Welcome
To
Thinking Programming

Workshop

Today's FDP Session Title

Cleansing Data 

Data Analytics 

Data Visualization 

Contact Me

	https://www.gowrishankarnath.com
	gowrishankarnath@acm.org
	@g_s_nath
	https://in.linkedin.com/in/gowrishankarnath
	https://github.com/gowrishankarnath

---

Prerequisite for this FDP Session

Familiarity with the basics of Python Programming language including

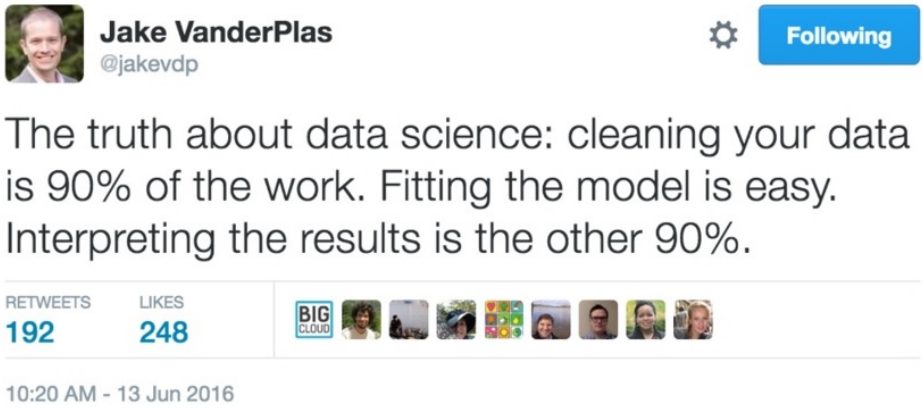
- Functions
- Strings
- Lists
- Dictionaries
- Tuples
- Sets
- Classes and Objects

Flow of this Session

- Cursory look at **NumPy** upon which Pandas is built
- We shall discuss the **nitty-gritty** aspects of **Pandas** for Data Analytics
- **Matplotlib** library will be used for meaningful and informative Data Visualization

---

This FDP Session's Mission



Data Cleansing

Data cleansing or data cleaning is the process of detecting and correcting (or removing) corrupt or inaccurate records from a record set, table, or database and refers to identifying incomplete, incorrect, inaccurate or irrelevant parts of the data and then replacing, modifying, or deleting the dirty or coarse data.

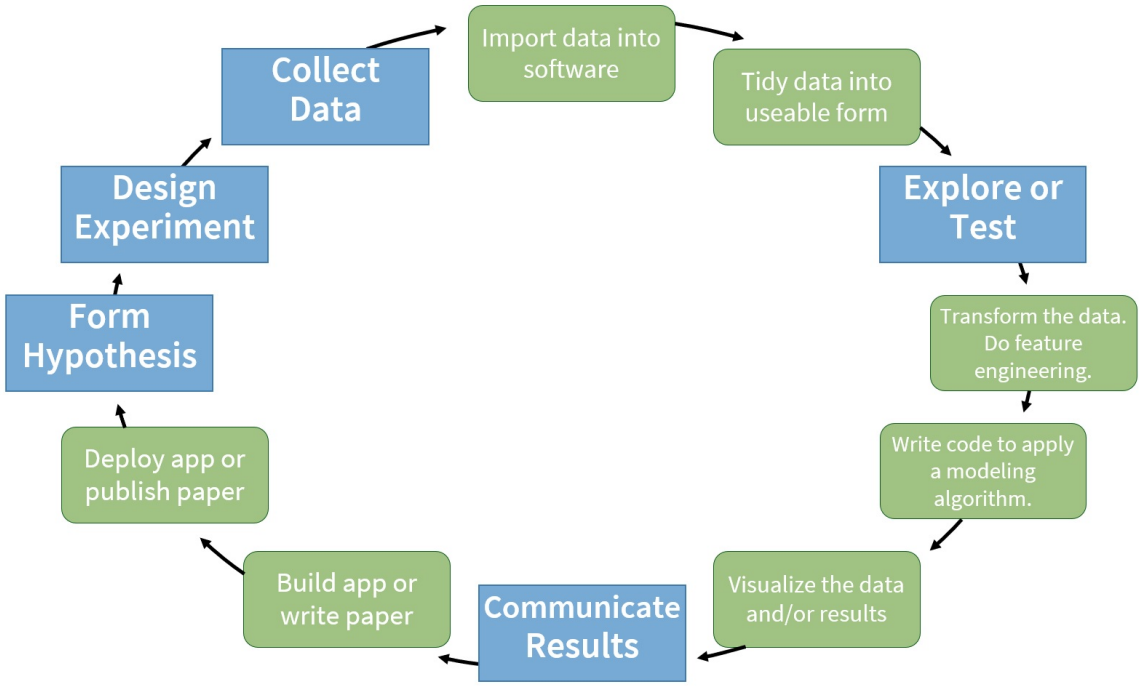
Data cleansing may be performed interactively with data wrangling tools, or as batch processing through scripting.

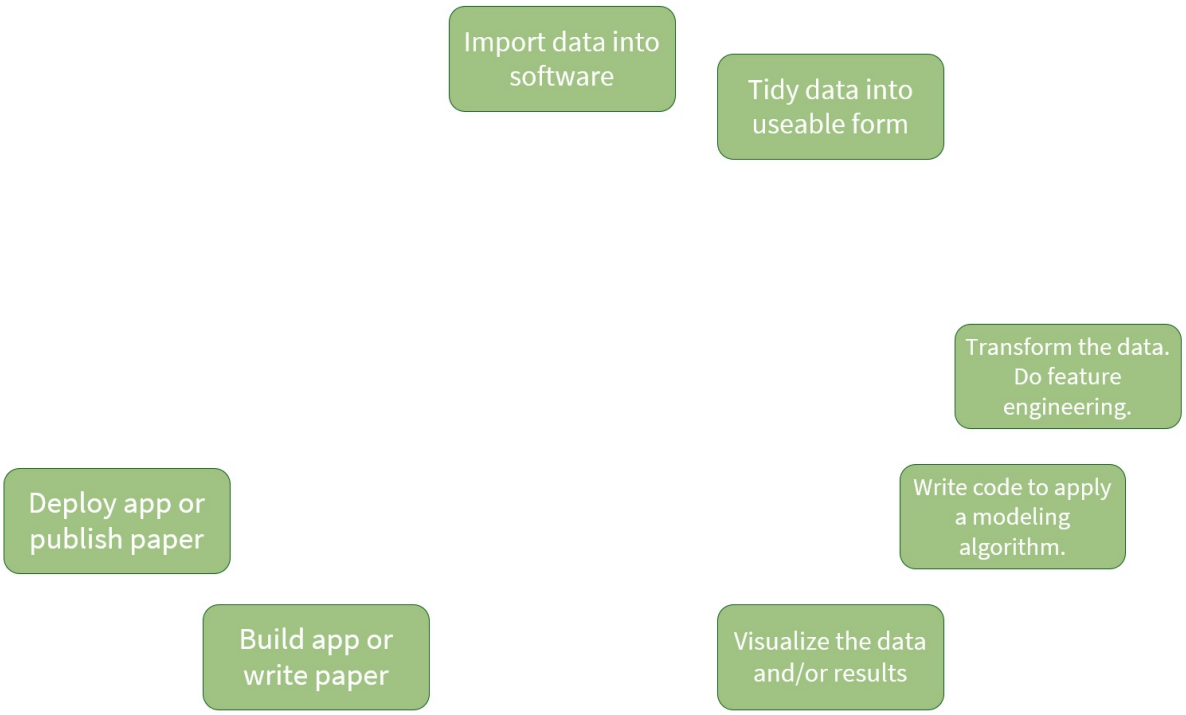
After cleansing, a data set should be consistent.

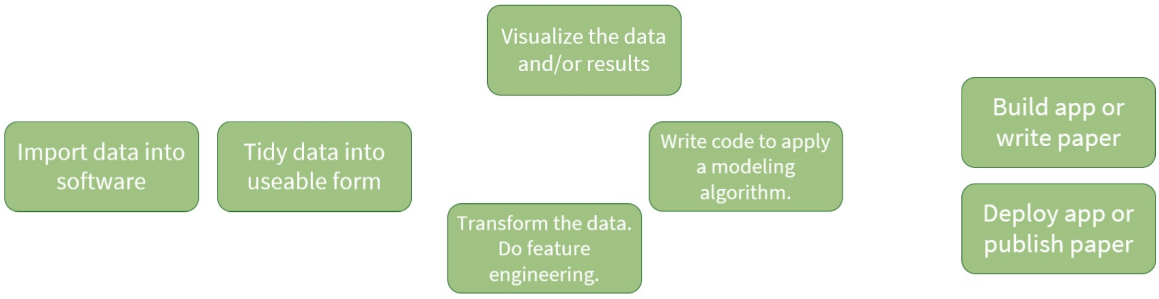
The inconsistencies detected or removed may have been originally caused by user entry errors, by corruption in transmission or storage, or by different data dictionary definitions of similar entities in different stores.

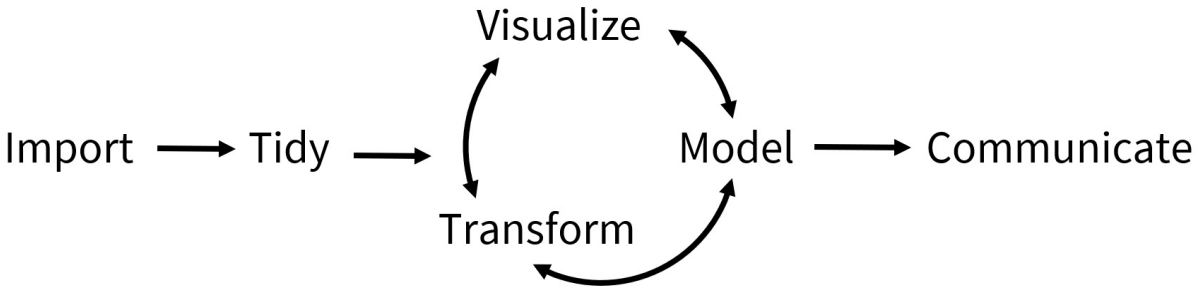
Data Analytics

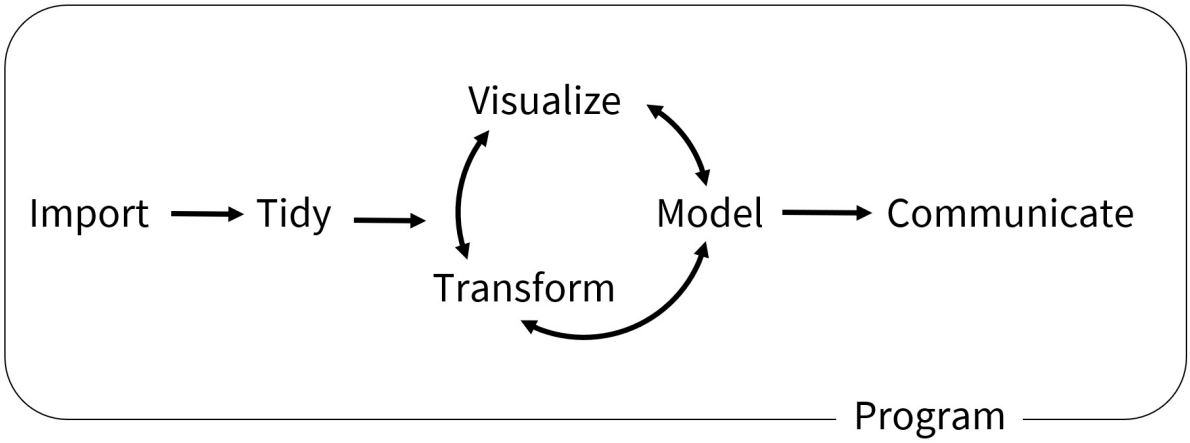
Data analytics refers to the process of compiling the data and analyze the data using technical tools in order to present findings to management to help inform business decision making.


Collect Data
Import data into software
Tidy data into useable form
Explore or Test
Transform the data. Do feature engineering.
Write code to apply a modeling algorithm.
Visualize the data and/or results
Communicate Results
Build app or write paper
Deploy app or publish paper
Form Hypothesis
Design Experiment

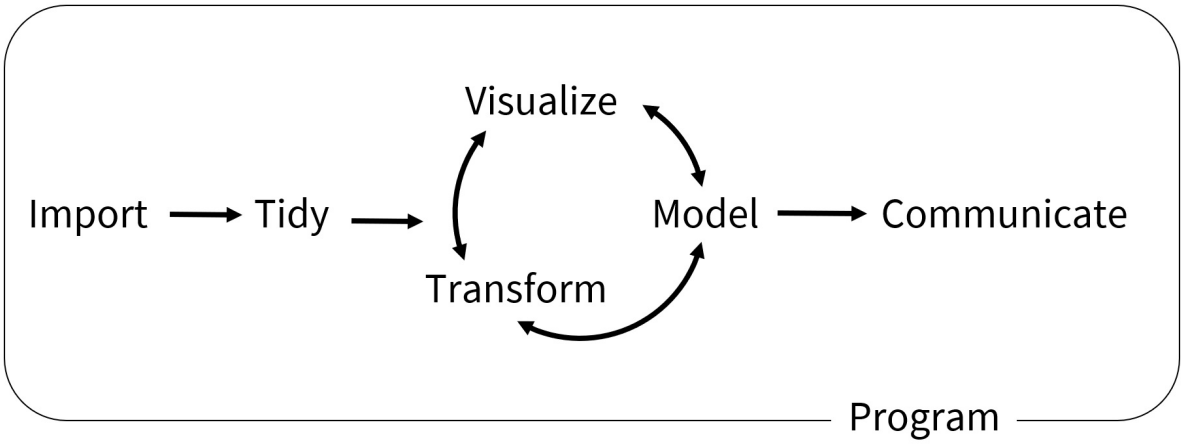

Import data into software
Tidy data into useable form
Transform the data. Do feature engineering.
Deploy app or publish paper
Write code to apply a modeling algorithm.
Build app or write paper
Visualize the data and/or results



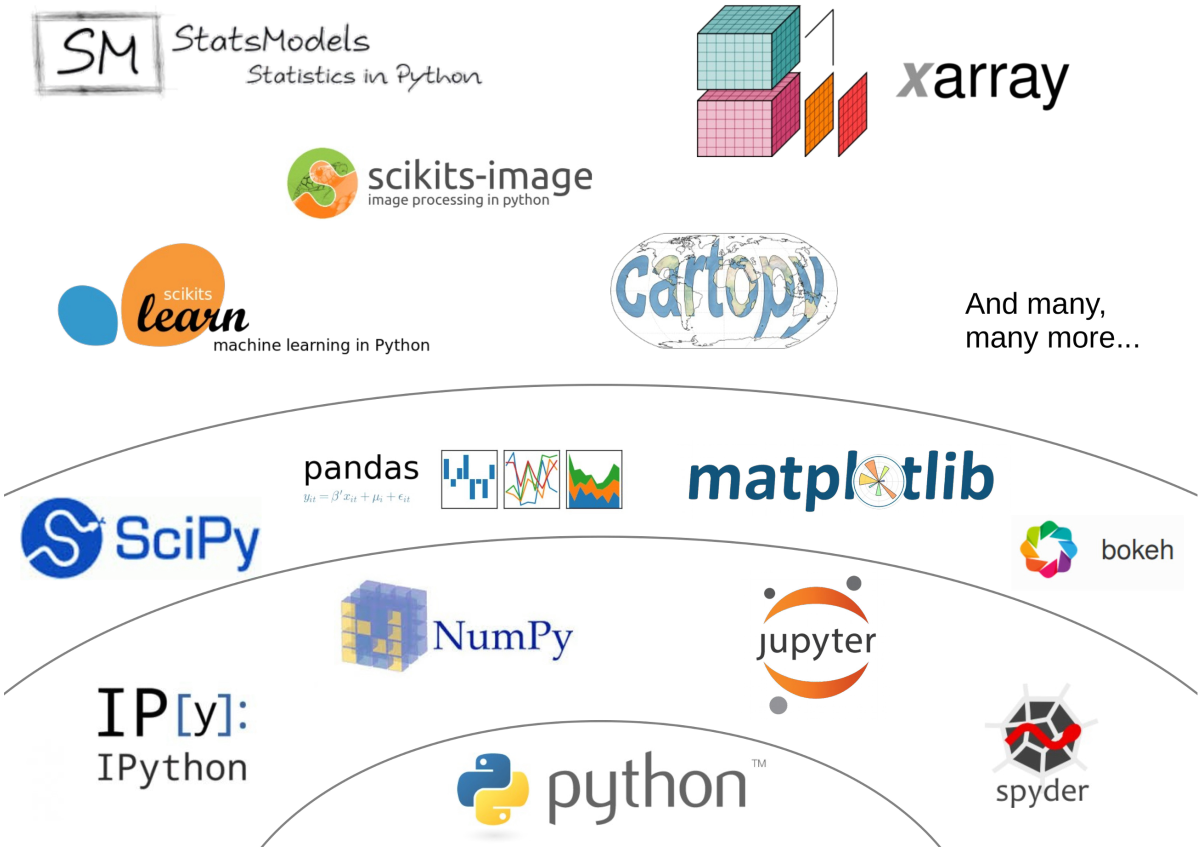




Data Analytics



Python Data Science Stack



Definition and Usage of map() function

The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

Syntax

> **map(function, iterables)**

function --> The function to execute for each item
iterable --> A sequence, collection or an iterator object. You can send as many iterables as you like, just make sure the function has one parameter for each iterable

In [ ]:
```
def myfunc(n):
    return len(n)


x = map(myfunc, ('apple', 'banana', 'cherry'))
print(x)
print(type(x))
# convert the map into a list, for readability:
print(list(x))
```

```
<map object at 0x0000026EF928F520>
<class 'map'>
[5, 6, 6]
```

In [ ]:
```
def myfunc(a, b):
    return a + b


x = map(myfunc, ('apple', 'banana', 'cherry'),
        ('orange', 'lemon', 'pineapple'))
# convert the map into a list, for readability:
print(list(x))
```

```
['appleorange', 'bananalemon', 'cherrypineapple']
```

Definition and Usage of filter() function

The filter() function returns an iterator where the items are filtered through a function to test if the item is accepted or not.

Syntax

> **filter(function, iterable)**

function --> A Function to be run for each item in the iterable
iterable --> The iterable to be filtered

In [ ]:
```
ages = [5, 12, 17, 18, 24, 32]


def myFunc(x):
    if x < 18:
        return False
    else:
        return True


adults = filter(myFunc, ages)

for x in adults:
    print(x)
```

```
18
24
32
```

What are lambda functions in Python?

In Python, an anonymous function is a function that is defined without a name.

While normal functions are defined using the `def` keyword in Python, anonymous functions are defined using the `lambda` keyword.

Hence, anonymous functions are also called **lambda** functions

How to use lambda Functions in Python?

A lambda function in python has the following syntax.

> **lambda arguments: expression**

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

In [ ]:
```
# Here is an example of lambda function that doubles the input value.

# Program to show the use of lambda functions
double = lambda x: x * 2

print(double(5))

# In the above program, lambda x: x * 2 is the lambda function.
# Here x is the argument and x * 2 is the expression that gets evaluated and returne
```

```
10
```

Example use with filter()

Here is an example use of filter() function to filter out only even numbers from a list.

In [ ]:
```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x % 2 == 0), my_list))

print(new_list)
```

```
[4, 6, 8, 12]
```

Example use with map()

Here is an example use of map() function to double all the items in a list.

In [ ]:
```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)
```

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

---

reduce() in Python

The syntax for reduce() is

> **reduce(fun, seq)**

reduce() function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along. This function is defined in “functools” module.

In [ ]:
```
# python code to demonstrate working of reduce()
```

```
# importing functools for reduce()
import functools

# initializing list
lis = [1, 3, 5, 6, 2]

# using reduce to compute sum of list
print("The sum of the list elements is : ")
print(functools.reduce(lambda a, b: a + b, lis))

# using reduce to compute maximum element from list
print("The maximum element of the list is : ")
print(functools.reduce(lambda a, b: a if a > b else b, lis))
```

```
The sum of the list elements is :
17
The maximum element of the list is :
6
```

- At first step, first two elements of sequence are picked and the result is obtained.
- Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.
- This process continues till no more elements are left in the container.
- The final returned result is returned and printed on console.

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

The Syntax

> **newlist = [expression for item in iterable if condition == True]**

The return value is a new list, leaving the old list unchanged.

Condition

The condition is like a filter that only accepts the items that valuate to True.

Iterable

The iterable can be any iterable object, like a list, tuple, set etc.

Expression

The expression is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list.

In [ ]:

```
# Based on a list of fruits, you want a new list, containing only the fruits with th

# Without list comprehension you will have to write a for statement with a condition

fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
```

```
    if "a" in x:
        newlist.append(x)

print(newlist)
```

```
['apple', 'banana', 'mango']
```

In [ ]:
```
# With list comprehension you can do all that with only one line of code:

fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

```
['apple', 'banana', 'mango']
```

Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create dictionaries.

The minimal syntax for dictionary comprehension is:

> **dictionary = {key: value for vars in iterable}**

In [ ]:
```
# Example 1: Dictionary Comprehension
# Consider the following code:

square_dict = dict()
for num in range(1, 11):
    square_dict[num] = num * num
print(square_dict)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

In [ ]:
```
# Now, let's create the dictionary in the above program using dictionary comprehensi

# dictionary comprehension example
square_dict = {num: num * num for num in range(1, 11)}
print(square_dict)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

In both programs, we have created a dictionary square_dict with number-square key/value pair.

However, using dictionary comprehension allowed us to create a dictionary in a single line.

In [ ]:
```
#item price in dollars
old_price = {'milk': 1.02, 'coffee': 2.5, 'bread': 2.5}

dollar_to_pound = 0.76
new_price = {item: value*dollar_to_pound for (item, value) in old_price.items()}
print(new_price)
```

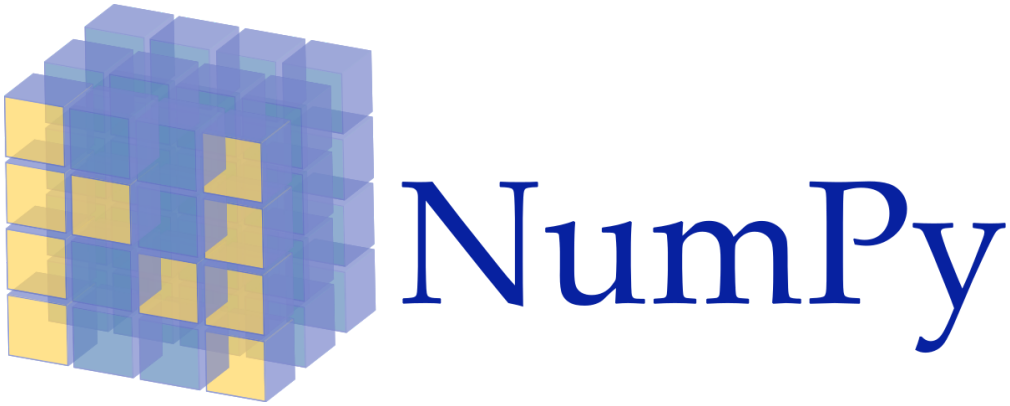
```
{'milk': 0.7752, 'coffee': 1.9, 'bread': 1.9}
```

In [ ]:
```
original_dict = {'jack': 38, 'michael': 48, 'guido': 57, 'john': 33}
```

```
even_dict = {k: v for (k, v) in original_dict.items() if v % 2 == 0}
print(even_dict)
```

```
{'jack': 38, 'michael': 48}
```

Introduction to NumPy



NumPy is, just like SciPy, Scikit-Learn, Pandas, etc. one of the packages that you just can't miss when you're learning data science, mainly because this library provides you with an array data structure that holds some benefits over Python lists, such as: being more compact, faster access in reading and writing items, being more convenient and more efficient.

NumPy, which stands for Numerical Python is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

NumPy History

- Numeric, the ancestor of NumPy, was developed by Jim Hugunin.
- Another package Numarray was also developed, having some additional functionalities.
- In 2005, Travis Oliphant created NumPy package by incorporating the features of Numarray into Numeric package.
- There are many contributors to this open source project.

---

ndarray

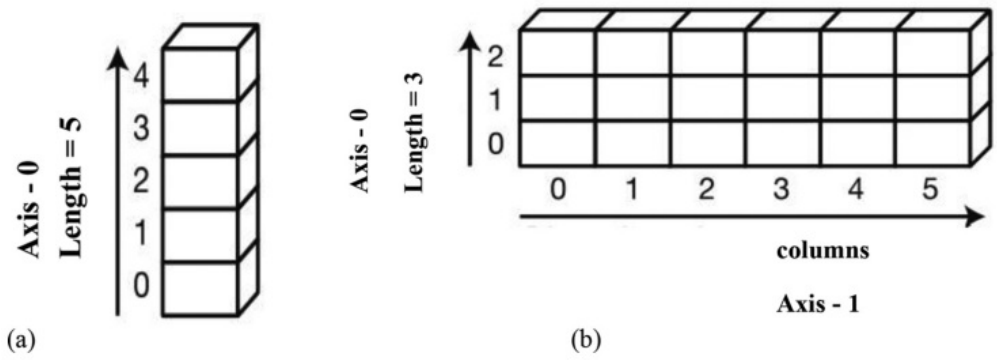
- NumPy's main object is the homogeneous multidimensional array.
- An array is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers and represented by a single variable.
- NumPy's array class is called **ndarray**. It is also known by the alias **array**.

- In NumPy arrays, the individual data items are called as **elements**. All the elements of an array should be of the same type.
- Arrays can be made up of any number of dimensions. In NumPy, dimensions are called **axes**.

Axes

Each **axes** (**or dimension**) of an array has a length which is the total number of elements in that direction.

The **size** of an array is the total number of elements contained in an array in all the dimension. The **size** of NumPy arrays are fixed; once created it cannot be changed again.



The above FIGURE shows the axes (or dimensions) and lengths of two example arrays

(a) is a one-dimensional array and

- That axis has five elements in it, so we say it has a length of five.
- All the rows are indicated by Axis-0

(b) is a two-dimensional array.

- A two-dimensional array is made up of rows and columns.
- All the rows are indicated by Axis-0 and all the columns are indicated by Axis-1.
- In a two-dimensional array, Axis-0 has three elements in it, so its length is three and Axis-1 has six elements in it, so its length is six.

Notice that for each axis, the indexes range from 0 to length – 1. Array indexes are 0-based. That is, if the length of a dimension is n, the index values range from 0 to n – 1.

Using NumPy Code

In order to use NumPy in your program, you need to import NumPy library. For example,

import numpy as np

numpy is usually renamed as np.

NumPy Arrays Creation Using array() Function

- You can create a NumPy array from a regular Python list or tuple using the np.array() function.
- The type of the resulting array is deduced from the type of the elements.
- Each element in ndarray is an object of data-type object (called **dtype**).

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

In [ ]:
```
int_number_array = np.array([1,2,3,4])
print(f"Int number array {int_number_array}")
```

```
Int number array [1 2 3 4]
```

In [ ]:
```
print(f"Type is {type(int_number_array)}")
```

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

In [ ]:
```
print(f"Dtype is {int_number_array.dtype}")
```

```
Dtype is int32
```

In [ ]:
```
float_number_array = np.array([9.1, 8.1, 8.8, 3.0])
print(f"Float number Dtype is {float_number_array.dtype}")
```

```
Float number Dtype is float64
```

In [ ]:
```
two_dimensional_array_list = np.array([[1,2,3], [4,5,6]])
print(f"Type is {type(two_dimensional_array_list)}")
print(f"Two Dimensional Array List is \n {two_dimensional_array_list}")
```

```
Type is <class 'numpy.ndarray'>
Two Dimensional Array List is
 [[1 2 3]
 [4 5 6]]
```

In [ ]:
```
two_dimensional_array_tuple = np.array(((1,2,3), (4,5,6)))
print(f"Two Dimensional Array tuple is \n {two_dimensional_array_tuple}")
```

```
Two Dimensional Array tuple is
 [[1 2 3]
 [4 5 6]]
```

In [ ]:
```
array_dtype = np.array([1,2,3,4], dtype = np.float64)
print(f"Array Display {array_dtype}")
print(f"Dtype is {array_dtype.dtype}")
```

```
Array Display [1. 2. 3. 4.]
Dtype is float64
```

- NumPy provides a large set of numeric datatypes that you can use to construct arrays.

- NumPy tries to guess a **datatype** when you create an array, but functions that construct arrays also usually include an optional argument to explicitly specify the datatype.
- In the above code, the type of int_number_array variable is dtype('int32')
- You can also explicitly specify the data type of array by assigning type values like np.float64, np.int32, and others to a dtype attribute and pass it as a second argument to np.array() function.

Array Attributes

The various array attributes of NumPy are

1. **ndarray.ndim**
   Gives the number of axes or dimensions in the array
2. **ndarray.shape**
   Gives the dimensions of the array. For an array with n rows and m columns, shape will be a tuple of integers (n, m).
3. **ndarray.size**
   Gives the total number of elements of the array.
4. **ndarray.dtype**
   Gives an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally, NumPy provides its own types like np.int32, np.int16, np.float64, and others.
5. **ndarray.itemsize**
   Gives the size of each element of the array in bytes.

In [ ]:
```
array_attributes = np.array([[10, 20, 30], [14, 12, 16]])
print("array_attributes")
print("\n")
print(f"ndim is \n {array_attributes.ndim}")
```

```
array_attributes


ndim is
 2
```

In [ ]:
```
print(f"shape is \n {array_attributes.shape}")
```

```
shape is
 (2, 3)
```

In [ ]:
```
print(f"size is \n {array_attributes.size}")
```

```
size is
 6
```

In [ ]:
```
print(f'dtype is \n {array_attributes.dtype}')
```

```
dtype is
 int32
```

In [ ]:

```
print(f'itemsize is \n {array_attributes.itemsize}')
```

```
itemsize is
 4
```

Array Creation Functions

NumPy provides many functions to create arrays

In [ ]:
```
print(f"Zeros array \n {np.zeros((2,3))}")
```

```
Zeros array
 [[0. 0. 0.]
 [0. 0. 0.]]
```

In [ ]:
```
print(f"Ones array \n {np.ones((3,4))}")
```

```
Ones array
 [[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

In [ ]:
```
print(f"Empty array \n {np.empty((2,3))}")
```

```
Empty array
 [[0. 0. 0.]
 [0. 0. 0.]]
```

In [ ]:
```
print(f"Full array \n {np.full((3,3), 2)}")
```

```
Full array
 [[2 2 2]
 [2 2 2]
 [2 2 2]]
```

In [ ]:
```
print(f"Eye array \n {np.eye(3,3)}")
```

```
Eye array
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

In [ ]:
```
print(f"Random array \n {np.random.random((2,2))}")
```

```
Random array
 [[0.85902212 0.16895704]
 [0.92073058 0.95943957]]
```

In [ ]:
```
# The third arguent is the size
print(f"Random array integer \n {np.random.randint(10, 30, 6)}")
```

```
Random array integer
 [17 20 25 25 28 24]
```

In [ ]:
```
print(f"Arange array \n {np.arange(10, 30, 5)}")
```

```
Arange array
 [10 15 20 25]
```

In [ ]:
```
print(f"Arange array \n {np.arange(0, 2, 0.3)}")
```

```
Arange array
 [0.  0.3 0.6 0.9 1.2 1.5 1.8]
```

In [ ]:
```
print(f"linspace array \n {np.linspace(0, 2, 9)}")
```

```
linspace array
 [0.   0.25 0.5  0.75 1.   1.25 1.5  1.75 2.  ]
```

np.arange()

The syntax for arange() is,

> *np.arange([start,]stop, [step,][dtype=None])*

Returns evenly spaced values within a given interval where start (a number and optional) is the start of interval and its default value is zero, stop (a number) is the end of interval, and step (a number and is optional) is the spacing between the values and dtype is the type of output array.

np.linspace()

The syntax for linspace is,

> *numpy.linspace(start, stop, num=50, dtype=None)*

Returns evenly spaced numbers over a specified interval where start is the starting value of the sequence, stop is the end value of the sequence, and num (an integer and optional) is the number of samples to generate. Default is 50.
Must be non-negative. The optional dtype is the type of the output array.

Integer Indexing, Array Indexing, Boolean Array Indexing, Slicing and Iterating in Arrays

Slicing: Similar to Python lists, NumPy arrays can be sliced.

One-dimensional arrays can be indexed, sliced, and iterated over, much like lists and other Python sequences.

Since arrays may be multidimensional, you must specify a slice for each dimension of the array and they should be separated by a comma.

In [ ]:
```
a = np.arange(5)
print(a)
```

```
[0 1 2 3 4]
```

In [ ]:
```
print(a[2])
```

```
2
```

In [ ]:
```
print(a[2:4])
```

```
[2 3]
```

In [ ]:
```
a[:4:2] = -999
```

In [ ]:
```
print(a)
```

```
[-999    1 -999    3    4]
```

In [ ]:
```
print(a[::-1])
```

```
[   4    3 -999    1 -999]
```

In [ ]:
```
print("Display each element")
for each_element in a:
    print(each_element)
```

```
Display each element
-999
1
-999
3
4
```

For multi-dimensional arrays you can specify an index or slice per axis. For example,

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

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
8
```

In [ ]:
```
print(a[:2, 1:3])
```

```
[[2 3]
 [6 7]]
```

In [ ]:
```
lower_axes = a[1, :]
print(lower_axes)
```

```
[5 6 7 8]
```

In [ ]:
```
print(lower_axes.ndim)
```

```
1
```

In [ ]:
```
same_axes = a[1:2, :]
print(same_axes)
print(same_axes.ndim)
```

```
[[5 6 7 8]]
2
```

In [ ]:
```
print(a[:, 1])
```

```
[ 2  6 10]
```

In [ ]:
```
print(a[:, 1:2])
```

```
[[ 2]
 [ 6]
 [10]]
```

In [ ]:
```
for row in a:
    print(row)
```

```
[1 2 3 4]
[5 6 7 8]
[ 9 10 11 12]
```

In [ ]:
```
for each_element in a.flat:
    print(each_element)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
```

If one wants to perform an operation on each element in the array, one can use the **flat** attribute, which is an iterator over all the elements of the array.

In [ ]:
```
import numpy as np
a = np.array([[11, 12], [13, 14], [15, 16]])
print(a)
print("Values greater than 13")
print(a[a > 13])
```

```
[[11 12]
 [13 14]
 [15 16]]
Values greater than 13
[14 15 16]
```

Boolean array indexing lets you select the elements of an array that satisfy some condition.

With Boolean array indexing, you explicitly choose which items in the array you want and which ones you do not want. With Boolean array indexing of a[a > 13], elements greater than 13 in the array a are displayed as a one-dimensional array

---

Basic Arithmetic Operations on NumPy Arrays

Basic mathematical functions perform element-wise operation on arrays and are available both as operator overloads and as functions in the NumPy module.

In [ ]:
```
a = np.array( [20, 30, 40, 50] )
b = np.arange(4)
print(b)
```

```
[0 1 2 3]
```

In [ ]:
```
print(a + b)
print(np.add(a, b))
```

```
[20 31 42 53]
[20 31 42 53]
```

In [ ]:
```
print(a - b)
print(np.subtract(a, b))
```

```
[20 29 38 47]
[20 29 38 47]
```

In [ ]:
```
A = np.array( [[1, 1], [6, 1]] )
B = np.array( [[2, 8], [3, 4]] )
print(A * B)
print(np.multiply(A, B))
```

```
[[ 2  8]
 [18  4]]
[[ 2  8]
 [18  4]]
```

In [ ]:
```
print(A / B)
print(np.divide(A, B))
```

```
[[0.5   0.125]
 [2.    0.25 ]]
[[0.5   0.125]
 [2.    0.25 ]]
```

In [ ]:
```
print(np.dot(A, B))
```

```
[[ 5 12]
 [15 52]]
```

Matrix product is carried out here

In [ ]:
```
print(B**2)
```

```
[[ 4 64]
 [ 9 16]]
```

Mathematical Functions in NumPy

Various mathematical functions are supported in NumPy. A few frequently used mathematical functions are shown below.

In [ ]:
```
a = np.array( [20, 30, 40, 50] )
print(np.sin(a))
```

```
print(np.cos(a))
print(np.tan(a))
```

```
[ 0.91294525 -0.98803162  0.74511316 -0.26237485]
[ 0.40808206  0.15425145 -0.66693806  0.96496603]
[ 2.23716094 -6.4053312  -1.11721493 -0.27190061]
```

In [ ]:
```
a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
print(np.floor(a))
```

```
[-2. -2. -1.  0.  1.  1.  2.]
```

The floor() function having syntax as floor(x), returns the largest integer value less than or equal to x, element-wise.

In [ ]:
```
print(np.ceil(a))
```

```
[-1. -1. -0.  1.  2.  2.  2.]
```

The ceil() function having a syntax as ceil(x), returns the smallest integer value greater than or equal to x, element-wise

In [ ]:
```
print(np.sqrt([1,4,9]))
```

```
[1. 2. 3.]
```

In [ ]:
```
print(np.maximum([2, 3, 4], [1, 5, 2]))
```

```
[2 5 4]
```

In [ ]:
```
print(np.minimum([2, 3, 4], [1, 5, 2]))
```

```
[1 3 2]
```

In [ ]:
```
print(np.sum([0.5, 1.5]))
```

```
2.0
```

In [ ]:
```
print(np.sum([[0, 1], [0, 5]], axis=0))
```

```
[0 6]
```

In [ ]:
```
print(np.sum([[0, 1], [0, 5]], axis=1))
```

```
[1 5]
```

Changing the Shape of an Array

You can change the shape of an array. For example,

In [ ]:
```
a = np.floor(10*np.random.random((3, 4)))
print(a)
```

```
[[2. 4. 9. 9.]
 [5. 8. 0. 4.]
```

```
 [1. 5. 4. 1.]]
```

In [ ]:
```
print(a.shape)
```

```
(3, 4)
```

In [ ]:
```
print(a.ravel())
```

```
[2. 4. 9. 9. 5. 8. 0. 4. 1. 5. 4. 1.]
```

In [ ]:
```
print(a.reshape(6,2))
```

```
[[2. 4.]
 [9. 9.]
 [5. 8.]
 [0. 4.]
 [1. 5.]
 [4. 1.]]
```

Both **ravel()** and **reshape()** return a modified array but do not change the original array.

The function **ravel()** returns a flattened array, such as a one-dimensional array, containing the elements of the input.
The function **reshape()** gives a new shape to an array without changing its data.

Stacking and Splitting of Arrays

You can stack several arrays together or split an array to several arrays. For example,

Several arrays can be stacked together along different dimensions using **vstack()** and **hstack()** functions.

With **vstack()** and **hstack()** functions you are stacking the arrays a and b together in row-wise and column-wise fashion.

In [ ]:
```
a = np.array([[3, 1], [8, 7]])
b = np.array([[2, 4], [4, 8]])
print("Vertical Stack")
print(np.vstack((a, b)))
print("\n")
print("Horizontal Stack")
print(np.hstack((a, b)))
```

```
Vertical Stack
[[3 1]
 [8 7]
 [2 4]
 [4 8]]


Horizontal Stack
[[3 1 2 4]
 [8 7 4 8]]
```

The **split()** function divides the array into subarrays along a specified axis. The function takes three parameters.

> **numpy.split(ary, indices_or_sections, axis)**

ary
Input array to be split

indices_or_sections

Can be an integer, indicating the number of equal sized subarrays to be created from the input array. If this parameter is a 1-D array, the entries indicate the points at which a new subarray is to be created.

axis
Default is 0

In [ ]:
```
import numpy as np
a = np.arange(9)

print('First array:')
print(a)
print('\n')

print('Split the array in 3 equal-sized subarrays:')
b = np.split(a, 3)
print(b)
print('\n')

print('Split the array at positions indicated in 1-D array:')
b = np.split(a, [4, 7])
print(b)
```

```
First array:
[0 1 2 3 4 5 6 7 8]


Split the array in 3 equal-sized subarrays:
[array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]


Split the array at positions indicated in 1-D array:
[array([0, 1, 2, 3]), array([4, 5, 6]), array([7, 8])]
```

Use **hsplit()** function to split an array along its horizontal axis.

You can either specify the number of equally shaped arrays to return or specify the columns after which the division should occur. Use **vsplit()** function to split an array along the vertical axis

In [ ]:
```
a = np.floor(10*np.random.random((2, 12)))
print(a)
```

```
[[3. 6. 6. 8. 1. 5. 8. 5. 1. 2. 2. 5.]
 [0. 9. 8. 8. 5. 9. 3. 7. 4. 1. 7. 7.]]
```

In [ ]:
```
print(np.hsplit(a, 3))
```

```
[array([[3., 6., 6., 8.],
       [0., 9., 8., 8.]]), array([[1., 5., 8., 5.],
```

```
       [5., 9., 3., 7.]]), array([[1., 2., 2., 5.],
       [4., 1., 7., 7.]])]
```

You split array a into three subarrays.

In [ ]:
```
print(np.vsplit(a, 2))
```

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

Use vsplit() function to split an array along the vertical axis.

---

where() function

The **numpy.where()** function returns the indices of elements in an input array where the given condition is satisfied.

> **numpy.where(condition[, x, y])**

Parameters:
condition: When True, yield x, otherwise yield y.
x, y: Values from which to choose. x, y and condition need to be broadcastable to some shape.

In [ ]:
```
arr = np.array([11, 12, 13, 14])
# high_values = ['High', 'High', 'High', 'High']
# low_values = ['Low', 'Low', 'Low', 'Low']
# numpy where() with condition argument
result = np.where(arr > 12,
                  ['High', 'High', 'High', 'High'],
                  ['Low', 'Low', 'Low', 'Low'])
print(result)


# Now we want to convert arr Numpy array arr to another array of the same size,
# where it will contain the values from lists high_values and low_values.
# Like, if the value in arr is greater than 12 then replace it with the correspondin
# from high_values i.e 'High'. Whereas, if the value in arr is less then 12 then rep
# with the corresponding value in low_values i.e. 'Low'. So, our new numpy array sho
```

```
['Low' 'Low' 'High' 'High']
```

In [ ]:
```
# Create a numpy array from list
arr = np.array([11, 12, 14, 15, 16, 17])
# pass condition expression only
result = np.where((arr > 12) & (arr < 16),
                  ['A', 'A', 'A', 'A', 'A', 'A'],
                  ['B', 'B', 'B', 'B', 'B', 'B'])
print(result)

# Here we executed multiple conditions on the array arr and it returned a bool array
# Then numpy.where() iterated over the bool array and for every True it yields corre
# element from the first list and for every False it yields the corresponding elemen
```

```
['B' 'B' 'A' 'A' 'B' 'B']
```

Insertion and Deletion

The **insert** function inserts values in the input array along the given axis and before the given index. If the type of values is converted to be inserted, it is different from the input array. Insertion is not done in place and the function returns a new array. Also, if the axis is not mentioned, the input array is flattened.

The insert() function takes the following parameters

> **numpy.insert(arr, obj, values, axis)**

Where,

arr
Input array

obj
The index before which insertion is to be made

values
The array of values to be inserted

axis
The axis along which to insert. If not given, the input array is flattened

In [ ]:

```
import numpy as np
a = np.array([[1, 2], [3, 4], [5, 6]])

print('First array:')
print(a)
print('\n')

print('Axis parameter not passed. The input array is flattened before insertion.')
print(np.insert(a, 3, [11, 12]))
print('\n')
print('Axis parameter passed. The values array is broadcast to match input array.')

print('Broadcast along axis 0:')
print(np.insert(a, 1, [11], axis=0))
print('\n')

print('Broadcast along axis 1:')
print(np.insert(a, 1, 11, axis=1))
```

```
First array:
[[1 2]
 [3 4]
 [5 6]]


Axis parameter not passed. The input array is flattened before insertion.
[ 1  2  3 11 12  4  5  6]


Axis parameter passed. The values array is broadcast to match input array.
Broadcast along axis 0:
[[ 1  2]
 [11 11]
 [ 3  4]
 [ 5  6]]
```

```
Broadcast along axis 1:
[[ 1 11  2]
 [ 3 11  4]
 [ 5 11  6]]
```

The **deletion()** function returns a new array with the specified subarray deleted from the input array. As in case of insert() function, if the axis parameter is not used, the input array is flattened. The function takes the following parameters −

> **Numpy.delete(arr, obj, axis)**

arr
Input array

obj
Can be a slice, an integer or array of integers, indicating the subarray to be deleted from the input array

axis
The axis along which to delete the given subarray. If not given, arr is flattened

In [ ]:
```
import numpy as np
a = np.arange(12).reshape(3, 4)

print('First array:')
print(a)
print('\n')

print('Array flattened before delete operation as axis not used:')
print(np.delete(a, 5))
print('\n')

print('Column 2 deleted:')
print(np.delete(a, 1, axis=1))
print('\n')
```

```
First array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]


Array flattened before delete operation as axis not used:
[ 0  1  2  3  4  6  7  8  9 10 11]


Column 2 deleted:
[[ 0  2  3]
 [ 4  6  7]
 [ 8 10 11]]
```

Broadcasting

Broadcasting is a powerful mechanism that allows NumPy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger

array, and we want to use the smaller array multiple times to perform some operation on the larger array.

Rules of Broadcasting

Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:

- **Rule 1:** If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
- **Rule 2:** If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- **Rule 3:** If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

To make these rules clear, let's consider a few examples in detail.

Broadcasting example 1

Let's look at adding a two-dimensional array to a one-dimensional array:

In [ ]:
```
import numpy as np

M = np.ones((2, 3))
a = np.arange(3)
```

Let's consider an operation on these two arrays. The shape of the arrays are

```
M.shape = (2, 3)
a.shape = (3,)
```

We see by rule 1 that the array `a` has fewer dimensions, so we pad it on the left with ones:

```
M.shape -> (2, 3)
a.shape -> (1, 3)
```

By rule 2, we now see that the first dimension disagrees, so we stretch this dimension to match:

```
M.shape -> (2, 3)
a.shape -> (2, 3)
```

The shapes match, and we see that the final shape will be `(2, 3)`

In [ ]:
```
M + a
```

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

Broadcasting example 2 Let's take a look at an example where both arrays need to be broadcast:

In [ ]:
```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

Again, we'll start by writing out the shape of the arrays:

```
a.shape = (3, 1)
b.shape = (3,)
```

Rule 1 says we must pad the shape of b with ones:

```
a.shape -> (3, 1)
b.shape -> (1, 3)
```

And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

```
a.shape -> (3, 3)
b.shape -> (3, 3)
```

Because the result matches, these shapes are compatible. We can see this here:

In [ ]:
```
a + b
```

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

Broadcasting example 3 Now let's take a look at an example in which the two arrays are not compatible:

In [ ]:
```
M = np.ones((3, 2))
a = np.arange(3)
```

This is just a slightly different situation than in the first example: the matrix M is transposed. How does this affect the calculation? The shape of the arrays are

```
M.shape = (3, 2)
a.shape = (3,)
```

Again, rule 1 tells us that we must pad the shape of a with ones:

```
M.shape -> (3, 2)
a.shape -> (1, 3)
```

By rule 2, the first dimension of a is stretched to match that of M:

```
M.shape -> (3, 2)
a.shape -> (3, 3)
```

Now we hit rule 3–the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

In [ ]:
```
M + a
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
C:\Users\GOWRIS~1\AppData\Local\Temp/ipykernel_3088/3374645918.py in <module>
```

```
----> 1 M + a

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

Pandas

- **Pandas** is an open source Python library for data analysis.
- Started by Wes McKinney in 2009
- Emerged from the finance industry. Motivated by the toolbox in R programming language for manipulating data easilty. But Python is a better language :-)
- First public release in 0.3 in February 2011.
- Grown and maintained by a huge community now, headed by Jeff Reback
- Open Source (BSD Licensed)
- Has become a **corner stone of the SciPy ecosystem** for all things related to Data!!

Data Structures

Pandas introduces two new data structures to Python - **Series** and **DataFrame**, both of which are built on top of NumPy (this means it's fast).

In [ ]:
```
import pandas as pd
import numpy as np
import pprint as pp
```

Series

A **Series** is a one-dimensional object similar to an array, list, or column in a table. It will assign a labeled index to each item in the Series. By default, each item will receive an index label from 0 to N, where N is the length of the Series minus one.

Pandas Series is created using series() method and its syntax is,

s = pd.Series(data, index=None)

Here, s is the Pandas Series, data can be a Python dict, a ndarray, or a scalar value (like 5).

In [ ]:
```
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
print(type(s))
```

```
<class 'pandas.core.series.Series'>
```

In [ ]:
```
print(s)
```

```
a   -1.659197
b   -1.055822
c    1.276263
d    0.458444
e   -1.583545
dtype: float64
```

In [ ]:
```
print(s.index)
```

```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

In [ ]:
```
print(s.values)
```

```
[-1.65919728 -1.05582156  1.27626319  0.45844424 -1.58354515]
```

In [ ]:
```
print(pd.Series(np.random.randn(5)))
```

```
0    1.534788
1    0.264475
2   -0.712104
3    1.522914
4   -0.751795
dtype: float64
```

Create Series from Dictionaries

In [ ]:
```
d = {'a' : 0., 'b' : 1., 'c' : 2.}
print(pd.Series(d))
print("\n")
print(pd.Series(d, index=['b', 'c', 'd', 'a']))
```

```
a    0.0
b    1.0
c    2.0
dtype: float64


b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Create Series from Scalar data

In [ ]:
```
pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
```

Out[ ]:
```
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

Series Indexing and Slicing

In [ ]:
```
pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
print(s)
```

```
a   -1.659197
b   -1.055822
c    1.276263
d    0.458444
e   -1.583545
dtype: float64
```

In [ ]:
```
print(s[0])
```

```
-1.6591972773760035
```

In [ ]:
```
print(s[1:3])
```

```
b   -1.055822
c    1.276263
dtype: float64
```

In [ ]:
```
print(s[:3])
```

```
a   -1.659197
b   -1.055822
c    1.276263
dtype: float64
```

In [ ]:
```
print(s[s > .5])
```

```
b    0.783201
c    1.190083
e    0.535637
dtype: float64
```

In [ ]:
```
print(s[[4, 3, 1]])
```

```
e   -1.583545
d    0.458444
b   -1.055822
dtype: float64
```

In [ ]:
```
print(s['a'])
```

```
-1.6591972773760035
```

In [ ]:
```
print(s['e'])
```

```
-1.583545153866696
```

In [ ]:
```
print('e' in s)
```

```
True
```

In [ ]:
```
print('f' in s)
```

```
False
```

Working with Text Data

The Pandas Series supports a set of string processing methods that make it easy to operate on each element of the array.

These methods are accessible via the str attribute and they generally have the same name as that of the built-in Python string methods.

In [ ]:
```
empires_ds = pd.Series(["Vijayanagara", "Roman", "Chola", "Mongol", "Akkadian"])
print(empires_ds.str.lower())
```

```
0    vijayanagara
1           roman
2           chola
3          mongol
4        akkadian
dtype: object
```

In [ ]:
```
print(empires_ds.str.upper())
```

```
0    VIJAYANAGARA
1           ROMAN
2           CHOLA
3          MONGOL
4        AKKADIAN
dtype: object
```

In [ ]:
```
print(empires_ds.str.len())
```

```
0    12
1     5
2     5
3     6
4     8
dtype: int64
```

In [ ]:
```
tennis_ds = pd.Series([' Seles ', ' Graph ', ' Williams '])
```

In [ ]:
```
print(tennis_ds.str.strip())
```

```
0       Seles
1       Graph
2    Williams
dtype: object
```

In [ ]:
```
print(tennis_ds.str.contains(' '))
```

```
0    True
1    True
2    True
dtype: bool
```

In [ ]:
```
marvel_ds = pd.Series(['Thor_loki', 'Thor_Hulk', 'Gamora_Storm'])
print(marvel_ds.str.split('_'))
```

```
0       [Thor, loki]
1       [Thor, Hulk]
2    [Gamora, Storm]
dtype: object
```

In [ ]:
```
planets = pd.Series(["Venus", "Earth", "Saturn"])
print(planets.str.replace("Earth", "Mars"))
```

```
0     Venus
1      Mars
2    Saturn
dtype: object
```

In [ ]:
```
letters_ds = pd.Series(['a', 'b', 'c', 'd'])
print(letters_ds.str.cat(sep=','))
```

```
a,b,c,d
```

In [ ]:
```
names_ds = pd.Series(['Jahnavi', 'Adelmo', 'Pietro', 'Alejandro'])
print(names_ds.str.count('e'))
```

```
0    0
1    1
2    1
3    1
dtype: int64
```

In [ ]:
```
print(names_ds.str.startswith('A'))
```

```
0    False
1     True
2    False
3     True
dtype: bool
```

In [ ]:
```
print(names_ds.str.endswith('O'))
```

```
0    False
1    False
2    False
3    False
dtype: bool
```

In [ ]:
```
print(names_ds.str.find('J'))
```

```
0    0
1   -1
2   -1
```

```
3   -1
dtype: int64
```

DataFrame

A DataFrame is a tablular data structure comprised of rows and columns, akin to a spreadsheet, database table. You can also think of a DataFrame as a group of Series objects that share an index (the column names).

Name	Age	Gender	Rating
Steve	32	Male	3.45
Lia	28	Female	4.6
Vin	45	Male	3.9
Katie	38	Female	2.78

As can be seen in the above figure, DataFrame is a two-dimensional array with heterogeneous data. The table represents the data of a sales team of an organization with their overall performance rating. The data is represented in rows and columns. Each column represents an attribute and each row represents a person.

Data Type of Columns
The data types of the four columns are as follows:

Column	Type
Name	String
Age	Integer
Gender	String
Rating	Float

Key Points

- Heterogeneous Data
- Size Mutable
- Data Mutable
- Labeled Axes (rows and columns)

DataFrame accepts many different kinds of input like Dict of one-dimensional ndarrays, lists, dicts, or Series, two-dimensional ndarrays, structured or record ndarray, a dictionary of Series, or another DataFrame.

df = pd.DataFrame(data=None, index=None, columns=None)

Here, df is the DataFrame and data can be NumPy ndarray, dict, or DataFrame.

Along with the data, you can optionally pass an index (row labels) and columns (column labels) attributes as arguments.

Create DataFrame from Dictionary of Series/Dictionaries

In [ ]:
```
dict_series = {
    "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(dict_series)
print(df)
```

```
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0
```

In [ ]:
```
print(df.shape)
```

```
(4, 2)
```

In [ ]:
```
print(df.index)
```

```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

In [ ]:
```
print(df.columns)
```

```
Index(['one', 'two'], dtype='object')
```

In [ ]:
```
print(list(df.columns))
```

```
['one', 'two']
```

In [ ]:
```
dicts_only = {'a':[1,2,3], 'b':[4,5,6]}
dict_df = pd.DataFrame(dicts_only)
print(dict_df)
```

```
   a  b
0  1  4
1  2  5
2  3  6
```

In [ ]:
```
print(dict_df.index)
```

```
RangeIndex(start=0, stop=3, step=1)
```

Create DataFrame from ndarrays/lists/list of dictionaries

In [ ]:
```
dict_ndarrays = {"one": np.random.random(5), "two": np.random.random(5)}
```

In [ ]:
```
print(pd.DataFrame(dict_ndarrays))
```

```
        one       two
```

```
0  0.967265  0.693171
1  0.420759  0.804217
2  0.925897  0.957192
3  0.473513  0.125198
4  0.953426  0.509628
```

In [ ]:
```
print(pd.DataFrame([[1,2,3,4,5], [6,7,8,9,10]]))
```

```
   0  1  2  3   4
0  1  2  3  4   5
1  6  7  8  9  10
```

In [ ]:
```
dict_lists = {'one': [1, 2, 3, 4, 5], 'two': [5, 4, 3, 2, 1]}
print(pd.DataFrame(dict_lists))
```

```
   one  two
0    1    5
1    2    4
2    3    3
3    4    2
4    5    1
```

In [ ]:
```
print(pd.DataFrame(dict_lists, index=['a', 'b', 'c', 'd', 'e']))
```

```
   one  two
a    1    5
b    2    4
c    3    3
d    4    2
e    5    1
```

In [ ]:
```
lists_dicts = [{'a':1, 'b':2}, {'a':5, 'b':10, 'c':20}]
print(pd.DataFrame(lists_dicts))
```

```
   a   b     c
0  1   2   NaN
1  5  10  20.0
```

DataFrame Column Selection, Addition and Deletion

In [ ]:
```
la_liga = {
    "Ranking": [1, 2, 3],
    "Team": ["Barcelona", "Atletico Madrid", "Real Madrid"],
}
df = pd.DataFrame(la_liga)
print(df)
```

```
   Ranking             Team
0        1        Barcelona
1        2  Atletico Madrid
2        3      Real Madrid
```

In [ ]:
```
df["Played"] = [34, 36, 38]
```

In [ ]:
```
df["Won"] = [27, 23, 22]
```

In [ ]:

```
print(df[["Played", "Won"]])
```

```
   Played  Won
0      34   27
1      36   23
2      38   22
```

In [ ]:
```
df["Points"] = df["Won"] * 2
```

In [ ]:
```
print(df)
```

```
   Ranking             Team  Played  Won  Points
0        1        Barcelona      34   27      54
1        2  Atletico Madrid      36   23      46
2        3      Real Madrid      38   22      44
```

In [ ]:
```
df["Lost"] = [1, 5, 6]
print(df)
```

```
   Ranking             Team  Played  Won  Points  Lost
0        1        Barcelona      34   27      54     1
1        2  Atletico Madrid      36   23      46     5
2        3      Real Madrid      38   22      44     6
```

In [ ]:
```
df["Drawn"] = df["Played"] - df["Won"] - df["Lost"]
print(df)
```

```
   Ranking             Team  Played  Won  Points  Lost  Drawn
0        1        Barcelona      34   27      54     1      6
1        2  Atletico Madrid      36   23      46     5      8
2        3      Real Madrid      38   22      44     6     10
```

In [ ]:
```
df["Year"] = 2018
print(df)
```

```
   Ranking             Team  Played  Won  Points  Lost  Drawn  Year
0        1        Barcelona      34   27      54     1      6  2018
1        2  Atletico Madrid      36   23      46     5      8  2018
2        3      Real Madrid      38   22      44     6     10  2018
```

In [ ]:
```
del df["Year"]
```

In [ ]:
```
df.pop("Drawn")
```

Out[ ]:
```
0     6
1     8
2    10
Name: Drawn, dtype: int64
```

In [ ]:
```
df.insert(5, 'Goal Difference', [63, 38, 42])
print(df)
```

```
   Ranking             Team  Played  Won  Points  Goal Difference  Lost
0        1        Barcelona      34   27      54               63     1
1        2  Atletico Madrid      36   23      46               38     5
2        3      Real Madrid      38   22      44               42     6
```

Displaying Data in DataFrame

In [ ]:
```
df = pd.DataFrame(
    {
        "WorldCup_Winner": ["Brazil", "Germany", "Argentina", "Brazil", "Spain"],
        "Year": [1962, 1974, 1986, 2002, 2010],
    }
)

print(df)
print("\n")
print(df.columns)
```

```
  WorldCup_Winner  Year
0          Brazil  1962
1         Germany  1974
2       Argentina  1986
3          Brazil  2002
4           Spain  2010


Index(['WorldCup_Winner', 'Year'], dtype='object')
```

In [ ]:
```
print("Head")
print(df.head(2))
print("\n")
print("Tail")
print(df.tail(2))
```

```
Head
  WorldCup_Winner  Year
0          Brazil  1962
1         Germany  1974


Tail
  WorldCup_Winner  Year
3          Brazil  2002
4           Spain  2010
```

In [ ]:
```
df['WorldCup_Winner'].unique()
```

Out[ ]:
```
array(['Brazil', 'Germany', 'Argentina', 'Spain'], dtype=object)
```

In [ ]:
```
df['WorldCup_Winner'].unique().tolist()
```

Out[ ]:
```
['Brazil', 'Germany', 'Argentina', 'Spain']
```

In [ ]:
```
df.transpose()
```

Out[ ]:

	0	1	2	3	4
WorldCup_Winner	Brazil	Germany	Argentina	Brazil	Spain
Year	1962	1974	1986	2002	2010

In [ ]:
```
df.sort_values(by=['Year'], ascending = False)
```

Out[ ]:

	WorldCup_Winner	Year
4	Spain	2010
3	Brazil	2002
2	Argentina	1986
1	Germany	1974
0	Brazil	1962

The syntax for sort_values() method is,

df.sort_values(by, axis=0, ascending=True)

where the by parameter can be a string, list of strings, index label, column label, list of index labels, or list of column labels to sort by.

If the value of the axis is 0 then by may contain column labels.

If the value of the axis is 1, then by may contain index labels.

In [ ]:
```
df.sort_index(ascending = False)
```

Out[ ]:

	WorldCup_Winner	Year
4	Spain	2010
3	Brazil	2002
2	Argentina	1986
1	Germany	1974
0	Brazil	1962

In [ ]:
```
df['WorldCup_Winner'].value_counts()
```

Out[ ]:
```
Brazil       2
Spain        1
Germany      1
Argentina    1
Name: WorldCup_Winner, dtype: int64
```

The **value_counts()** method when chained with a DataFrame, returns a Series object containing counts of unique values .

In [ ]:
```
df['WorldCup_Winner'].value_counts().index.tolist()
```

Out[ ]:
```
['Brazil', 'Spain', 'Germany', 'Argentina']
```

In [ ]:
```
df['WorldCup_Winner'].value_counts().values.tolist()
```

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

Using DataFrame assign() method

In [ ]:
```
df_mountain = pd.DataFrame(
    {"Mountain": ["Mount Everest", "K2", "Kangchenjunga"], "Length": [8848, 8611, 85
)
df_mountain.assign(Ranking=[1, 2, 3])
```

Out[ ]:

	Mountain	Length	Ranking
0	Mount Everest	8848	1
1	K2	8611	2
2	Kangchenjunga	8586	3

In [ ]:
```
df = pd.DataFrame({'A':[2, 4, 6], 'B':[3, 6, 9]})
df.assign(C = lambda x:x['A'] ** 2)
```

Out[ ]:

	A	B	C
0	2	3	4
1	4	6	16
2	6	9	36

DataFrame Indexing and Selecting Data

The Python and NumPy indexing operator `[ ]` and the `"."` dot operators provide quick and easy access to Pandas data structures across a wide range of use cases.

However, since the type of the data to be accessed isn't known in advance, directly using standard operators has some optimization limits.

For production code, it is highly recommend that you take advantage of the optimized pandas data access methods `.loc()` and `.iloc()`

The .loc() data access method

The `.loc[]` method is primarily `label based` but may also be used with

- A single scalar label
- A list of labels
- A slice object
- A Boolean array

The `.loc[]` method takes two arguments separated by `','`. The first one indicates the row and the second one indicates columns. The range operator can also be used as argument.

In [ ]:
```
df = pd.DataFrame(
    np.random.rand(5, 5),
    index=["row_1", "row_2", "row_3", "row_4", "row_5"],
    columns=["col_1", "col_2", "col_3", "col_4", "col_5"],
)
```

```
print("Display DataFrame")
print(df)
print("\n")
print("Display Row 1 Contents")
df.loc["row_1"]
```

```
Display DataFrame
          col_1     col_2     col_3     col_4     col_5
row_1  0.990475  0.813931  0.282323  0.272060  0.096406
row_2  0.919191  0.020265  0.297995  0.251167  0.246483
row_3  0.802696  0.870913  0.126242  0.413905  0.692361
row_4  0.770466  0.810844  0.110056  0.749679  0.629315
row_5  0.960473  0.174073  0.905638  0.285938  0.431395


Display Row 1 Contents
```

Out[ ]:

```
col_1    0.990475
col_2    0.813931
col_3    0.282323
col_4    0.272060
col_5    0.096406
Name: row_1, dtype: float64
```

In [ ]:

```
df.loc['row_2', 'col_3']
```

Out[ ]:

```
0.297995151231605
```

In [ ]:

```
df.loc[['row_1', 'row_2'],['col_2', 'col_3']]
```

Out[ ]:

	col_2	col_3
row_1	0.813931	0.282323
row_2	0.020265	0.297995

In [ ]:

```
df.loc[:, ['col_2', 'col_3']]
```

Out[ ]:

	col_2	col_3
row_1	0.813931	0.282323
row_2	0.020265	0.297995
row_3	0.870913	0.126242
row_4	0.810844	0.110056
row_5	0.174073	0.905638

The .iloc() data access method

The `.iloc[]` method is primarily `integer based` but may also be used with

- A list of integers
- A range of values
- A Boolean array

In [ ]:

```
df.iloc[1]
```

Out[ ]:
```
col_1    0.919191
col_2    0.020265
col_3    0.297995
col_4    0.251167
col_5    0.246483
Name: row_2, dtype: float64
```

In [ ]:
```
df.iloc[3:5, 0:2]
```

Out[ ]:

	col_1	col_2
row_4	0.770466	0.810844
row_5	0.960473	0.174073

In [ ]:
```
df.iloc[:3, :]
```

Out[ ]:

	col_1	col_2	col_3	col_4	col_5
row_1	0.208566	0.989302	0.735267	0.952551	0.831082
row_2	0.997031	0.870984	0.578950	0.937640	0.371607
row_3	0.330112	0.375134	0.712488	0.378875	0.414045

In [ ]:
```
df.iloc[:,:]
```

Out[ ]:

	col_1	col_2	col_3	col_4	col_5
row_1	0.208566	0.989302	0.735267	0.952551	0.831082
row_2	0.997031	0.870984	0.578950	0.937640	0.371607
row_3	0.330112	0.375134	0.712488	0.378875	0.414045
row_4	0.061693	0.721538	0.202972	0.447766	0.527475
row_5	0.379384	0.541739	0.517974	0.204813	0.113300

In [ ]:
```
df.iloc[2:, 2:]
```

Out[ ]:

	col_3	col_4	col_5
row_3	0.712488	0.378875	0.414045
row_4	0.202972	0.447766	0.527475
row_5	0.517974	0.204813	0.113300

In [ ]:
```
df.iloc[:,1]
```

Out[ ]:
```
row_1    0.989302
row_2    0.870984
row_3    0.375134
row_4    0.721538
row_5    0.541739
Name: col_2, dtype: float64
```

In [ ]:
```
df[df > 0.2]
```

Out[ ]:

	col_1	col_2	col_3	col_4	col_5
row_1	0.208566	0.989302	0.735267	0.952551	0.831082
row_2	0.997031	0.870984	0.578950	0.937640	0.371607
row_3	0.330112	0.375134	0.712488	0.378875	0.414045
row_4	NaN	0.721538	0.202972	0.447766	0.527475
row_5	0.379384	0.541739	0.517974	0.204813	NaN

Handling the Missing Data

Missing data is always a problem in real life scenarios. Areas like machine learning and data mining face severe issues in the accuracy of their model predictions because of poor quality of data caused by missing values. In these areas, missing value treatment is a major point of focus to make their models more accurate and valid.

When and Why Is Data Missed?

Let us consider an online survey for a product. Many a times, people do not share all the information related to them. Few people share their experience, but not how long they are using the product; few people share how long they are using the product, their experience but not their contact information. Thus, in some or the other way a part of data is always missing, and this is very common in real time.

Let us now see how we can handle missing values, say `NA` or `NaN` using Pandas. `NaN` means Not a Number.

In [ ]:
```
# import the pandas library
import pandas as pd
import numpy as np

df = pd.DataFrame(
    np.random.randn(5, 3),
    index=["a", "c", "e", "f", "h"],
    columns=["one", "two", "three"],
)

df = df.reindex(["a", "b", "c", "d", "e", "f", "g", "h"])
# Using reindexing, we have created a DataFrame with missing values.
print(df)
```

```
        one       two     three
a  2.014595 -0.441174 -0.306696
b       NaN       NaN       NaN
c -0.774608  0.376916  0.936396
d       NaN       NaN       NaN
e -0.459956  0.360368 -0.063948
f -1.344253 -0.413500  1.086477
g       NaN       NaN       NaN
h  1.222063 -1.322749  1.087837
```

Check for Missing Values

To make detecting missing values easier (and across different array dtypes), Pandas provides the `isnull()` and `notnull()` functions, which are also methods on Series and DataFrame objects.

In [ ]:
```
import pandas as pd
import numpy as np

df = pd.DataFrame(
    np.random.randn(5, 3),
    index=["a", "c", "e", "f", "h"],
    columns=["one", "two", "three"],
)

df = df.reindex(["a", "b", "c", "d", "e", "f", "g", "h"])

print(df["one"].isnull())
```

```
a    False
b     True
c    False
d     True
e    False
f    False
g     True
h    False
Name: one, dtype: bool
```

In [ ]:
```
import pandas as pd
import numpy as np

df = pd.DataFrame(
    np.random.randn(5, 3),
    index=["a", "c", "e", "f", "h"],
    columns=["one", "two", "three"],
)

df = df.reindex(["a", "b", "c", "d", "e", "f", "g", "h"])

print(df["one"].notnull())
```

```
a     True
b    False
c     True
d    False
e     True
f     True
g    False
h     True
Name: one, dtype: bool
```

Calculations with Missing Data

- When summing data, `NA` will be treated as `Zero`
- If the data are all `NA`, then the result will be `NA`

In [ ]:
```
import pandas as pd
import numpy as np

df = pd.DataFrame(
    np.random.randn(5, 3),
```

```
    index=["a", "c", "e", "f", "h"],
    columns=["one", "two", "three"],
)

df = df.reindex(["a", "b", "c", "d", "e", "f", "g", "h"])

print(df["one"].sum())
```

```
-1.91928177770235
```

Cleaning / Filling Missing Data

Pandas provides various methods for cleaning the missing values. The fillna function can "fill in" NA values with non-null data in a couple of ways, which we have illustrated in the following sections.

Replace NaN with a Scalar Value

The following program shows how you can replace `"NaN"` with `"0"`.

In [ ]:
```
import pandas as pd
import numpy as np

df = pd.DataFrame(
    np.random.randn(3, 3), index=["a", "c", "e"], columns=["one", "two", "three"]
)

df = df.reindex(["a", "b", "c"])

print("Display DataFrame")
print(df)
print("\n")

print("NaN replaced with '0':")
print(df.fillna(0))
```

```
Display DataFrame
        one       two     three
a  0.399897 -0.761325  0.215413
b       NaN       NaN       NaN
c -0.293793  1.042044  0.793723


NaN replaced with '0':
        one       two     three
a  0.399897 -0.761325  0.215413
b  0.000000  0.000000  0.000000
c -0.293793  1.042044  0.793723
```

Fill NA Forward and Backward

Method	Action
ffill	fills with forward values
bfill	fills with backward values

In [ ]:
```
import pandas as pd
import numpy as np
```

```
df = pd.DataFrame(
    np.random.randn(5, 3),
    index=["a", "c", "e", "f", "h"],
    columns=["one", "two", "three"],
)

df = df.reindex(["a", "b", "c", "d", "e", "f", "g", "h"])
print("Display DataFrame")
print(df)
print("\n")
print("Filled with Forward Values")
print(df.fillna(method="ffill"))
```

```
Display DataFrame
        one       two     three
a -0.867275 -0.196523 -0.375862
b       NaN       NaN       NaN
c -0.046849  0.154467  0.570363
d       NaN       NaN       NaN
e -0.055993  1.434254 -0.907202
f -0.428485  1.449075 -1.725541
g       NaN       NaN       NaN
h -0.054629 -0.032209 -0.046278


Filled with Forward Values
        one       two     three
a -0.867275 -0.196523 -0.375862
b -0.867275 -0.196523 -0.375862
c -0.046849  0.154467  0.570363
d -0.046849  0.154467  0.570363
e -0.055993  1.434254 -0.907202
f -0.428485  1.449075 -1.725541
g -0.428485  1.449075 -1.725541
h -0.054629 -0.032209 -0.046278
```

In [ ]:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(
    np.random.randn(5, 3),
    index=["a", "c", "e", "f", "h"],
    columns=["one", "two", "three"],
)

df = df.reindex(["a", "b", "c", "d", "e", "f", "g", "h"])
print("Display DataFrame")
print(df)
print("\n")

print("Filled with Backward Values")
print(df.fillna(method="bfill"))
```

```
Display DataFrame
        one       two     three
a -0.764667  0.464803  0.581098
b       NaN       NaN       NaN
c -0.334854 -0.431129 -0.040747
d       NaN       NaN       NaN
e -0.103632  0.853901  0.364664
f  0.114206  0.707693  0.638021
g       NaN       NaN       NaN
h  0.640991 -0.362189  1.263950
```

```
Filled with Backward Values
        one       two     three
a -0.764667  0.464803  0.581098
b -0.334854 -0.431129 -0.040747
c -0.334854 -0.431129 -0.040747
d -0.103632  0.853901  0.364664
e -0.103632  0.853901  0.364664
f  0.114206  0.707693  0.638021
g  0.640991 -0.362189  1.263950
h  0.640991 -0.362189  1.263950
```

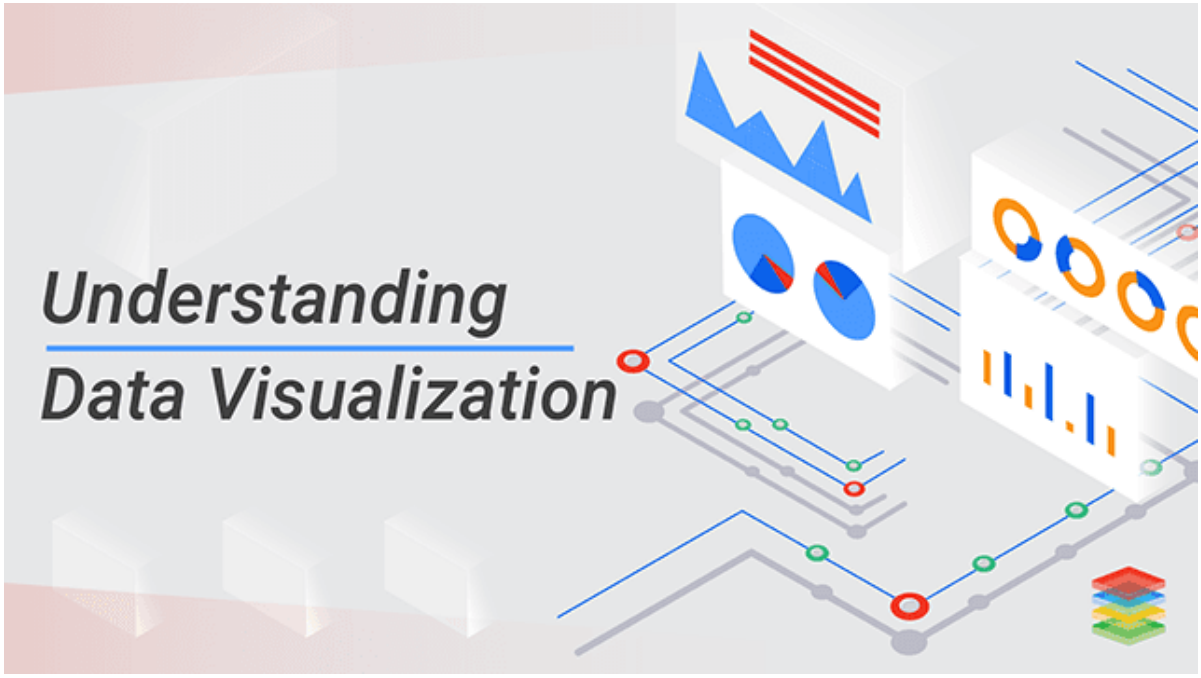
Drop Missing Values

If you want to simply exclude the missing values, then use the `dropna()` function along with the axis argument. By default, axis = 0, i.e., along row, which means that if any value within a row is NA then the whole row is excluded.

In [ ]:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(
    np.random.randn(5, 3),
    index=["a", "c", "e", "f", "h"],
    columns=["one", "two", "three"],
)

print("Display DataFrame")
print(df)
print("\n")

print("Drop Missing Values")
df = df.reindex(["a", "b", "c", "d", "e", "f", "g", "h"])
print(df.dropna())
```

```
Display DataFrame
        one       two     three
a  0.486551  1.375397 -0.877042
c -1.506840 -1.436973  1.074966
e -0.184624 -0.080299  0.963973
f  0.622986  0.382013  1.041979
h -0.832825 -0.012596  0.956276


Drop Missing Values
        one       two     three
a  0.486551  1.375397 -0.877042
c -1.506840 -1.436973  1.074966
e -0.184624 -0.080299  0.963973
f  0.622986  0.382013  1.041979
h -0.832825 -0.012596  0.956276
```



What is Data Visualization?

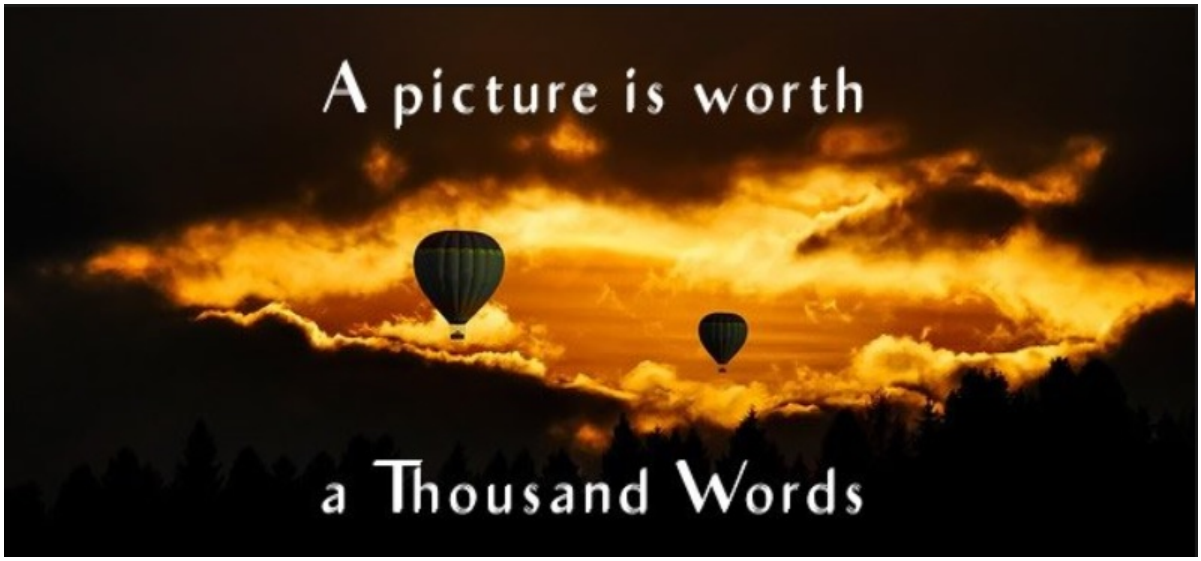
- Data Visualization is basically a graphical representation of information and data.
- It is a visual content through which people understand the significance of data.
- In general, patterns, trends, and correlations might go unnoticed in text-based form data but through visualizations, with various techniques, it can be exposed and recognized easier.

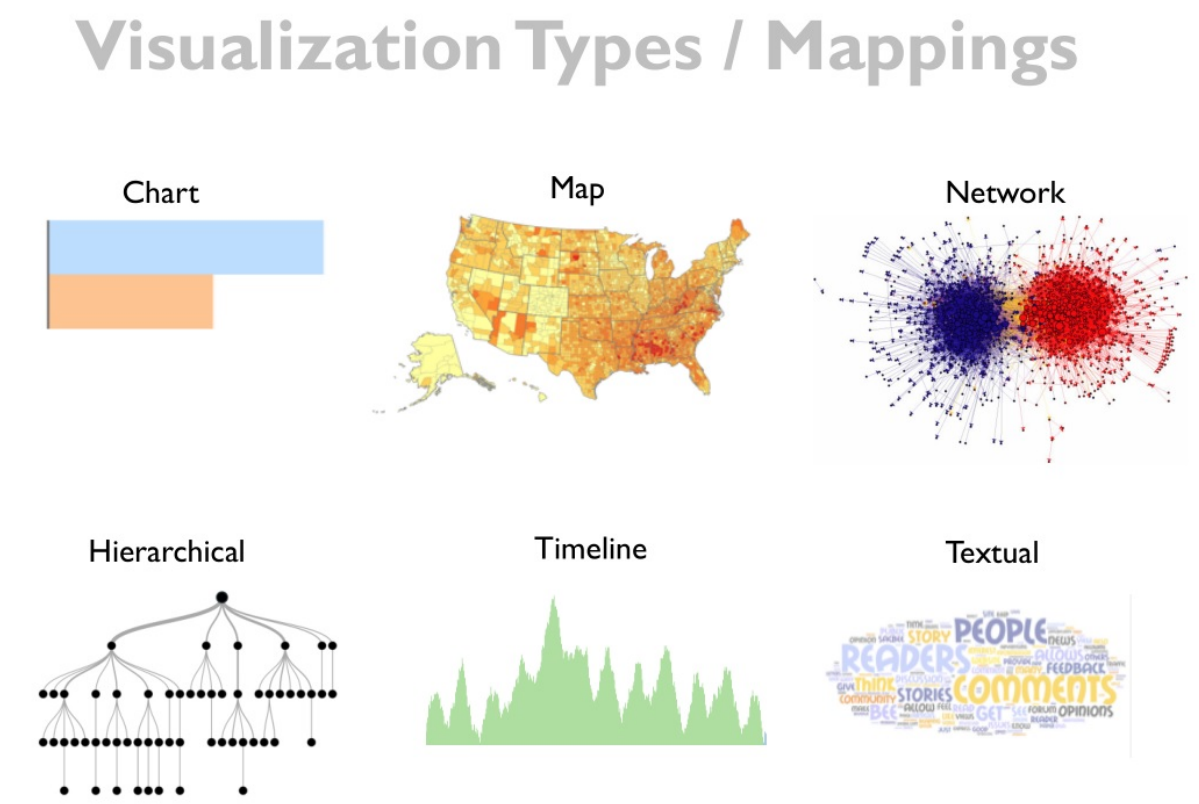
Power of Visualization

Count the “8”s

094026484703364338112201922411706617179386362840535012774977775741603594439227825014545795883321091511077342211299131153390824373906153048606092192347355250749740807178349437116722169576995339650996446716804111346

094026484703364338112201922411706617179386362840535012774977775741603594439227825014545795883321091511077342211299131153390824373906153048606092192347355250749740807178349437116722169576995339650996446716804111346





Uses of Data Visualization

A: Support reasoning about information (analysis)

- Finding relationships
- Discover structure
- Quantifying values and influences
- Should be part of a query/analyze cycle

B: Inform and persuade others (communication)

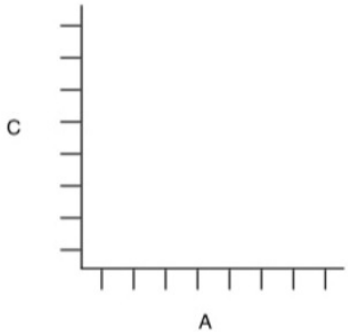
- Capture attention, engage
- Tell a story visually

- Focus on certain aspects, and omit others

Elements of a Plot

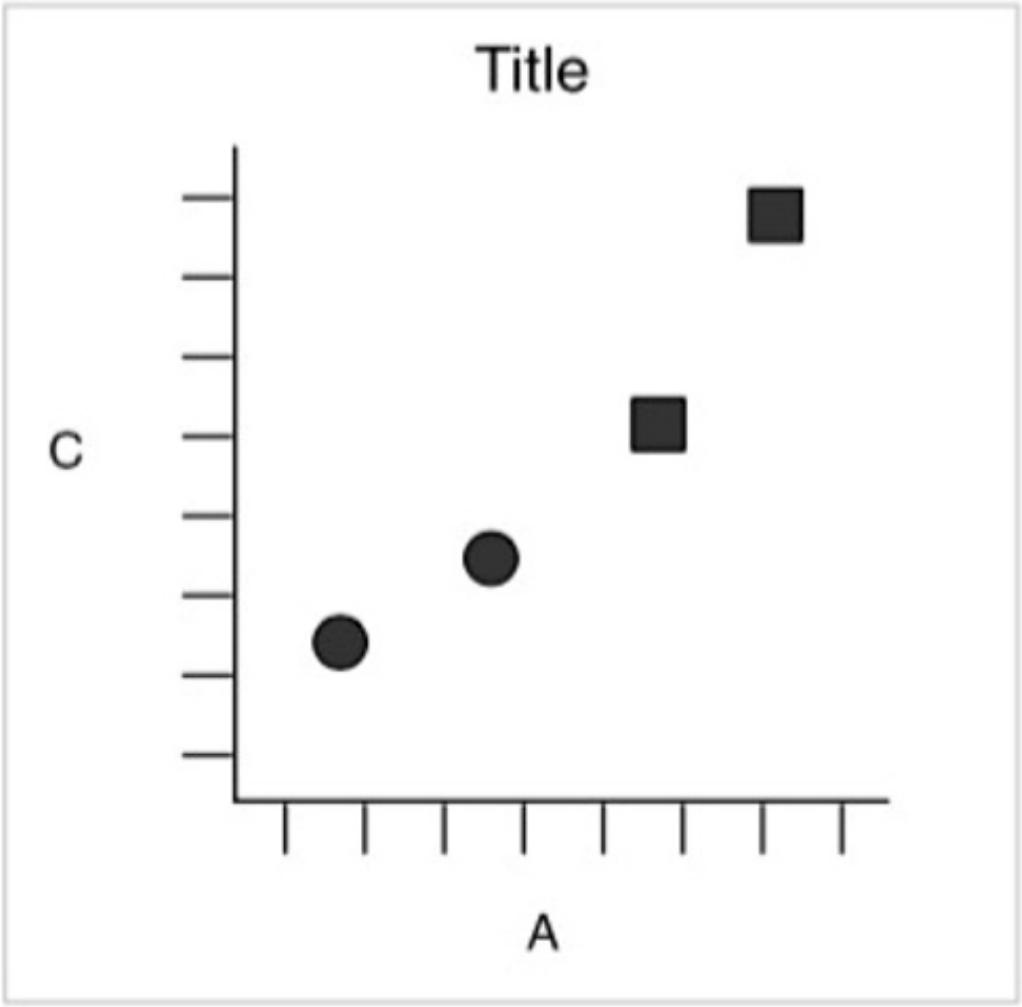
Geometric Objects

Scales & Coordinates

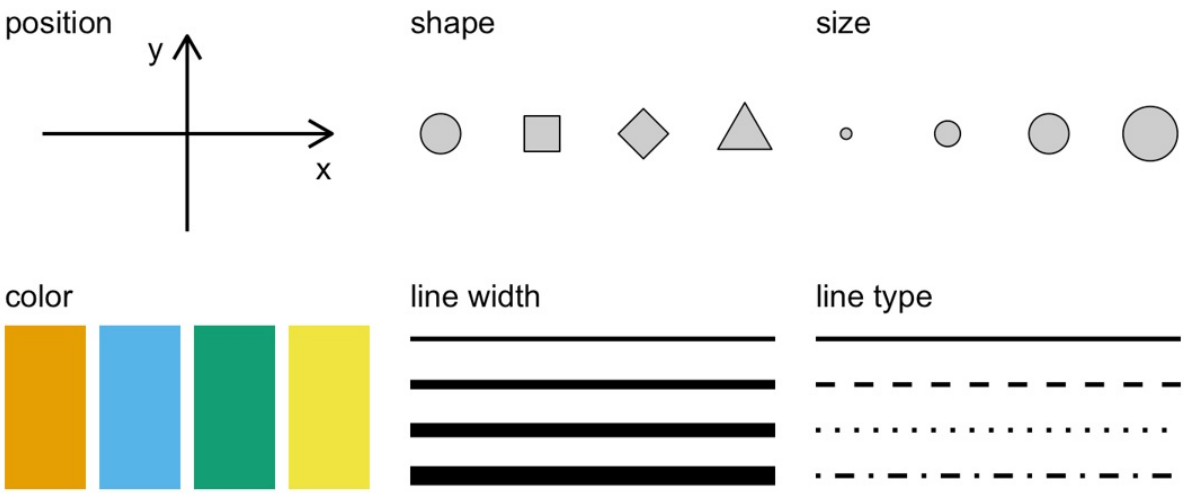


Annotations

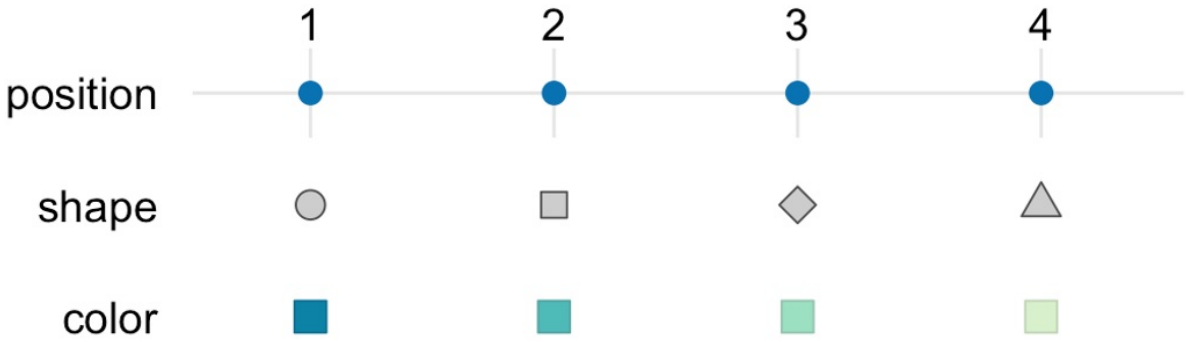
Elements of a Plot

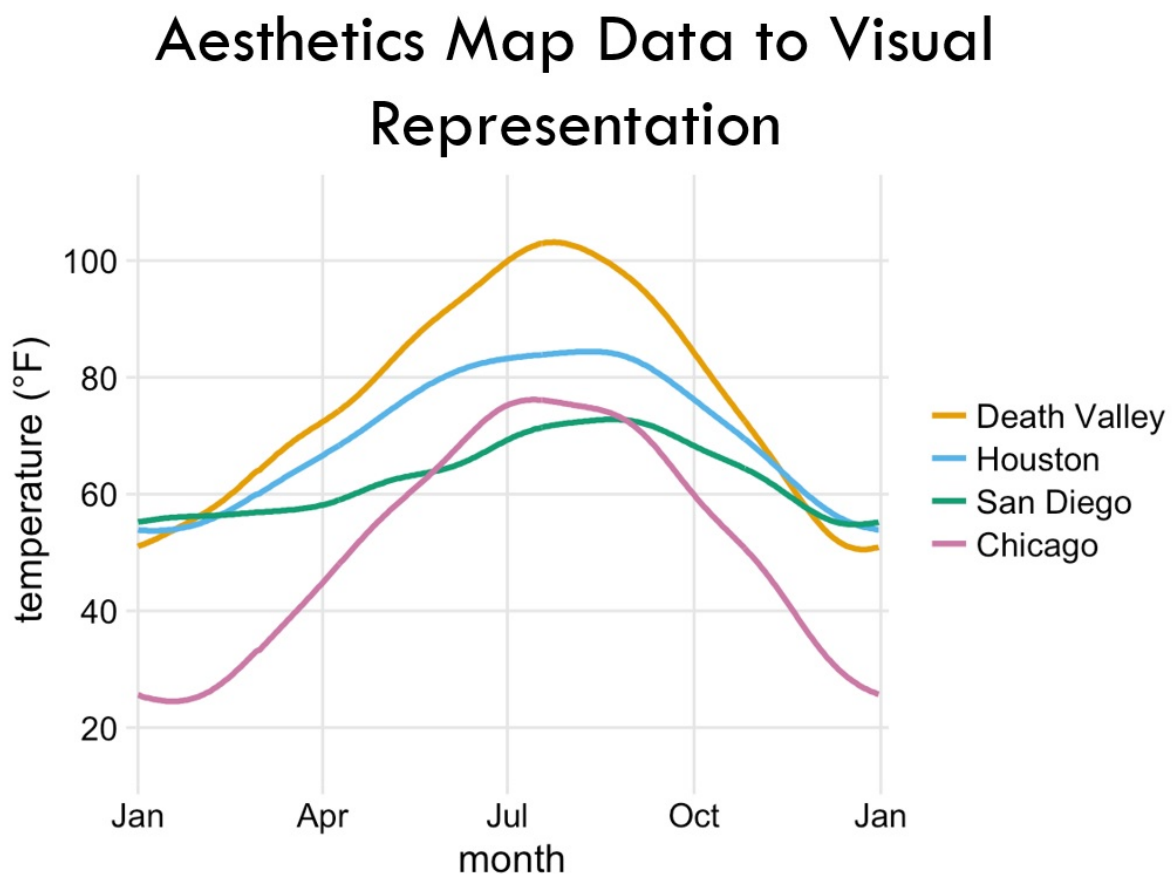


Aesthetics of a Plot



Aesthetics Map Data to Visual Representation





Matplotlib is a popular Python library that can be used to create data visualizations quite easily. It is probably the single most used Python package for 2D-graphics along with limited support for 3D-graphics.
It provides both, a very quick way to visualize data from Python and publication-quality figures in many formats.
</br>

Also, It was designed from the beginning to serve two purposes:

- Allow for interactive, cross-platform control of figures and plots.
- Make it easy to produce static vector graphics files without the need for any GUIs.

Much like Python itself, Matplotlib gives the developer complete control over the appearance of their plots.
It tries to make easy things easy and hard things possible.
We can generate plots, histograms, bar charts, error charts, scatter plots, etc. with just a few lines of code.
For simple plotting, the `pyplot` module within the matplotlib package provides a MATLAB-like

interface to the underlying `object-oriented plotting library.`
It implicitly and automatically creates figures and axes to achieve the desired plot.

How to import Python Matplotlib

To get started with Python Matplotlib, we first import the package.
It is a common practice to `import matplotlib.pyplot` using the alias `as plt` .

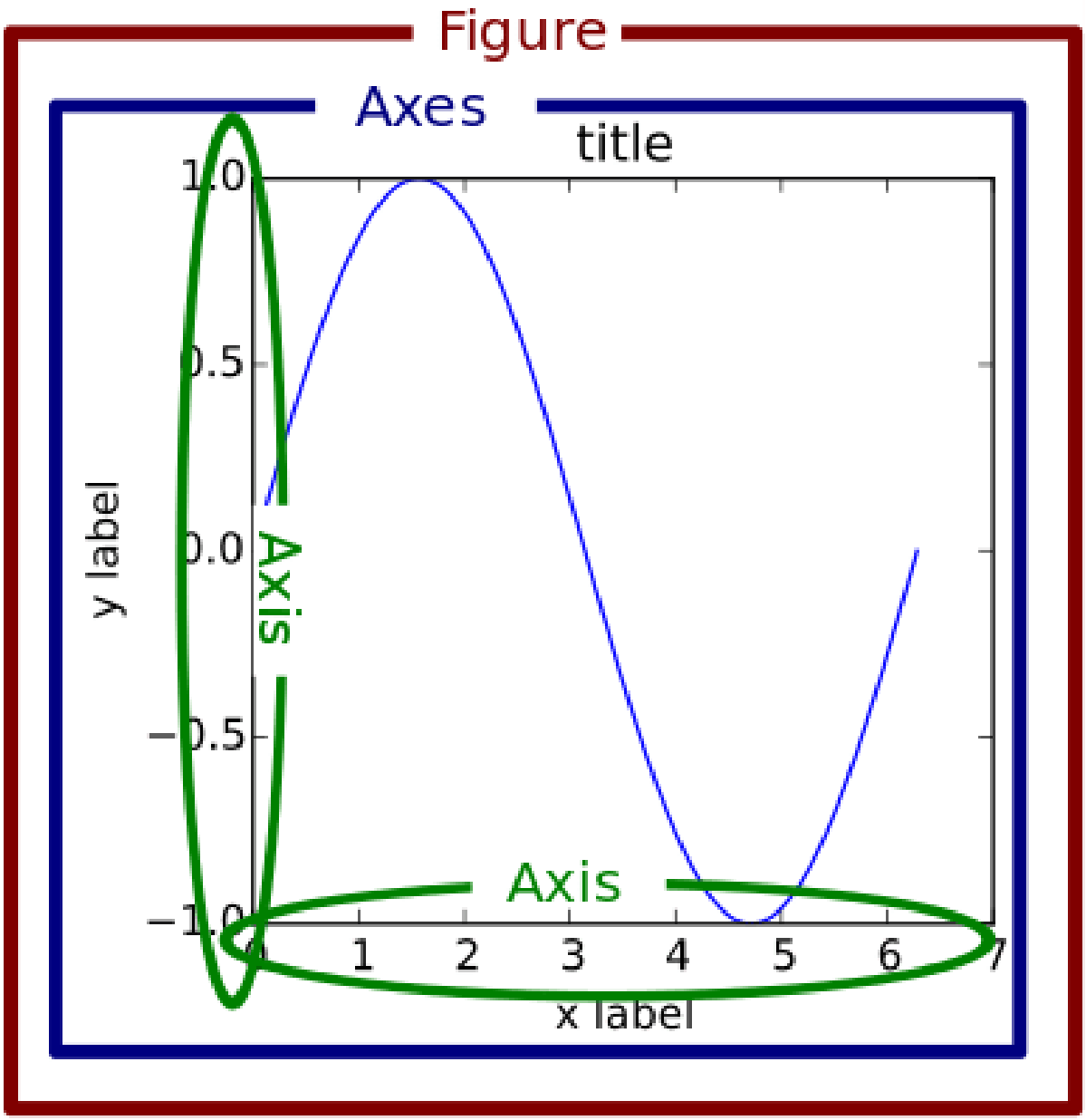
The pyplot being the sub-package within Matplotlib provides the common charting functionality.

In [ ]:
```
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

The style package adds support for easy-to-switch plotting `"styles"` .

There are a number of pre-defined styles provided by Matplotlib. For example, there's a pre-defined style called `"ggplot"` , which emulates the aesthetics of ggplot (a popular plotting package for R).

Anatomy of a Plot

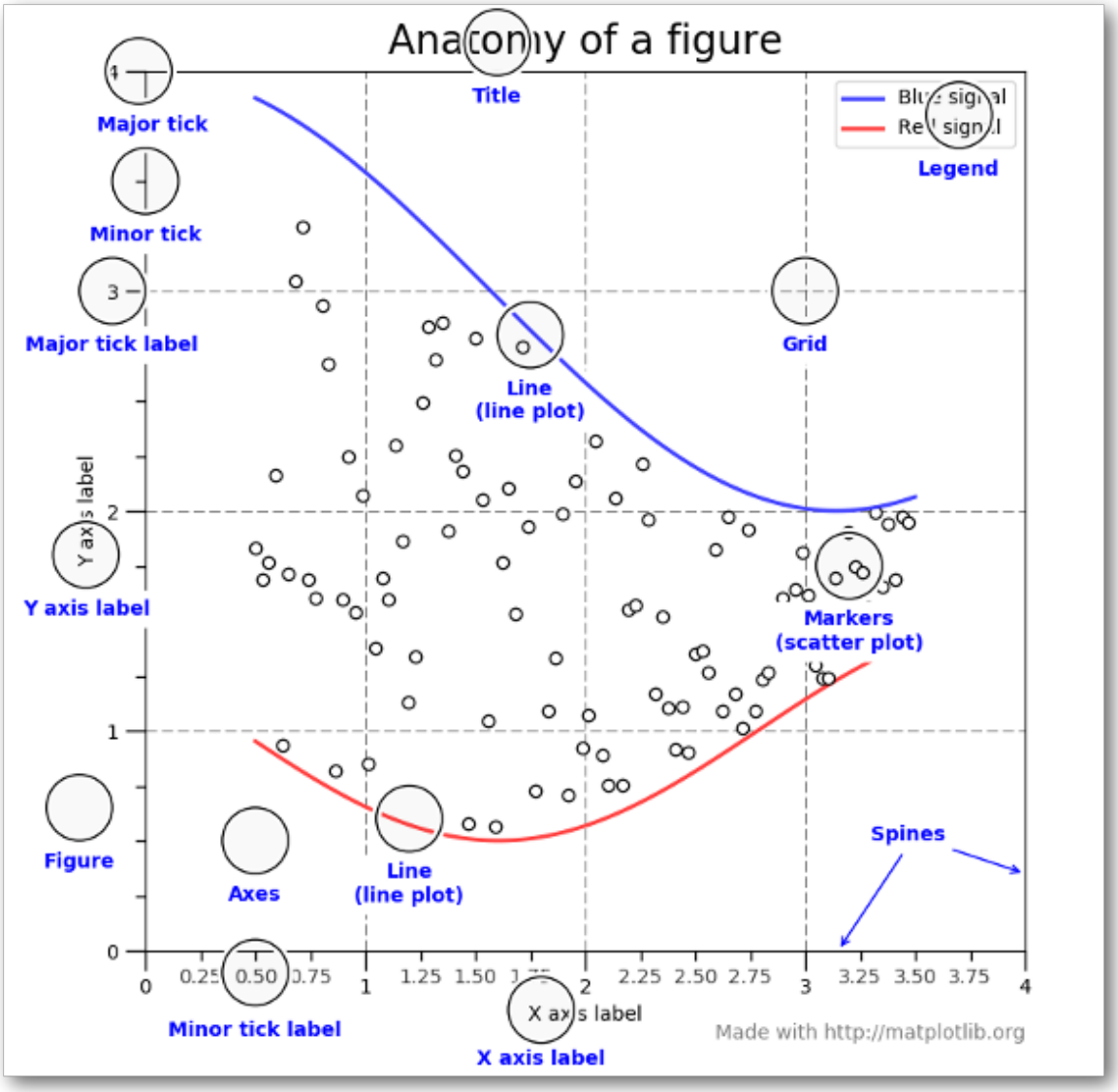


Python Matplotlib allows creating a wide variety of plots and graphs.
</br> Matplotlib is a large project and can seem daunting at first. However, we will start learning the components and it should feel much smaller and approachable.
</br>

Different sources use `'plot'` to mean different things. So let us begin by defining specific terminology used across the domain.

`Figure` is the top-level container in the hierarchy. It is the overall window where everything is drawn. We can have multiple independent figures, and each figure can have multiple `Axes.` It can be created using the `figure` method of pyplot module. </br>

`Axes` is where the plotting occurs. The axes are effectively the area that we plot data on. Each Axes has an X-Axis and a Y-Axis.



Two Approaches for creating Plots

1. Object Oriented Approach

It will take more time to code but you'll have full control of your figure.

The idea is that you create a `'figure'` object, which you can think of it as a bounding box of the whole visualization you're going to build, and one or more `'axes'` object, which are subplots of the visualization, (Don't ask me why these subplots called `'axes'` . The name just sucks...) and the subplots can be manipulated through the methods of these 'axes' objects.

All plotting is done with respect to an `Axes` . An `Axes` is made up of `Axis` objects and many other things.

An `Axes` object must belong to a `Figure` .

Most commands that we will ever issue in Python mathplotlib will be with respect to this Axes object.

Typically, we will set up a Figure, and then add Axes on to it.

In [ ]:
```
fig = plt.figure()
```

```
<Figure size 432x288 with 0 Axes>
```

Upon running the above example, nothing happens really.

It only creates a figure of size 432 x 288 with 0 Axes.

Also, Python Matplotlib will not show anything until told to do so.

Python will wait for a call to show method to display the plot.

This is because we might want to add some extra features to the plot before displaying it, such as title and label customization.

Hence, we need to call plt.show() method to show the figure as shown below:

In [ ]:
```
plt.show()
```

As there is nothing to plot, there will be no output.

Axes

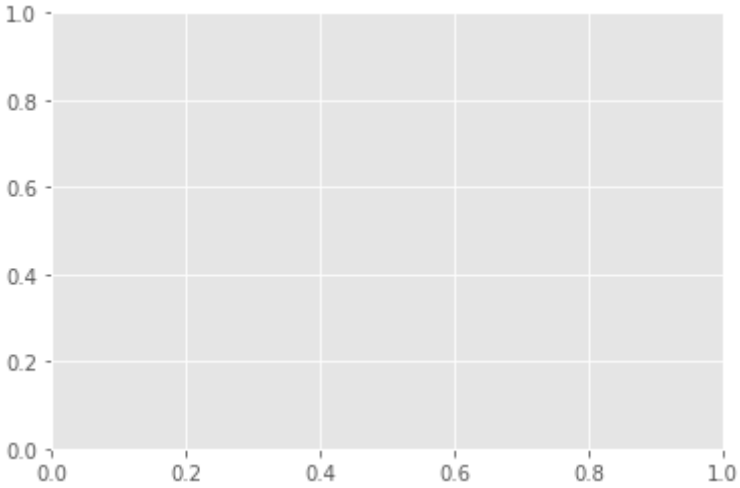
All plotting is done with respect to an `Axes`. An `Axes` is made up of `Axis` objects and many other things.

An `Axes` object must belong to a `Figure`.

Most commands that we will ever issue in Python mathplotlib will be with respect to this Axes object. Typically, we will set up a Figure, and then add Axes on to it. In most cases, we find that adding a subplot fits our need perfectly. A subplot is an axes on a grid system.

`add_subplot` method adds an Axes to the figure as part of a subplot arrangement.

In [ ]:
```
# Creating figure
fig = plt.figure()

# Creating subplot
ax = fig.add_subplot(111) # Subplot with 1 row and 1 column at the index 1
plt.show()
```



The above code adds a single plot to the figure `fig` with the help of `add_subplot()` method.

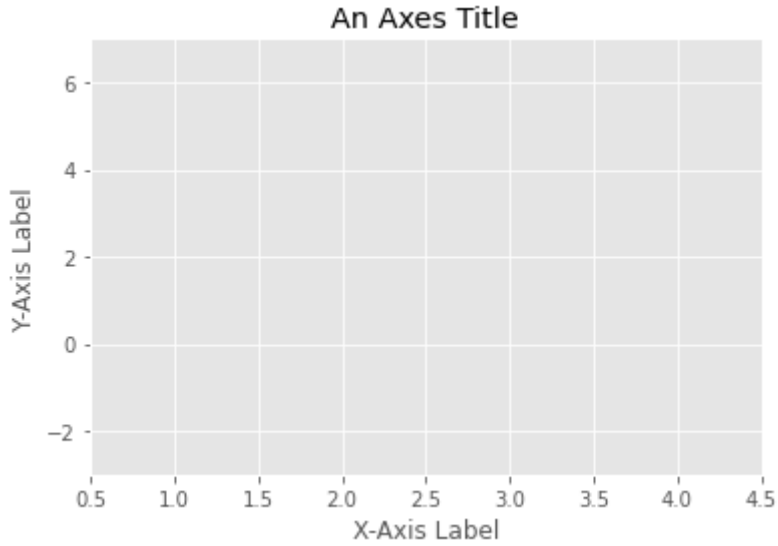
The output we get is a blank plot with axes ranging from 0 to 1 as shown above.

In Python matplotlib, we can customize the plot using a few more built-in methods.

Let us add the title, X-axis label, Y-axis label, and set limit range on both axes. This is illustrated in the below code snippet.

In [ ]:

```
fig = plt.figure()

# Creating subplot/axes
ax = fig.add_subplot(111)

# Setting axes/plot title
ax.set_title('An Axes Title')

# Setting X-axis and Y-axis limits
ax.set_xlim([0.5, 4.5])
ax.set_ylim([-3, 7])

# Setting X-axis and Y-axis labels
ax.set_ylabel('Y-Axis Label')
ax.set_xlabel('X-Axis Label')

# Showing the plot
plt.show()
```



While we are on the topic, we can control the size of the figure through the figsize argument, which expects a tuple of (width, height) in inches.

In [ ]:
```
fig = plt.figure(figsize=(8, 4))
plt.show()
```

```
<Figure size 576x288 with 0 Axes>
```

---

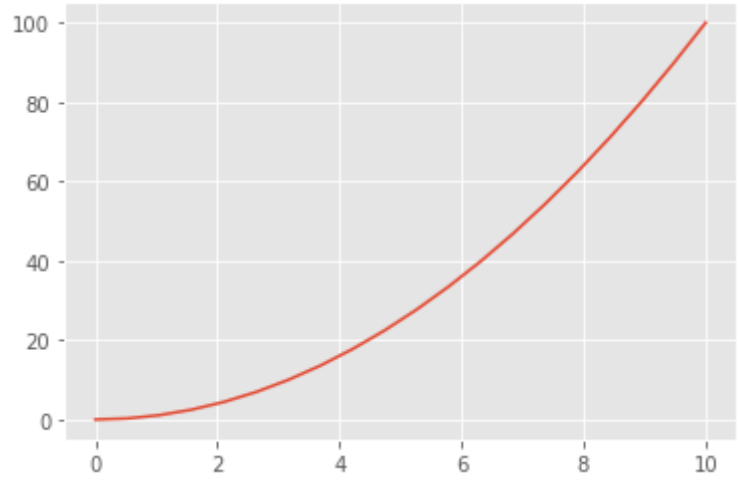
2. State Based Approach

Which is good for creating easy plots (you call a bunch of plt.somefunction() to plot each component in the graph), but you don't have too much control of the graph

In [ ]:
```
# We will begin by importing Matplotlib using:
import matplotlib.pyplot as plt
```

In [ ]:
```
import numpy as np
x = np.linspace(0, 10, 20)
y = x ** 2
plt.plot(x, y)
```

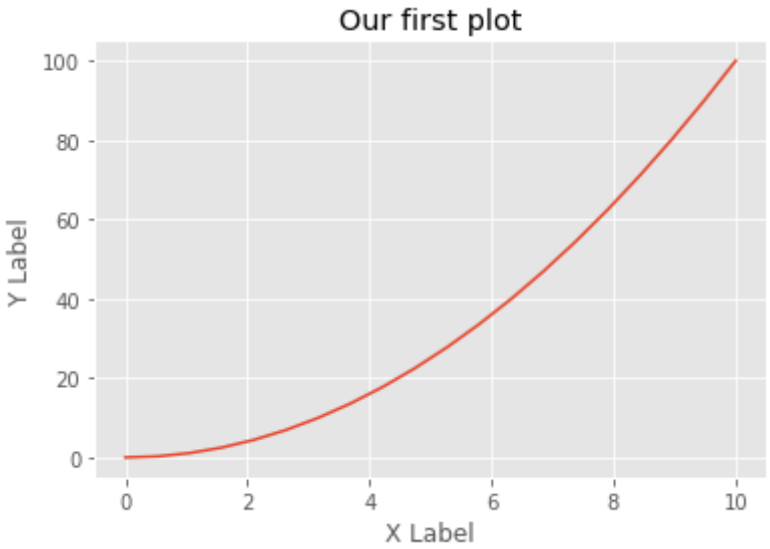
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1c55ce0e790>]
```



In [ ]:
```

```

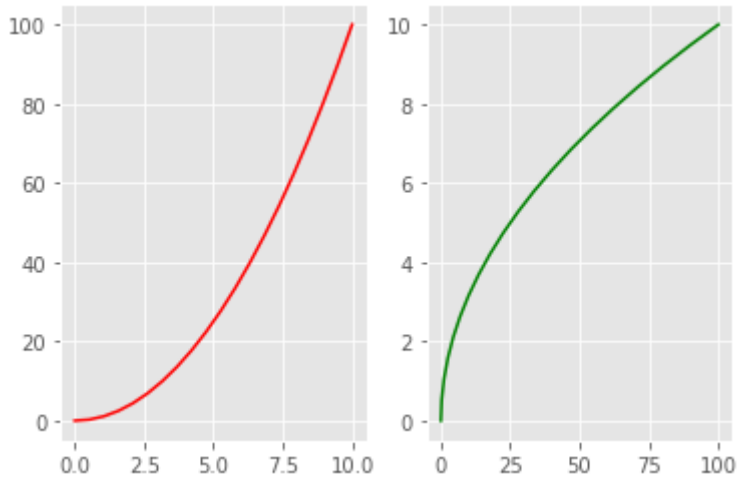
```
plt.plot(x, y)
plt.title('Our first plot')
plt.xlabel('X Label')
plt.ylabel('Y Label')
```

Out[ ]:
```
Text(0, 0.5, 'Y Label')
```



In [ ]:
```
plt.subplot(1, 2, 1)
plt.plot(x, y, 'red')
plt.subplot(1, 2, 2)
plt.plot(y, x, 'green')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1c55cf071f0>]
```



This `.subplot()` method takes in three parameters, namely:

`nrows:` the number of rows the `Figure` should have.

</br> `ncols:` the number of columns the `Figure` should have.

</br> `plot_number:` which refers to a specific plot in the `Figure` .

Back to Object-oriented approach for Matplotlib

Anytime we see something like below:

</br>

```
fig = plt.figure()
ax = fig.add_subplot(111) </br>
```

can be replaced with the following:
</br>

```
fig, ax = plt.subplots()
```

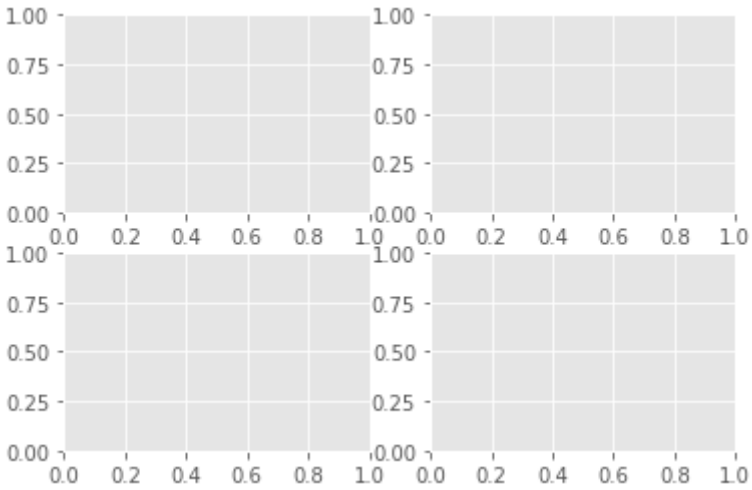
Both versions of code produce the same output.

Multiple Axes

A figure can have more than one Axes on it. </br>

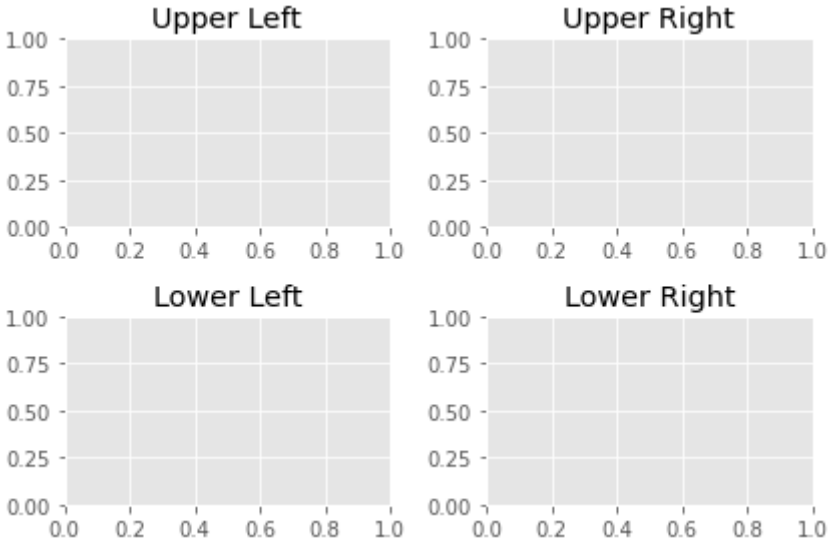
In Python matplotlib, the easiest way is to use `plt.subplots()` call to create a figure and add the axes to it automatically. Axes will be on a regular grid system. For example,

In [ ]:
```
# Creating subplots with 2 rows and 2 columns
fig, axes = plt.subplots(nrows=2, ncols=2)
plt.show()
```



Upon running the above code, Python Matplotlib would generate a figure with four subplots added arranged in two rows and two columns as shown above.

In [ ]:
```
# Create a figure with four subplots and shared axes
fig, axes = plt.subplots(nrows=2, ncols=2)
axes[0, 0].set_title('Upper Left')
axes[0, 1].set_title('Upper Right')
axes[1, 0].set_title('Lower Left')
axes[1, 1].set_title('Lower Right')
plt.tight_layout()
plt.show()
```



The above code created a figure with four subplots and shared X and Y axes.

Axes are shared among subplots in row-wise and column-wise manner.

We then set a title to each subplot using the set method for each subplot.

Subplots are arranged in a clockwise fashion with each subplot having a unique index.

Line Plot

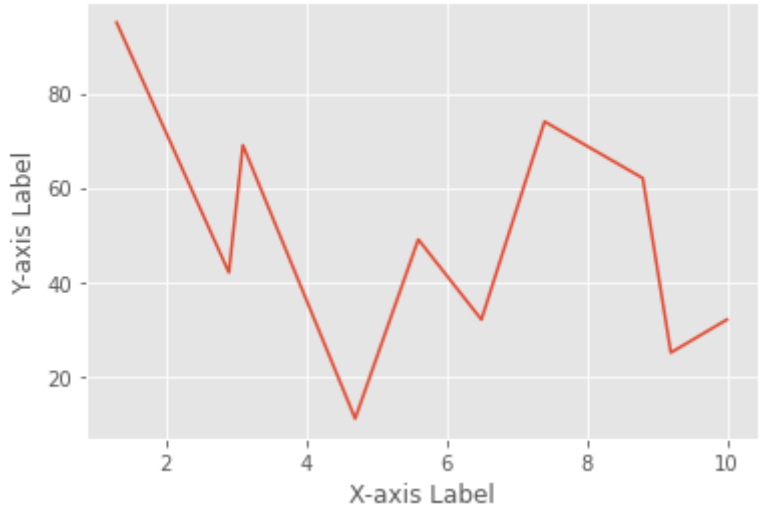
In Python matplotlib, a line plot can be plotted using the plot method.

It plots Y versus X as lines and/or markers.

Below we discuss a few scenarios for plotting line.

In [ ]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

# Defining coordinates to be plotted on X and Y axes respectively
x = [1.3, 2.9, 3.1, 4.7, 5.6, 6.5, 7.4, 8.8, 9.2, 10]
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]

# Plot lists 'x' and 'y'
ax.plot(x, y)

# Plot axes labels and show the plot
ax.set_xlabel('X-axis Label')
ax.set_ylabel('Y-axis Label')
plt.show()
```



The call to plot takes minimal arguments possible, i.e. values for Y-axis only.

In such a case, Python Matplotlib will implicitly consider the index of elements in list y as the input to the X-axis as demonstrated in the below example:

In [ ]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

# Defining 'y' coordinates
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]

# Plot list 'y'
ax.plot(y)

# Plot axes labels and show the plot
ax.set_xlabel('Index Values')
ax.set_ylabel('Elements in List Y')
plt.show()
```

Here, we define a list called y that contains values to be plotted on Y-axis.

The plots created above uses the default line style and color.

The optional parameter fmt in the plot method is a convenient way for defining basic formatting like color, marker, and line-style. It is a shortcut string notation consisting of color, marker, and

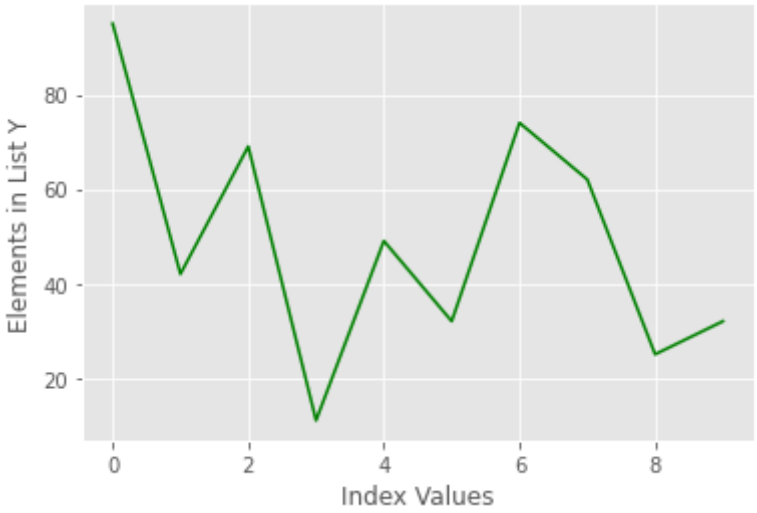
line:

```
fmt = '[color][marker][line]'
```

Each of them is optional. If not provided, the value from the style cycle is used. We use this notation in the below example to change the line color:

In [ ]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

# Plot line with green color
ax.plot(y, 'g')

# Plot axes labels and show the plot
ax.set_xlabel('Index Values')
ax.set_ylabel('Elements in List Y')
plt.show()
```



Following the fmt string notation, we changed the color of a line to green using the character **g** which refers to the line color.

Markers are added using the same notation as shown below:

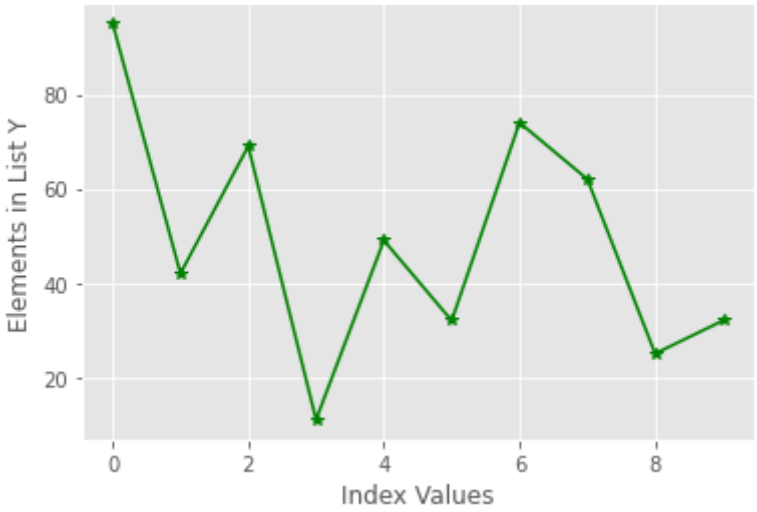
In [ ]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

# Plot continuous green line with circle markers
plt.plot(y, 'go-')

# Plot axes labels and show the plot
plt.xlabel('Index Values')
plt.ylabel('Elements in List Y')
plt.show()
```

Here, the fmt parameters **g** refers to the green color, **o** refers to circle markers and **-** refers to a continuous line to be plotted.

It is possible to change marker style by tweaking marker parameter in the fmt string as shown below:

In [ ]:
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

# Plot continuous green line with asterisk markers
ax.plot(y, 'g*-')

# Plot axes labels and show the plot
ax.set_xlabel('Index Values')
ax.set_ylabel('Elements in List Y')
plt.show()
```



In the above plots, the line and markers share the same color, i.e. green specified by the fmt string.

If we are to plot line and markers with different colors, we can use multiple plot methods to achieve the same.

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

```
fig, ax = plt.subplots()

# Plot list 'y'
ax.plot(y, 'g')

# Plot red circle markers
ax.plot(y, 'r*')

# Plot axes labels and show the plot
ax.set_xlabel('Index Values')
ax.set_ylabel('Elements in List Y')
plt.show()
```

The above code plots line along with red circle markers.

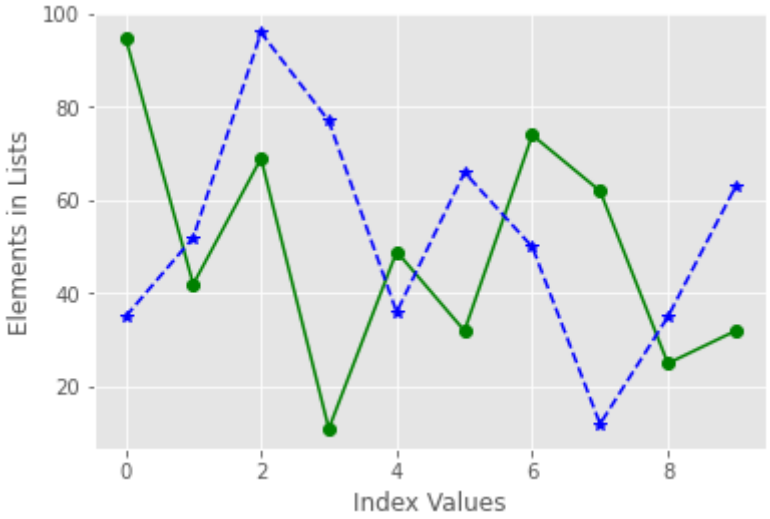
Here, we first plot the line with the default style and then attempt to plot markers with attributes **r** referring to red color and ***** referring to asterisk.

On the same lines, we can plot multiple sets of data using the same technique.

The example given below plots two lists on the same plot.

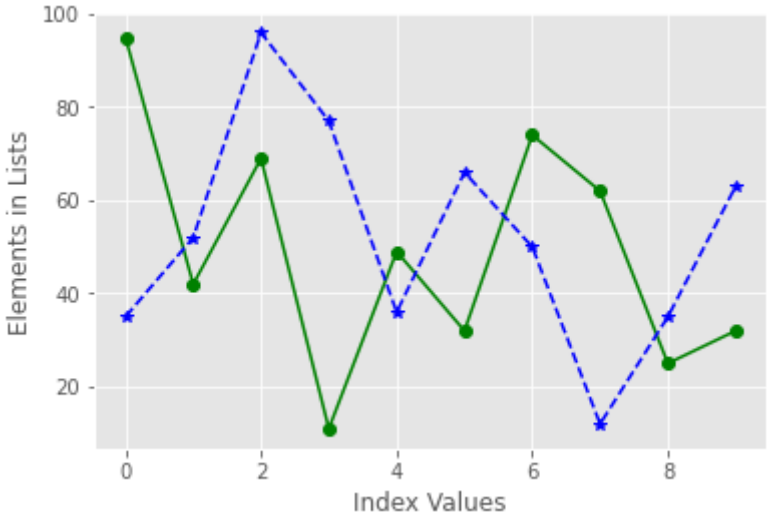
In [ ]:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

# Define two lists
y1 = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]
y2 = [35, 52, 96, 77, 36, 66, 50, 12, 35, 63]

# Plot lists and show them
ax.plot(y1, 'go-')
ax.plot(y2, 'b*--')

# Plot axes labels and show the plot
ax.set_xlabel('Index Values')
ax.set_ylabel('Elements in Lists')
plt.show()
```



We can achieve the same result as shown above using the different technique as shown below:

In [ ]:
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

# Plot lists and show them
ax.plot(y1, 'go-', y2, 'b*--')

# Plot axes labels and show the plot
ax.set_xlabel('Index Values')
ax.set_ylabel('Elements in Lists')
plt.show()
```



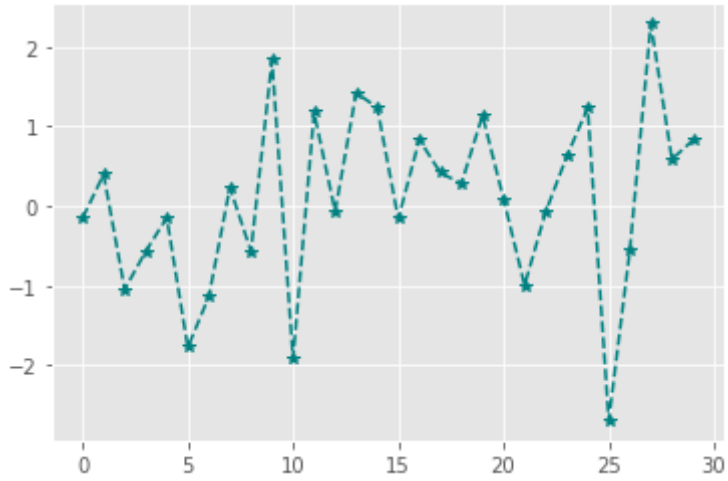
Essentially, the plot method makes it very easy to plot sequential data structure such as list, NumPy arrays, pandas series, etc.

Similar to plotting lists, we can plot NumPy arrays directly via the plot method.

Let us plot NumPy one dimensional array.

In [ ]:
```
# Importing NumPy library
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
```

```
# Drawing 30 samples from a standard normal distribution into an array 'arr'
arr = np.random.normal(size=30)

# Plotting 'arr' with dashed line-style and * markers
ax.plot(arr, color='teal', marker='*', linestyle='dashed')
plt.show()
```



In the above example, we draw thirty samples from a normal distribution into an array arr which in turn gets plotted in dashed line along with asterisk markers.

Plotting two-dimensional arrays follows the same pattern. We provide a 2-D array to a plot method to plot it.

In [ ]:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

# Creating a two dimensional array 'arr_2d' with 40 samples and shape of (20, 2)
arr_2d = np.random.normal(size=40).reshape(20, 2)

# Plotting the array
ax.plot(arr_2d)
plt.show()
```

Let us now move our focus to plot pandas data structures.

The pandas library use the standard convention as Python matplotlib for plotting directly from its data structures.

The pandas also provide a plot method which is equivalent to the one provided by Python matplotlib.

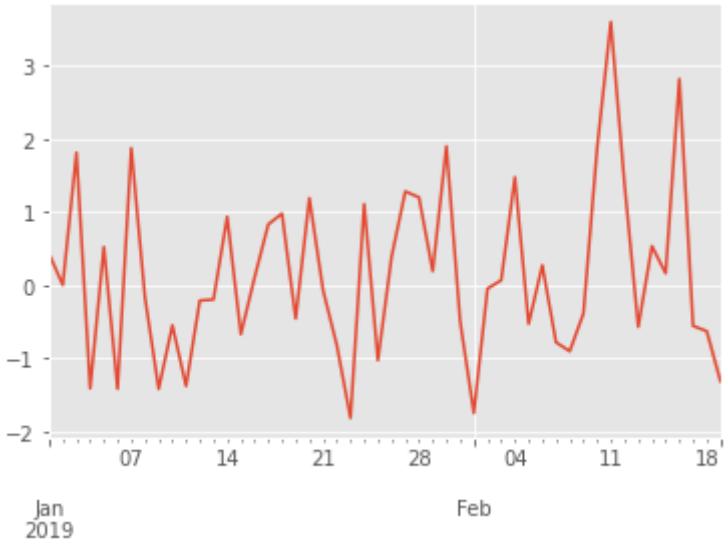
Hence, the plot method can be called directly from pandas Series and DataFrame objects.

The plot method on Series and DataFrame is just a simple wrapper around plt.plot().

The below example illustrates plotting pandas Series object:

In [ ]:
```
# Importing necessary libraries
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt

fig, ax = plt.subplots()

# Creating pandas Series with 50 samples drawn from normal distribution
ts = pd.Series(np.random.normal(size=50),
               index=pd.date_range(start='1/1/2019', periods=50))

# Plotting pandas Series
ts.plot()
```

Out[ ]:
```
<AxesSubplot:>
```



In the above example, we call the plot method directly on pandas Series object `ts` .

Alternatively, we could have called `plt.plot(ts)` .

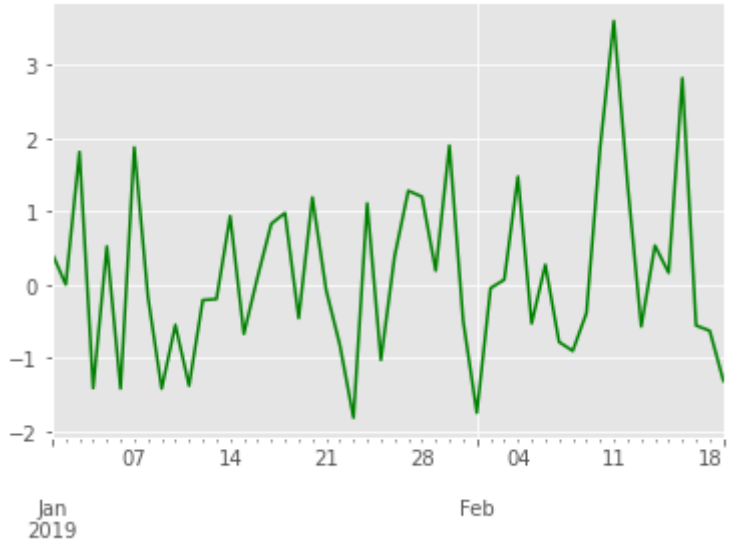
Calling ts.plot() is equivalent to calling plt.plot(ts) and both calls would result in almost the same output as shown above.

Additionally, the plot() method on pandas object supports almost every attribute that plt.plot() supports for formatting.

For example, calling the plot method on pandas objects with a color attribute would result in a plot with color mentioned by its value. This is shown below:

In [ ]:
```
# Plotting pandas Series in green color
ts.plot(color='green')
```

Out[ ]:
```
<AxesSubplot:>
```



Scatter Plot

In Python matplotlib, scatter plots are used to visualize the relationship between two different data sets.

Python Matplotlib provides the scatter method within pyplot sub-module using which scatter plots can be generated.

`plt.scatter` generates a scatter plot of y vs x with varying marker size and/or color.

The x and y parameters are data positions and it can be array-like sequential data structures.

There are some instances where we have data in the format that lets us access particular variables with string.

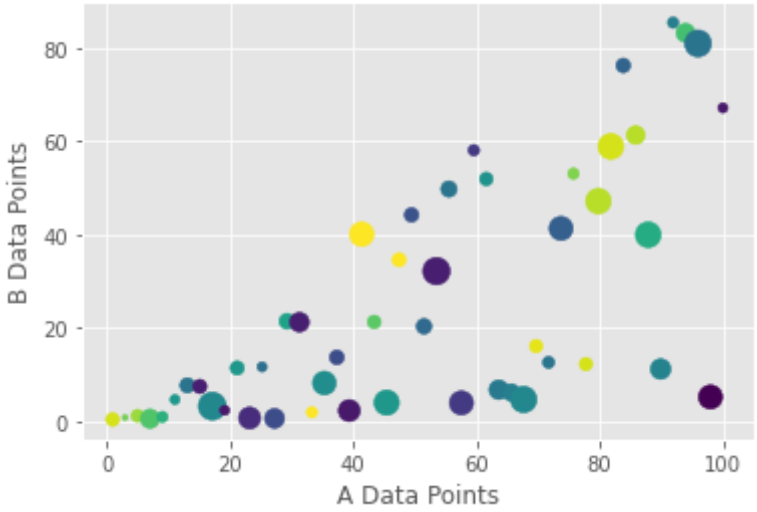
For example, Python dictionary or pandas dataframe.

Python Matplotlib allows us to provide such an object with the data keyword argument to the scatter method to directly plot from it.

The following example illustrates this using a dictionary.

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

fig, ax = plt.subplots()
```

```
# Creating a dictionary with three key-value pairs
dictionary = {'a': np.linspace(1, 100, 50),
              'c': np.random.randint(0, 50, 50),
              'd': np.abs(np.random.randn(50)) * 100}

# Creating a new dictionary key-value pair
dictionary['b'] = dictionary['a'] * np.random.rand(50)

# Plotting a scatter plot using argument 'data'
ax.scatter('a', 'b', c='c', s='d', data=dictionary)

# Labeling the plot and showing it
ax.set_xlabel('A Data Points')
ax.set_ylabel('B Data Points')
plt.show()
```



In the above code, we created a dictionary with four key-value pairs.

Values in key a and b contain fifty random values to be plotted on a scatter plot.

Key c contains fifty random integers and key d contains fifty positive floats which represent color and size respectively for each scatter data point.

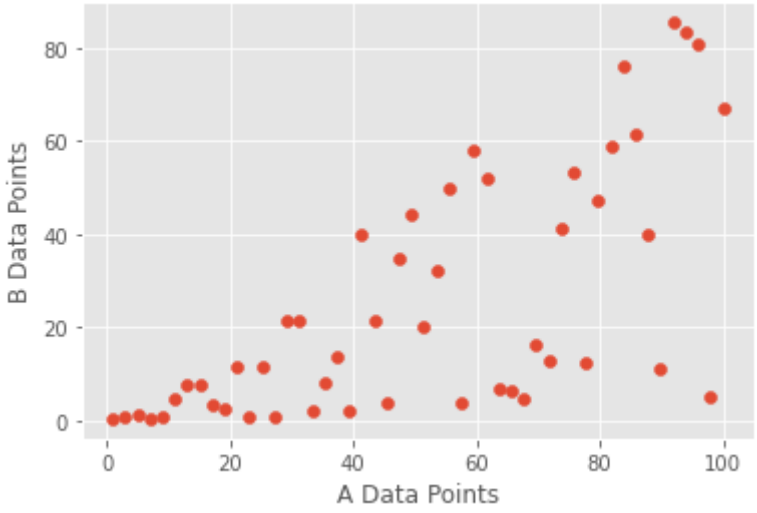
Then, a call to plt.scatter is made along with all keys and the dictionary as the value to data.

The argument c within the call refers to color to be used and the argument s represents the size of a data point.

These arguments c and s are optional. A simple scatter plot with the same color and size gets plotted when we omit these optional arguments as shown in the following example:

In [ ]:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

# Creating a scatter plot without color and same
ax.scatter(dictionary['a'], dictionary['b'])

# Labeling the plot and showing it
ax.set_xlabel('A Data Points')
ax.set_ylabel('B Data Points')
plt.show()
```



To better understand the working of scatter plots, let us resort to our old friends: lists x and y that we defined earlier while learning line plots and build scatter plots on them. To refresh our memory, we re-define the same lists below:

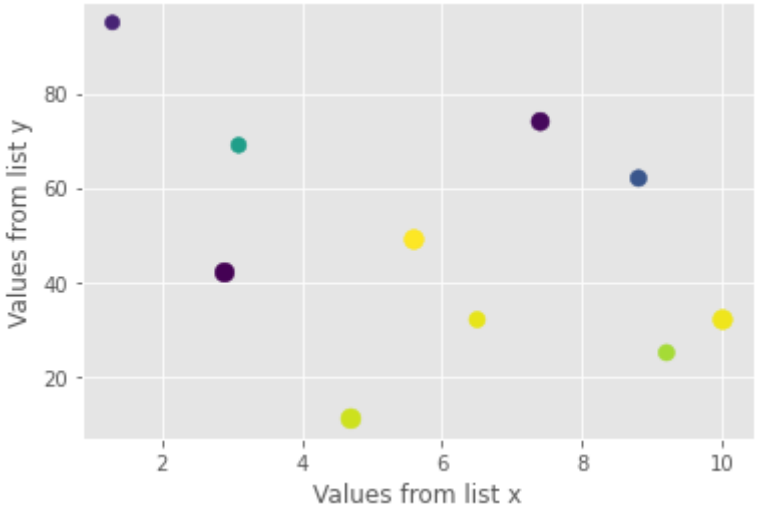
In [ ]:
```
# Data points for scatter plot
x = [1.3, 2.9, 3.1, 4.7, 5.6, 6.5, 7.4, 8.8, 9.2, 10]
y = [95, 42, 69, 11, 49, 32, 74, 62, 25, 32]
```

In addition to these lists, we would be defining two more NumPy arrays color and size which determines the color and size respectively of each data point while plotting the scatter plot.

In [ ]:
```
# Arrays which defines color and size of each data point
color = np.random.rand(10)
size = np.random.randint(50, 100, 10)
```

Now that we have data points ready, we can plot a scatter plot out of them as below:

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

fig, ax = plt.subplots()

# Creating a scatter plot
ax.scatter(x, y, c=color, s=size)

# Labeling the plot and showing it
ax.set_xlabel('Values from list x')
ax.set_ylabel('Values from list y')
plt.show()
```



The scatter plot would contain data points each with different color and size (as they are randomly generated). The output is shown below:

In the financial domain, scatter plots are widely used to determine the relations between two data sets visually.

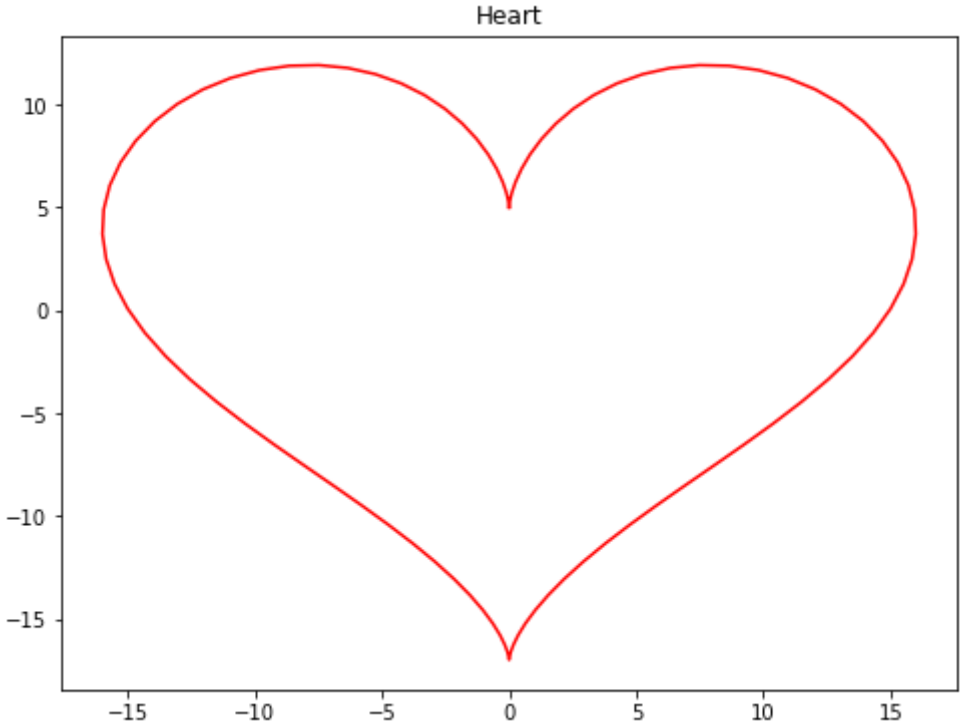
Let's plot a scatter plot of AdjOpen and AdjClose prices of AAPL stock that we have in pandas dataframe data.

We can directly use the plot method on the dataframe to plot scatter plots even just like line plots.

However, we need to specify that we are interested in plotting a scatter plot using the argument `kind='scatter'` as shown below:

In [ ]:

```
import numpy as np
from matplotlib import pyplot as plt

fig, ax = plt.subplots(figsize=(8, 6))

# Creating equally spaced 100 data in range 0 to 2*pi
theta = np.linspace(0, 2 * np.pi, 100)

# Generating x and y data
x = 16 * (np.sin(theta) ** 3)
y = 13 * np.cos(theta) - 5 * np.cos(2*theta) - 2 * \
    np.cos(3*theta) - np.cos(4*theta)

# Plotting
ax.plot(x, y, color='red')
ax.set_title('Heart')
plt.show()
```



In [ ]: