode = myexplode)
```

To add a shadow to the pie chart, we set **shadow** parameter to True. For example,

```
plt.pie(x, labels = mylabels, startangle = 90, explode = myexplode, shadow = True)
```

We can also set color of each wedge with the colors parameter. The colors parameter, if specified, must be an array with one value for each wedge. For example,

```
x = [35, 25, 25, 15]
mycolors = ["black", "hotpink", "b", "#4CAF50"]
plt.pie(x, colors = mycolors)
```

We use **legend()** function to add list of explanation for each wedge. To add a header to the legend, add the **title** parameter to the legend function. For example,

```
x = [35, 25, 25, 15]
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(x)
plt.legend(title = "Four Fruits", labels = mylabels)
```