Introduction to Python.

- Python is an open-source, interpreted, high-level, general-purpose programming language.
- Python's design philosophy emphasizes code readability with its notable use of significant whitespace.
- Python is probably the easiest-to-learn and nicest-to-use programming language in widespread use.
- Python is a very expressive language, which means that we can usually write far fewer lines of Python code than would be required for an equivalent application written in, say, C++ or Java.
- Python can be used to program in procedural, object-oriented, and to a lesser extent, in a functional style.
- Python was conceived in the late 1980s as a successor to the ABC language.
- Python was created by Guido van Rossum and first released in 1991.
- Python 2.0, released in 2000, introduced features like list comprehensions and a garbage collection system with reference counting.
- Python 3.0 was released in 2008 and the current version of python is 3.8.3 (as of June-2020).
- The Python 2 language was officially discontinued in 2020.

Advantages of Python

- Easy Syntax
  - Python's syntax is easy to learn.
- Readability
  - Python's syntax is very clear, so it is easy to understand program code.
  - Python is often referred to as "executable pseudo-code" because its syntax mostly follows the conventions used by programmers to outline their ideas without the formal verbosity of code in most programming languages.
  - In other words, the syntax of Python is almost identical to the simplified "pseudo-code" used by many programmers to prototype and describe their solution to other programmers.
  - Thus Python can be used to prototype and test code which is later to be implemented in other programming languages.
- High-Level Language
  - Python looks more like a readable, human language than like a low-level language.
  - This gives you the ability to program at a faster rate.
- It's Free & Open Source
  - Python is both free and open-source.
  - The Python Software Foundation distributes pre-made binaries that are freely available for use on all major operating systems called CPython.

  - You can get CPython's source-code, too. Plus, you can modify the source code and distribute it as allowed by CPython's license.
- Cross-platform
  - Python runs on all major operating systems like Microsoft Windows, Linux, and Mac OS X.
- Widely Supported
  - Python has an active support community with many websites, mailing lists, and USENET "netnews" groups that attract a large number of knowledgeable and helpful contributors.
- It's Safe
  - Python doesn't have pointers like other C-based languages, making it much more reliable.
  - Along with that, errors never pass silently unless they're explicitly silenced.
  - This allows you to see and read why the program crashed and where to correct your error.
- Huge amount of additional open-source libraries
  - There are over 300 standard library modules that contain modules and classes for a wide variety of programming tasks.
  - Some libraries are listed below.
    - **matplotib** for plotting charts and graphs
    - **BeautifulSoup** for HTML parsing and XML
    - **NumPy** for scientific computing
    - **pandas** for performing data analysis
    - **SciPy** for engineering applications, science, and mathematics
    - **Scikit** for machine learning
    - **Django** for server-side web development
  - And many more...
- Extensible
  - In addition to the standard libraries, there are extensive collections of freely available add-on modules, libraries, frameworks, and tool-kits.

Installing Python

- For Windows & Mac:
  - To install python in windows you need to download installable file from https://www.python.org/downloads/
  - After downloading the installable file you need to execute the file.
- For Linux :
  - For ubuntu 16.10 or newer
    - sudo apt-get update
    - sudo apt-get install python3.8
- To verify the installation
  - Windows :

    - python --version
  - Linux:
    - **python3** --version (Linux might have python2 already installed, you can check python 2 using **python --version**)
- Alternatively, we can use anaconda distribution for the python installation
  - http://anaconda.com/downloads
  - Anaconda comes with many useful inbuilt libraries.

Hello World (first program) in Python

- To write Python programs, we can use any text editors or IDE (Integrated Development Environment).
- Create new file in editor, save it as first**.py** (Extensions for python programs will be .py)
- Write below code in the first.py file:

```
print("Hello World from python")
```

  Note: Python lines do **not** end with a semicolon (**;**) like most other programming languages.
- To run the python file open command prompt and change directory to where your python file is

```
D:\>cd B.E
D:\B.E>cd 5th
D:\B.E\5th>cd "Phython 2020"
D:\B.E\5th\Phython 2020>cd Demo
```

- Next, run python command (python filename.py)

```
D:\B.E\5th\Phython 2020\Demo>python first.py
Hello World from python
```

Data types in Python

- A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.
- Python provides various standard data types that define the storage method on each of them.
- The data types defined in Python are given below.
  - Numbers
    - Integer
    - Complex Number
    - Float
  - Boolean
  - Sequence Type
    - String
    - List
    - Tuple

- Set
- Dictionary

- **Numbers**
  - Number stores numeric values.
  - The integer, float, and complex values belong to a Python Numbers data-type.
  - Python provides the type() function to know the data-type of the variable.
  - Python creates Number objects when a number is assigned to a variable.
  - Python supports three types of numeric data.
    - **Int** - Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to **int**
    - **Float** - Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate up to 15 decimal points.
    - **complex** - A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc. (Note: imaginary part in python will be denoted with **j** suffix)

      Example:

```
a = 5
print("The type of a", type(a))

b = 40.5
print("The type of b", type(b))

c = 1+3j
print("The type of c", type(c))
print(" c is a complex number", isinstance(1+3j,complex))
```

      Output:

```
The type of a <class 'int'>
The type of b <class 'float'>
The type of c <class 'complex'>
c is complex number: True
```

- **Boolean**
  - Boolean type provides two built-in values, True and False, these values are used to determine within the given statement is true or false.
  - It is denoted by bool.
  - True can be represented by any non-zero value or 'True' whereas false can be represented by the 0 or 'False'.
  - Example:

```
# Python program to check the boolean type
print(type(True))
print(type(False))
print(false)
```

Output:

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

- **Sequence Type, Set and Dictionary** are discussed in subsequent topics

Variables in Python

- A Python variable is a reserved memory location to store values.
- Unlike other programming languages, Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- Python uses Dynamic Typing so,
  - We need not specify the data types to the variable as it will internally assign the data type to the variable according to the value assigned.
  - we can also reassign the different data types to the same variable, the variable data type will change to a new data type automatically.
  - We can check the current data type of the variable with the **type(variablename)** in-built function.
- Rules for variable name
  - Name cannot start with digit
  - Space not allowed
  - Cannot contain special character
  - Python keywords not allowed
  - **Should** be in lower case

Strings in Python

- The stringsOrdered Sequence of character such as “darshan”, ‘college’, “**રાજકોટ**” etc...
- Strings are arrays of bytes representing Unicode characters.
- A string can be represented as single, double, or triple quotes.
- String with triple Quotes allows multiple lines.
- Square brackets can be used to access elements of the string, Ex. “Darshan”[1] = a, characters can also be accessed with a reverse index like “Darshan”[-1] = n.
- String in python is immutable.
- Square brackets can be used to access elements of the string, Ex. “Darshan”[1] = a, characters can also be accessed with a reverse index like “Darshan”[-1] = n.

```
            x = "  D  a  r  s  h  a  n  "
        index =    0  1  2  3  4  5  6
Reverse index =    0 -6 -5 -4 -3 -2 -1
```

String slicing in Python.

- We know string is a sequence of individual characters, and therefore individual characters in a string can be extracted using the item access operator ([])
- Access operator ([]) is much more versatile and can be used to extract not just one item or character, but an entire slice (subsequence).
- We can get the substring in python using string slicing, we can specify start index, end index, and steps (colon-separated) to slice the string.
- Syntax :

```
x = 'our string'
subx1 = x[startindex]
subx2 = x[startindex:endindex]
subx3 = x[startindex:endindex:steps]
```

- The startindex, endindex, and steps values must all be integers, It extracts the substring from startindex till endindex (not including endindex) with steps defining the increment of index, If we specify steps to be -1 it will extract the reversed string.
- Example:

```
x = 'darshan institute of engineering and technology, rajkot,
gujarat, INDIA'
subx1 = x[0:7]
subx2 = x[49:55]
subx3 = x[66:]
subx4 = x[::2]
subx5 = x[::-1]
print(subx1)
print(subx2)
print(subx3)
print(subx4)
print(subx5)
```

Output:

```
darshan
rajkot
INDIA
drhnisiueo niern n ehooy akt uaa,IDA
AIDNI ,tarajug ,tokjar ,ygolonhcet dna gnireenigne fo etutitsni nahsrad
```

String functions in Python

Python has lots of built-in methods that you can use on strings, we are going to cover some frequently used methods for string like

- **len()** is **not a string function** but we can use this method to get the length of string.

```
x = "Darshan"
print(len(x))
```

Output:

```
7 (length of "Darshan")
```

- **count()** method will return the number of times a specified value occurs in a string.

```
x = "Darshan"
ca = x.count('a')
print(ca)
```

Output:

```
2 (occurrence of 'a' in "Darshan")
```

- **title()**, **lower()**, **upper()** will returns capitalized, lower case and upper case string respectively

```
x = "darshan Institute, rajkot"
c = x.title()
l = x.lower()
u = x.upper()
print(c)
print(l)
print(u)
```

Output:

```
Darshan Institute, Rajkot
darshan institute, rajkot
DARSHAN INSTITUTE, RAJKOT
```

- **istitle()**, **islower()**, **isupper()** will returns True if the given string is capitalized, lower case and upper case respectively.

```
x = 'darshan institute, rajkot'
c = x.istitle()
l = x.islower()
u = x.isupper()
print(c)
print(l)
print(u)
```

Output:

```
False
True
False
```

- **find(), rfind(), replace()**
  - **find**() method will search the string and returns the index at which the specified value is found.

```
x = 'darshan institute, rajkot, india'
f = x.find('in')
print(f)
```

Output:

```
8 (index of 'in' in x)
```

  - **rfind**() will search the string and returns the last index at which the specified value is found.

```
x = 'darshan institute, rajkot, india'
r = x.rfind('in')
print(r)
```

Output:

```
27 (last index of 'in'  in x)
```

  - **replace**() will replace str1 with str2 from our string and return the updated string

```
x = 'darshan institute, rajkot, india'
r = x.replace('india','INDIA')
print(r)
```

Output:

```
darshan institute, rajkot, INDIA
```

- **index(), rindex()**
  - **index**() method will search the string and returns the index at which the specified value is found, but if unable to find the string it will raise an exception.

```
x = 'darshan institute, rajkot, india'
f = x.index('in')
print(f)
```

Output:

```
8 (index of 'in' in x)
```

  - **rindex**() will search the string and returns the last index at which the specified value is found, but if unable to find the string it will raise an exception.

```
x = 'darshan institute, rajkot, india'
r = x.rindex('in')
print(r)
```

Output:

```
27 (last index of 'in' in x)
```

  - Note : **find**() and **index**() are almost the same, the only difference is if **find**() is unable to find then it will return -1 and if **index**() is unable to find, it will raise an exception.

- **isalpha(), isalnum(), isdecimal(), isdigit()**
  - **isalnum**() method will return true if all the characters in the string are alphanumeric (i.e either alphabets or numeric).

```
x = 'darshan123'
f = x.isalnum()
print(f)
```

Output:

```
True
```

  - **isalpha**() and **isnumeric**() will return true if all the characters in the string are only alphabets and numeric respectively.
  - **isdecimal**() will return true if all the characters in the string are decimal.
  - **Note:** str.isdigit only returns True when strings containing solely the digits 0-9, by contrast, str.isnumeric returns True if it contains any numeric characters.
- **strip(), lstrip(), rstrip()**
  - **strip**() method will remove whitespaces from both sides of the string and returns the string.
  - **rstrip**() and **lstrip**() will remove whitespaces from right and left side respectively.
- **split()** method splits a string into a list.

String Formatting with the str.format() Method

- The str.format() method provides a very flexible and powerful way of creating strings.
- The str.format() method returns a new string with the replacement fields in its string replaced by its arguments formatted suitably, for example:

```
x = '{0} institute, rajkot, INDIA'
r = x.format('darshan')
print(r)
```

Output:

```
        darshan institute, rajkot, INDIA
```

- There can be more than one argument in the format function, for example:

```
x = '{0} institute, {1}, {2}'
r = x.format('darshan', 'rajkot', 'INDIA')
print(r)
```

Output:

```
        darshan institute, rajkot, INDIA
```

- Each replacement field is identified by a field name in braces, if the field name is a simple integer it is taken to be the index position. {0} field name will be replaced by first argument, {1} field name will be replaced by the second argument, and this will continue till the last argument.

- We can use field name more than one time in the same string, for example

```
x = '{0} institute, {1}, {2}, welcome to {0}'
r = x.format('darshan', 'rajkot', 'INDIA')
print(r)
```

Output:

```
        darshan institute, rajkot, INDIA, welcome to darshan
```

- If we need to include braces inside format strings, we can do so by writing three brackets, for example:

```
x = '{{{0}}} institute'
r = x.format('darshan')
print(r)
```

Output: {darshan} institute

- If we try to concatenate a string and a number, Python will quite rightly raise a TypeError, But we can do this using str.format(), for example:

```
x = 'The price for {0} is ₹{1}'
r = x.format('iphone',50000)
print(r)
```

Output:

```
The price for iphone is ₹50000
```

- The default formatting of integers, floating-point numbers, and strings are often perfectly satisfactory, but if we want to fine control, we can easily do so by using format specifications.
- Below is the general form of a format specification

:	fill	align	sign	#	0	width	. precision	type
	Any character except }	< left > right ^ center = pad between sign and digits	+ force sign; - sign if needed; " " space or – as appropriate	Prefix ints with 0b,0o 0r	0-pad numbers	Minimum field width	Maximum field width for strings; number of decimal places for floating-point numbers.	ints b,c,d,n,o ,x,X; floats e,E,f,g, G,n,%

- String values formatting,

```
s = '{0} institute of engineering'
print(s.format('darshan')) #default formatting
```

Output:

```
darshan institute of engineering
```

```
s = '{0:25} institute of engineering'
print(s.format('darshan')) #minimum width
```

Output:

```
darshan                   institute of engineering
```

```
s = '{0:>25} institute of engineering'
print(s.format('darshan')) #right align, minimum width
```

Output:

```
                  darshan institute of engineering
```

```
s = '{0:^25} institute of engineering'
print(s.format('darshan')) #center align, minimum width
```

Output:

```
         darshan          institute of engineering
```

```
s = '{0:-^25} institute of engineering'
print(s.format('darshan')) #fill center align, minimum width
```

Output:

```
---------darshan--------- institute of engineering
```

```
s = '{0:.3} institute of engineering' #maximum width of 3
print(s.format('darshan'))
```

Output:

```
dar institute of engineering
```

- Integer values formatting,

```
s = 'amount = {0}' # default formatting
print(s.format(123456))
```

Output:

```
123456
```

```
s1 = 'amount = {0:0=10}' # 0 fill, minimum width 12 (technique 1)
print(s1.format(123456))

s2 = 'amount = {0:010}' # 0 pad, minimum width 12 (technique 2)
print(s2.format(123456))
```

Output:

```
amount = 0000123456
amount = 0000123456
```

```
s = 'amount = {0: }' # space or - sign
print(s.format(123456))
print(s.format(-123456))
```

Output: (Note: single quote is just to demonstrate the space in the positive numbers)

```
‘ 123456’
‘-123456’
```

```
s = 'amount = {0:+}' # force sign
print(s.format(123456))
print(s.format(-123456))
```

Output:

```
+123456
-123456
```

```
s1 = '{0:b} {0:o} {0:x} {0:X}'
s1 = '{0:#b} {0:#o} {0:#x} {0:#X}'
print(s1.format(12))
print(s2.format(12))
```

Output:

```
1100 14 c C
0b1100 0o14 0xc 0XC
```

- Decimal values formatting,

```
s = 'amount = {0}' # default formatting
print(s.format(12.3456789))
```

Output:

```
12.3456789
```

```
s1 = 'amount = {0:10.2e}' # exponential formatting with two precession
s2 = 'amount = {0:10.2f}' # floating formatting with two precession
print(s1.format(12345.6789))
print(s2.format(12345.6789))
```

Output:

```
amount =   1.23e+04
amount =   12345.68
```

Operators in Python

- Operators are used to performing operations on variables and values.
- We can segregate python operators in following groups,
  - Arithmetic operators
  - Assignment operators
  - Comparison operators
  - Logical operators
  - Identity operators
  - Membership operators
  - Bitwise operators
- We are going to explore some of the operators in this section.

Arithmetic Operators

Arithmetic operators can be used with numeric values or variables to perform common mathematical operations:

Operator	Description	Example	Output
+	Addition	A + B	13
-	Subtraction	A - B	7
/	Division	A / B	3.33333..33335
*	Multiplication	A * B	30
%	Modulus return the remainder	A % B	1
//	Floor division returns the quotient	A // B	3
**	Exponentiation	A ** B	10 * 10 * 10 = 1000

Note: Consider A = 10 and B = 3 in above table

Logical Operators

Logical operators can be used to merge conditional statements:

Operator	Description	Example	Output
and	Returns True if both statements are true	A > 5 **and** B < 5	True

or	Returns True if one of the statements is true	A > 5 **or** B > 5	True
not	Negate the result, returns True if the result is False	not(A>5)	False

Note: Consider A = 10 and B = 3 in above table

Identity Operators

Identity operators can be used to compare the objects, not if they are equal, but if they are the same object, with the same memory location:

Operator	Description	Example	Output
is	Returns True if both variables are the same object	A is B A is C	False True
is not	Returns True if both variables are **different** object	A is not B	True

Note: Consider A = [1,2], B = [1,2] and C = A in above table

Member Operators

Membership operators can be used to check if a sequence is presented in an object:

Operator	Description	Example	Output
in	Returns True if a sequence with the specified value is present in the object	A in B	True
not in	Returns True if a sequence with the specified value is not present in the object	A not in B	False

Note: Consider A = 2 and B = [1,2,3] in above table

Conditional statements in Python

Python supports 3 types of conditional statements,

1. if statements
2. if..else statements
3. if…elif…else statements

   python also supports nested if statements

if statement

- It is one of the powerful conditional statement.

- If statement is responsible for modifying the flow of execution of a program.
- If statement is always used with a condition.
- The condition is evaluated first before executing any statement inside the body of If.
- if statement is written using the **if** keyword followed by **condition** and **colon(:)**
- The syntax for if statement is as follows:

```
if some_condition :
   # Code to execute when condition is true
```

- Code to execute when the condition is true will be ideally written in the next line with Indentation (white space).
- Python relies on indentation to define the scope in the code (Other programming languages often use curly-brackets for this purpose).
- Python programming language assumes any non-zero and non-null values as True, and if it is either zero or null, then it is assumed as False value.
- Example:

```
x = 10

if x > 5 :
    print("X is greater than 5")
```

Output :

```
X is greater than 5
```

if else statements

- An else statement can be combined with an if statement.
- An else statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a False value.
- The else statement is an optional statement and there could be at most only one else statement following if.
- Example:

```
x = 3

if x > 5 :
    print("X is greater than 5")
else :
    print("X is less than 5")
```

Output:

```
X is less than 5
```

if elif else statements

- The elif is short for else if. It allows us to check for multiple expressions.
- If the condition for if is False, it checks the condition of the next elif block and so on.
- If all the conditions are False, the body of else is executed.
- Only one block among the several if...elif...else blocks is executed according to the condition.
- The if a block can have only one else block, but it can have multiple elif blocks.
- Example:

```
x = 10

if x > 12 :
    print("X is greater than 12")
elif x > 5 :
    print("X is greater than 5")
else :
    print("X is less than 5")
```

Output:

```
X is greater than 5
```

Looping statements in python

- There are two looping statements in python,
  - while loop
  - for loop

while loop

- While loop will continue to execute a block of code until specified condition remains True.
- For example
  - while hungry keep eating
  - while internet pack available, keep watching videos
  - etc..
- Syntax:

```
while some_condition :
    # Code to execute in loop
```

- Example:

```
x = 0
while x < 3 :
    print(x)
    x += 1    # x++ is invalid in python
```

Output:

```
0
1
2
```

- We can use an else statement with a while loop if we want to execute some block when the condition specified in the while statement is False.
- Example:

```
x = 5
while x < 3 :
    print(x)
    x += 1    # x++ is invalid in python
else :
    print("X is greater than 3")
```

Output:

```
X is greater than 3
```

For loop

- Many objects in python are iterable, meaning we can iterate over every element in the object, such as every element from the List, every character from the string, etc..
- Syntax:

```
for temp_item in iterable_object :
    # Code to execute for each object in iterable
```

- Example 1 (number list):

```
my_list = [1, 2, 3, 4]
for list_item in my_list :
    print(list_item)
```

Output:

```
1
2
3
4
```

- Example 2 (number list from the range):
  range() function will create a list from start till (not including) the value specified as the argument.

```
my_list = range(0,5)
for list_item in my_list :
    print(list_item)
```

Output:

```
0
1
2
3
4
```

- Example 3 (string list):

```
my_list = ["darshan","institute","rajkot","gujarat"]
for list_item in my_list :
    print(list_item)
```

Output:

```
darshan
institute
rajkot
gujarat
```

Break, continue and pass keywords

- **break** statement will breaks out of the closest enclosing loop, for example:

```
for temp in range(5) :
    if temp == 2 :
        break
    print(temp)
```

Output:

```
0
1
```

- **continue** statement will send execution control to top of the current closest enclosing loop, for example:

```
for temp in range(5) :
    if temp == 2 :
        continue
    print(temp)
```

Output:

```
0
1
3
4
```

- **pass** does nothing at all !!!!!, it can be used as a placeholder in conditions where we don't write anything, for example:

```
for temp in range(5) :
    pass
```

Functions in Python

- Creating clean repeatable code is a key part of becoming an effective programmer.
- In Python, a function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks, as our program grows larger and larger, functions make it more organized and manageable.
- A function is a block of code that only runs when it is called.
- Python provides built-in functions like print(), etc. but we can also create our functions, these functions are called **user-defined functions**.
- We can pass arguments/parameters into function and function can return some value as a result.
- In python user-defined functions can be created using def keyword followed by the name of the function and then its argument in the round brackets.
- General Syntax for the UDF in Python:

```
def function_name(arguments) :
    ''' docstring ''' #optional
    #code to execute when function is called
```

- Example:

```
def seperator() :
    print('================================')

print("hello world")
seperator()
print("from darshan college")
seperator()
print("rajkot")
```

Output:

```
hello world
================================
from darshan college
================================
rajkot
```

Arguments in Python Functions

- Data can be passed into functions as parameters/arguments.
- We can specify any number of arguments in the function inside parenthesis just after the function name as comma-separated values.

- Example (passing parameter)

```
def seperator(type) :
    if(type=='=') :
        print('==============================')
    elif(type=='*') :
        print('******************************')
    else :
        print('&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&')

print("hello world")
seperator('=')
print("from darshan college")
seperator('*')
print("rajkot")
```

Output:

```
hello world
==============================
from darshan college
******************************
rajkot
```

- All parameters in the Python language are passed by reference, which means if we change what parameter refers it will change the actual parameter value (if the parameter is mutable), below example uses List which is mutable. (we will discuss List later in this chapter)

```
def setName(name) :
    name[0] = "arjun bala"

name = ["Default"]
setName(name)
print(name)
```

Output:

```
['arjun bala']
```

- We can use **keyword arguments** in Python which gives freedom to specify arguments in any sequence.
- Example (keyword arguments):

```
def collegeDetails(collegeName, state) :
    print(collegeName + " is located in "+ state)

collegeDetails(state='Gujarat', collegeName='Darshan')
```

Output:

```
Darshan is located in Gujarat
```

- It is also possible to specify the **default value** to some arguments in the Python functions, such arguments are optional to pass when calling the function.
- Example (Default value):

```
def collegeDetails(collegeName, state='Gujarat') :
    print(collegeName + " is located in "+ state)

collegeDetails(collegeName='Darshan')
```

Output:

```
Darshan is located in Gujarat
```

- We can specify the dynamic argument using **Arbitrary Arguments (*args)** in python, which can help to accept any number of arguments while calling the function.
- Example (Arbitrary Arguments):

```
def collegeList(*colleges):
    print(colleges[0])
    # we can also loop through all the arguments
    # for c in colleges :
    #    print(c)

collegeList('darshan','nirma','vvp')
```

Output:

```
darshan
```

- We can also specify the dynamic keyword argument using **Arbitrary Keyword Arguments (**kwargs)** in python, which can help to accept any number of keyword arguments while calling the function.
- Example (Arbitrary Keyword Arguments):

```
def collegeDetail(**college) :
    print('College Name is '+college['name'])

collegeDetail(city='rajkot', state='gujarat', name='Darshan')
```

Output:

```
College Name is Darshan
```

- The **return** statement exits a function and passes the caller of the function of some value (result).

- Example (return statement):

```
def addition(n1,n2):
    return n1 + n2

ans = addition(10,5)
print(ans)
```

Output:

```
15
```

Data Structures in Python

- There are many types of Data Structures available in python, in this section we will discuss 4 inbuilt Data Structures,
  - **List**: Ordered Mutable sequence of objects
  - **Set**: Unordered collection of unique objects
  - **Tuple**: Ordered Immutable sequence of objects
  - **Dictionary**: Unordered key:value pair of objects

List

- A **List** is a data structure that holds an **ordered** collection of items i.e. you can store a sequence of items in a list.
- Once we have created a list, we can add, remove or search for items in the list.
- Since we can add and remove items, we say that a list is a **mutable** data type i.e. this type can be altered.
- A List can hold duplicate values and can be used similar to Arrays.
- A List will be represented with the Square Brackets '[**list**]'
- Example:

```
my_list = ['darshan', 'institute', 'rkot']
print(my_list[1])
print(len(my_list))
my_list[2] = "rajkot"
print(my_list)
print(my_list[-1])
```

Output:

```
institute (List index starts with 0)
3 (length of the List)
['darshan', 'institute', 'rajkot'] (Note :  spelling of rajkot is updated)
rajkot (-1 represent last element)
```

- We can use slicing similar to string to get the sublist from the list.

- Example (Slicing):

```
my_list = ['darshan', 'institute', 'rajkot','gujarat','INDIA']
print(my_list[1:3])
```

Output:

```
['institute', 'rajkot']  (Note: end index is not included)
```

Methods of List

- There are many built-in methods of List in Python, some important are discussed here,
  - list.**append**(element) will append the element at the end of the list.

    Example:

    ```
    my_list = ['darshan', 'institute', 'rajkot']
    my_list.append('gujarat')
    print(my_list)
    ```

    Output:

    ```
    ['darshan', 'institute', 'rajkot', 'gujarat']
    ```

  - list.**insert**() method will add element at the specified index in the list.

    Example:

    ```
    my_list = ['darshan', 'institute', 'rajkot']
    my_list.insert(2,'of')
    my_list.insert(3,'engineering')
    print(my_list)
    ```

    Output:

    ```
    ['darshan', 'institute', 'of', 'engineering', 'rajkot']
    ```

  - list.**extend**() method will add one data structure (List or any) to the current List.

    Example:

    ```
    my_list1 = ['darshan', 'institute']
    my_list2 = ['rajkot','gujarat']
    my_list1.extend(my_list2)
    print(my_list1)
    ```

    Output:

    ```
    ['darshan', 'institute', 'rajkot', 'gujarat']
    ```

  - list.**pop**() method will remove the last element from the list and return it.

Example:

```
my_list = ['darshan', 'institute','rajkot']
temp = my_list.pop()
print(temp)
print(my_list)
```

Output:

```
rajkot
['darshan', 'institute']
```

- list.**remove**() method will remove first element from the list (it doesn't return anything).

  Example:

```
my_list = ['darshan', 'institute', 'darshan','rajkot']
my_list.remove('darshan')
print(my_list)
```

  Output:

```
['institute', 'darshan', 'rajkot']
```

- list.**clear**() method will remove all elements from the list.

  Example:

```
my_list = ['darshan', 'institute', 'darshan','rajkot']
my_list.clear()
print(my_list)
```

  Output:

```
[]  (Empty List)
```

- list.**index**() method will return the first index of the specified element from the list.

  Example:

```
my_list = ['darshan', 'institute', 'darshan','rajkot']
id = my_list.index('institute')
print(id)
```

  Output:

```
1
```

- list.**count**() method will return a number of occurrences for the specified element in the list.

Example:

```
my_list = ['darshan', 'institute', 'darshan','rajkot']
c = my_list.count('darshan')
print(c)
```

Output:

```
2
```

- list.**reverse**() method will reverse the elements of the list.

  Example:

```
my_list = ['darshan', 'institute','rajkot']
my_list.reverse()
print(my_list)
```

  Output:

```
['rajkot', 'institute','darshan']
```

- list.**sort**() method will sort the elements of the list in ascending order, if we want to sort the elements in descending order we can specify the **reverse** parameter to be **True** while calling the sort method.

  Example:

```
my_list = ['darshan', 'college','of','enginnering','rajkot']
my_list.sort()
print(my_list)
my_list.sort(reverse=True)
print(my_list)
```

  Output:

```
['college', 'darshan', 'enginnering', 'of', 'rajkot']
['rajkot', 'of', 'enginnering', 'darshan', 'college']
```

Set

- A **Set** is a **mutable** data structure that holds an **unordered** collection of **unique** items.
- A Set will be represented with the Curly Brackets '**{ Set }**'
- It has similar methods as List, the only difference between List and Set is
  - List is **ordered** collection of items, whereas
  - Set is an **unordered** collection of **unique** items.
- Example:

```
my_set = {1,1,1,2,2,5,3,9}
print(my_set)
```

Output:

```
{1, 2, 3, 5, 9}
```

- Set also has a **union, intersection,** and **different** methods.

Tuple

- The tuple is an **immutable ordered** sequence of zero or more objects; it allows duplicate value similar to the List.
- Tuple will be represented with the Round Brackets '**(Tuple)**'
- Tuple supports the same slicing syntax as String/List.
- Similar to String, Tuple are immutable, so we cannot replace or delete any of their items.
- The only difference between List and Tuple is
  - List is **mutable,** whereas
  - Tuple is **immutable.**
- Example:

```
my_tuple = ('darshan','institute','of','engineering','of')
print(my_tuple)
print(my_tuple.index('engineering'))
print(my_tuple.count('of'))
print(my_tuple[-1])
```

Output:

```
('darshan', 'institute', 'of', 'engineering', 'of')
3
2
of
```

Dictionaries

- A dictionary (dict) is an unordered collection of zero or more key-value pairs whose keys are object references that refer to hashable objects, and whose values are object references referring to objects of any type.
- Dictionaries are mutable, so we can easily add, edit, or remove items, but as it is unordered we cannot slice the dictionary.
- Dictionary will be represented with the Curly Brackets '**{ key: value }**'

```
my_dict = { 'key1':'value1', 'key2':'value2' }
```

Note: key-value are separated by colon (**:**) and pairs of key-value are separated by comma (**,**)

- Example:

```
my_dict = {'college':"darshan", 'city':"rajkot",'type':"enginee
ring"}
print(my_dict['college'])
print(my_dict.get('city'))
```

Output:

```
darshan
rajkot
```

Note: Values can be accessed using key inside square brackets as well as using get() method

Methods of Dictionary

- dict.**keys**() method will return a list of all the keys associated with the Dictionary.

  Example:

```
my_dict = {'college':"darshan",
           'city':"rajkot",
           'type':"engineering"}
print(my_dict.keys())
```

  Output:

```
['college', 'city', 'type']
```

- dict.**values**() method will return list of all the values associated with the Dictionary.

  Example:

```
my_dict = {'college':"darshan",
           'city':"rajkot",
           'type':"engineering"}
print(my_dict.values())
```

  Output:

```
['darshan', 'rajkot', 'engineering']
```

- dict.**items**() method will return a list of tuples for each key-value pair associated with the Dictionary..

  Example:

```
my_dict = {'college':"darshan",
           'city':"rajkot",
           'type':"engineering"}
print(my_dict.items())
```

Output:

```
[('college', 'darshan'), ('city', 'rajkot'), ('type', 'engineering')]
```

List Comprehension

- List comprehensions offer a way to create lists based on existing iterable. When using list comprehensions, lists can be built by using any iterable, including strings, list, tuples.
- For example, if we want to create a list of characters from the string, we can use for loop like below example,
- Example (Using for loop):

```
mystr = 'darshan'
mylist = []
for c in mystr:
    mylist.append(c)

print(mylist)
```

Output:

```
['d', 'a', 'r', 's', 'h', 'a', 'n']
```

- The same output can be achieved using list comprehension with just one line of code.
- Syntax:

```
[ expression for item in iterable ]
                OR
[ expression for item in iterable if condition ]
```

- Example (Using List Comprehension):

```
mylist = [c for c in 'darshan']
print(mylist)
```

Output:

```
['d', 'a', 'r', 's', 'h', 'a', 'n']
```

- Similarly, we can use list comprehensions in many cases where we want to create a list out of other iterable, let's see another example of the use of List Comprehension.
- Example (Using for loop):

```
# list of square from 1 to 10
mylist = []
for i in range(1,11):
    mylist.append(i**2)

print(mylist)
```

Output

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Example (Using List Comprehension):

```
# list of square from 1 to 10
mylist = [ i**2 for i in range(1,11) ]
print(mylist)
```

Output

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- We can also use conditions in List Comprehension, for example, if we want a list of number from 1980 to 2020 which are divisible by 4 then we can use list comprehension.
- Example (Using List Comprehension):

```
leapYears = [i for i in range(1980,2021) if i%4==0]
print(leapYears)
```

Output

```
[1980, 1984, 1988, 1992, 1996, 2000, 2004, 2008, 2012, 2016, 2020]
```

What is Data Science?

- It is a branch of computer science where we study how to store, use, and analyze data for deriving information from it.
- Data Science is about data gathering, analysis, and decision-making.
- Data Science is about finding patterns in data, through analysis, and make future predictions.
- Data science is essentially a method of viewing and analyzing statistics and probability.

Core competencies of a data scientist

- The Data Scientist requires knowledge of a vast range of skills to perform the required tasks.
- Most of the times data scientists work in a team to provide the best results,
  - For example, someone who is good at gathering data might team up with an analyst and some talented in presenting the information.
  - It's a very difficult task to find a single person with all the required skills.
- Below are the areas in which a data scientist could find opportunity
  - Data Capture :
    - It is a process of managing data sources like databases, excel, pdf and text, etc.
    - It converts the unstructured data to structured data.
  - Analysis :
    - It requires knowledge of basic statistical tools.
    - It includes knowledge of specialized math tricks and algorithms for the analysis of data.
  - Presentations :
    - The presentation provides a graphical representation of the pattern.
    - It helps to represent the results of the data analysis to the end-user.

Creating the Data Science Pipeline

- Data science is partly art and partly engineering.
- The Data science pipeline requires the data scientist to follow particular steps in the preparation, analysis, and presentation of the data.
- General steps in the pipeline are
  - Preparing the data :
    - The data we gathered from various sources may not come directly in the structured format.

- We need to transform the data into a structured format.
- Transformation may require changing data types, the order in which data appears, and even the creation of missing data.

- Performing data analysis :
  - Data science provides access to a larger set of statistical methods and algorithms.
  - Sometimes a single approach may not provide the desired output, we need to use multiple algorithms to get the result.
  - The use of trial and error is part of the data science art.
- Learning from data :
  - As we iterate through various statistical analysis methods and apply algorithms to detect patterns, we begin learning from the data.
  - After learning from the data, the result of the algorithm may be different than initially, we predict the output.
- Visualizing :
  - Visualization means seeing the patterns in the data and then being able to react to those patterns.
  - It also means being able to see when data is not part of the pattern.
- Obtaining insights and data products :
  - The insights you obtain from manipulating and analyzing the data help you to perform real-world tasks. For example, you can use the results of an analysis to make a business decision.

Why Python?

- Python is the vision of a single person, Guido van Rossum, Guido started the language in December 1989 as a replacement for the ABC language.
- Grasping Python's core philosophy :
  - Guido started Python as a skunkworks project.
  - The core concept was to create Python as quickly as possible, yet created a flexible language, runs on any platform, and provides significant potential for extension.
  - Python is used to create an application of all types. It support four programming styles(programming paradigms)
  - Functional :
    - Treats every statement as a mathematical equation and avoids any form of state or mutable data.
    - The main advantage of this approach is having no side effects to consider.

- This coding style lends itself better than the others to parallel processing because there is no state to consider.
- Many developers prefer this coding style for recursion and lambda calculus.

- Imperative :
  - Performs computations as a direct change to program state.
  - This style is especially useful when manipulating data structures and produces elegant but simple code.
- Object-oriented :
  - Relies on data fields that are treated as objects and manipulated only through prescribed methods.
  - Python doesn't fully support this coding form because it can't implement features such as data hiding.
  - This is a useful coding style for complex applications because it supports encapsulation and polymorphism.
- Procedural :
  - Treats tasks as step-by-step iterations where common tasks are placed in functions that are called as needed.

Understanding Python's Role in Data Science

- Python has a unique attribute and is easy to use when it comes to quantitative and analytical computing.
- Python is widely used in data science and is a favorite tool along with being a flexible and open-sourced language.
- Its massive libraries are used for data manipulation and are very easy to learn even for a beginner data analyst.
- Apart from being an independent platform it also easily integrates with any existing infrastructure which can be used to solve the most complex problems.
- Python is preferred over other data science tools because of following features,
  - Powerful and Easy to use
  - Open Source
  - Choice of Libraries
  - Flexibility
  - Visualization and Graphics
  - Well supported

Considering Speed of Execution

- Analysis of data requires processing power.
- The dataset is so large that it may slow down a very powerful system.
- Following factors control the speed of execution for data science application
  - Dataset Size :
    - Data science relies on huge datasets in many cases.
    - The application type determines the size of the dataset in part, but dataset size also relies on the size of the source data.
    - Underestimating the effect of dataset size is deadly in data science applications, especially those that need to operate in real-time (such as self-driving cars).
  - Loading Technique :
    - The method we use to load data for analysis is critical, and we should always use the fastest.
    - Working with data in memory is always faster than working with data stored on disk.
    - Accessing local data is always faster than accessing it across a network.
    - Performing data science tasks that rely on network is probably the slowest method of all
  - Coding Style :
    - Anyone can create a slow application using any programming language by employing coding techniques that don't make the best use of programming language functionality.
    - To create fast data science applications, you must use best-of-method coding techniques.
  - Machine capabilities :
    - Running data science applications on a memory-constrained system with a slower processor is impossible.
    - The system you use needs to have the best hardware.
    - Given that data science, applications are both processor and disk-bound, you can't cut corners in any area and expect great results.
  - Analysis Algorithm
    - The algorithm you use determines the kind of result you obtain and controls execution speed.
    - We must experiment to find the best algorithm for a particular dataset.

Performing Rapid Prototyping and Experimentation

- Python is all about creating applications quickly and then experimenting with them to see how things work.
- The act of creating an application design in code without necessarily filling in all the details is called prototyping.
- Python uses less code than other languages to perform tasks, so prototyping goes faster.
- The fact that many of the actions you need to perform are already defined as part of libraries that you load into memory makes things go faster still.
- Data science doesn't rely on static solutions. You may have to try multiple solutions to find the particular solution that works best. This is where experimentation comes into play.
- After you create a prototype, you use it to experiment with various algorithms to determine which algorithm works best in a particular situation.

Using the Python Ecosystem for Data Science

- We need to load certain libraries to perform specific data science tasks in python.
- Following are the list of libraries which we are going to use in this chapter :
  - **NumPy** - Performing fundamental scientific computing.
  - **Pandas** - Performing data analysis using pandas
  - **Matplotlib** - Plotting the data
  - **SciPy** - Accessing scientific tools
  - **Scikit-learn** - Implementing machine learning
  - **Keras** and **TensorFlow -** Used for deep learning
  - **NetworkX -** Creating graphs
  - **Beautiful Soup -** Parsing HTML documents
- **NumPy :**
  - NumPy stands for *Numerical Python*
  - NumPy is a Python library used for working with arrays.
  - It provides a function to work with linear algebra, Fourier transform, and matrices.
  - It is an open-source project and we can use it freely.
  - It is a library consisting of multidimensional array objects and a collection of routines for processing those arrays.
  - Using NumPy, mathematical, and logical operations on arrays can be performed very easily.
  - In Python, we have lists that serve the purpose of arrays, but they are slow to process.

- NumPy provides an array object that is very much faster than traditional Python lists. the array object in NumPy is called ndarray.
- Arrays are very frequently used in data science, where speed and resources are very important.
- Features of NumPy are as follows.
  - powerful n-dimensional arrays
  - numerical computing tools
  - interoperable
  - performant
  - easy to use
  - open source

- **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 offers data structures and operations for manipulating numerical tables and time series.
  - The library is optimized to perform data science tasks especially fast and efficiently.
  - The basic principle behind pandas is to provide data analysis and modeling support for Python that is similar to other languages such as R.
- **Matplotlib :**
  - The matplotlib library gives a MATLAB like an interface for creating data presentations of the analysis.
  - The library is initially limited to 2-D output, but it still provides a means to express analysis graphically.
  - Without this library, we cannot create output that people outside the data science community could easily understand.
- **SciPy :**
  - The SciPy stack contains a host of other libraries that we can also download separately.
  - These libraries provide support for mathematics, science, and engineering.
  - When we obtain SciPy, we get a set of libraries designed to work together to create applications of various sorts, these libraries are
    - NumPy
    - Pandas
    - Matplotlib
    - Sympy

- **Scikit-learn :**
  - The Scikit-learn library is one of many Scikit libraries that build on the capabilities provided by NumPy and SciPy to allow Python developers to perform domain-specific tasks.
  - Scikit-learn library focuses on data mining and data analysis, it provides access to following sort of functionality:
    - Classification
    - Regression
    - Clustering
    - Dimensionality reduction
    - Model selection
    - Pre-processing
- **Keras and TensorFlow :**
  - Keras is an application programming interface (API) that is used to train deep learning models
  - An API often specifies a model for doing something, but it doesn't provide an implementation.
  - TensorFlow is an implementation for the keras, there are many other implementations for the keras like Microsoft's Cognitive Toolkit, CNKT and Theano
- **NetworkX :**
  - NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks (For example GPS setup to discover routes through city streets).
  - NetworkX also provides the means to output the resulting analysis in a form that humans understand.
  - The main advantage of using NetworkX is that nodes can be anything (including images) and edges can hold arbitrary data.
- **Beautiful Soup :**
  - Beautiful Soup is a Python package for parsing HTML and XML documents.
  - It creates a parse tree for parsed pages that can be used to extract data from HTML, which is useful for web scraping.

Introduction to Jupyter/ IPython

- IPython was originally developed by Fernando Perez in 2001 as an enhanced Python interpreter.
- In 2014, Project Jupyter started as a spin-off project from IPython.
- Jupyter provides following packages
  - **Jupyter notebook:** A web-based interface to programming environments of Python.
  - **QtConsole**: Qt based terminal for Jupyter kernels similar to IPython.
  - **Nbviewer**: Facility to share Jupyter notebooks.
  - **JupyterLab**: Modern web based integrated interface for all products.

Working with Data

- Data science applications require data by definition.
- Data requires by the application are stored in a different location and different format.
- We need techniques required to access data in several forms and locations.
  - Memory form – it represents a form of data store that your computer support natively.
  - Flat Files – it represents data are store on a computer hard drive in file format.
  - Relation database – in this large data are stored on the internet and small-sized data may be stored in the small file that appears on your hard drive as well.
- Storing data in local computer memory represents the fastest and most reliable means to access it.
- The data could reside anywhere. However, we don't interact with the data in its storage location. We load the data into memory from the storage location and interact with it in memory.

Basic IO operations in Python

- When it comes to storing, reading, or communicating data, working with the files of an operating system is both necessary and easy with Python.
- Before we can read or write a file, we have to open it using Python's built-in open() function.

File Open

- The open function is used for opening files.

  Syntax :

```
fileobject = open(filename [, accessmode][, buffering])
```

  - **filename** – it specifies the name of the file we want to open.

- **accessmode** – it determines the mode in which the file has to be opened.
- **buffering -** If buffering is set to 0, no buffering will happen, if set to 1 line buffering will happen, if greater than 1, then buffering action is performed with the indicated buffer size and if negative is given it will follow the system default buffering behaviour.
- We can specify a relative path in argument to **open** method, alternatively, we can also specify an absolute path.
- To specify absolute path,
  - In windows, f=open('D:\\folder\\subfolder\\filename.txt')
  - In mac & linux, f=open('/user/folder/subfolder/filename.txt')

- We suppose to close the file once we are done using the file in the Python using **close()** method.

```
f = open('college.txt')
---
f.close()
```

- List of possible file modes are list below

Mode	Description
r	Read only (default)
rb	Read only in binary format
r+	Read and Write both
rb+	Read and Write both in binary format
w	Write only
wb	Write only in binary format
w+	Read and Write both
wb+	Read and Write both in binary format
a	Opens file to append, if file not exist will create it for write
ab	Append in binary format, if file not exist will create it for write
a+	Append, if file not exist it will create for read & write both
ab+	Read and Write both in binary format

Read file in Python

- For reading file content in python following functions are available.
  - **read() :**
    - read() function is used for reading the full contents of a file.
    - **read(size)** will read specified bytes from the file, if we don't specify size it will return the whole file.

      Code:

```
f = open('college.txt')
data = f.read()
print(data)
```

Output:

```
Darshan Institute of Engineering and Technology - Rajkot
At Hadala, Rajkot - Morbi Highway,
Guiarat-363650. INDIA
```

- **readlines() :**
  - It will return a list of lines from the file.

  Code:

```
f = open('college.txt')
lines = f.readlines()
print(lines)
```

  Output:

```
['Darshan Institute of Engineering and Technology - Rajkot\n', 'At Hadala,
Rajkot - Morbi Highway,\n', 'Gujarat-363650, INDIA']
```

- **Iterate files :**
  - We can use for loop to get each line separately,

  Code:

```
f = open('college.txt')
lines = f.readlines()
for l in lines :
    print(l)
```

  Output:

```
Darshan Institute of Engineering and Technology - Rajkot
At Hadala, Rajkot - Morbi Highway,
Gujarat-363650, INDIA
```

"with" statement

- It is used for error handling.
- We may have typo in the filename or file we specified is moved/deleted, in such cases, there will be an error while executing code.
- To handle such situations we can use the new syntax of opening the file using **with** keyword.
- It allows us to ensure that a resource is "cleaned up" when the code that uses it finishes running, even if exceptions are thrown.
- The with statement itself ensures proper acquisition and release of resources
- with statement in your objects will ensure that you never leave any resource open.

```
with open('college.txt') as f :
    data = f.read()
    print(data)
```

- When we open a file using with we need not to close the file.

Write file in Python

- write() method will write the specified data to the file.

```
with open('college.txt','a') as f :
    f.write('Hello world')
```

- If we open a file with '**w**' mode it will overwrite the data to the existing file or will create a new file if the file does not exist.
- If we open a file with '**a**' mode it will append the data at the end of the existing file or will create a new file if the file does not exist.

Reading CSV files without any library functions

- A comma-separated values file is a delimited text file that uses a comma to separate values.
- Each line is a data record, Each record consists of many fields, separated by commas.

Example (Book.csv):

```
studentname,enrollment,cpi
abcd,123456,8.5
bcde,456789,2.5
cdef,321654,7.6
```

- We can use Microsoft Excel to access CSV files.
- In the future, we will access CSV files using different libraries, but we can also access CSV files without any libraries.

```
with open('Book.csv') as f :
  rows = f.readlines()
  isFirstLine = True # to ignore column headers
  for r in rows :
    if isFirstLine :
        isFirstLine = False
        continue
    cols = r.split(',')
    print('Student Name = ', cols[0], end=" ")
    print('\tEn. No. = ', cols[1], end=" ")
    print('\tCPI = \t', cols[2])
```

Output:

```
Student Name =  abcd      En. No. =  123456   CPI =   8.5

Student Name =  bcde      En. No. =  456789   CPI =   2.5

Student Name =  cdef      En. No. =  321654   CPI =   7.6
```

NumPy

- NumPy stands for *Numerical Python*
- NumPy is a Python library used for working with arrays.
- It provides the function to work with linear algebra, Fourier transform, and matrices.
- It is an open-source project and we can use it freely.
- It is a library consisting of multidimensional array objects and a collection of routines for processing those arrays.
- Using NumPy, mathematical, and logical operations on arrays can be performed very easily.
- In Python, we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy provides an array object that is very much faster than traditional Python lists.
- The array object in NumPy is called **ndarray**.
- Arrays are very frequently used in data science, where speed and resources are very important.

  Install :
  - conda install numpy *or*
  - pip install numpy
- Once we installed NumPy, import it in your applications by adding the import keyword

```
import numpy as np
--
```

- NumPy is usually imported under the np alias. Alias is an alternate name for referring to the same thing. now NumPy package can be referred to as np instead of numpy.

NumPy Array

- The most important object defined in NumPy is an N-dimensional array type called **ndarray**.
- It describes the collection of items of the same type, Items in the collection can be accessed using a zero-based index
- An instance of ndarray class can be constructed in many different ways, the basic ndarray can be created as below.

  Syntax :

```
import numpy as np

numpy.array(object [, dtype = None][, copy = True][, order = None][,
subok = False][, ndmin = 0])
```

- The above constructor of array takes the following parameters :

Parameters	Description
object	Any object exposing the array interface method returns an array, or any (nested) sequence like list, tuple, set, and dict.
dtype	The desired data type of array, An optional parameter.
copy	By default (true), the object is copied, an optional parameter.
order	C (row major) or F (column major) or A (any) (default), optional parameter.
subok	By default, the returned array is forced to be a base class array. If true, sub-classes passed through.
ndmin	Specifies minimum dimensions of the resultant array.

Example:

```
import numpy as np
a= np.array(['darshan','Insitute','rajkot'])
print(type(a))
print(a)
```

Output:

```
<class 'numpy.ndarray'>
['darshan' 'Insitute' 'rajkot']
```

NumPy Array creation routines

- The NumPy array can be created from other low-level constructors of ndarray.

numpy.empty

- It creates an uninitialized array of specified shape and dtype

Syntax

```
numpy.empty(shape[, dtype = float][, order = 'C'])
```

- The above constructor takes the following parameters :

Parameters	Description
shape	Shape of an empty array in int or tuple of int
dtype	Desired data type of array, optional parameter
order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Example:

```
import numpy as np
x = np.empty([3,2], dtype = int)
print x
```

Output:

```
[[140587109587816 140587109587816]
 [140587123623488 140587124774352]
 [ 94569341940000  94569341939976]]
```

numpy.zero

- zeros(n) function will return NumPy array of a given shape, filled with zeros.

Syntax

```
numpy.zeros(shape[, dtype = float][, order = 'C'])
```

The above constructor takes the following parameters:

Parameters	Description
shape	Shape of an empty array in int or tuple of int
dtype	Desired data type of array, optional parameter
order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Example:

```
import numpy as np
c = np.zeros(3)
print(c)
c1 = np.zeros((3,3)) #have to give as tuple
print(c1)
```

Output:

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

numpy.ones

- ones(n) function will return NumPy array of a given shape, filled with ones.

Syntax

```
numpy.ones(shape[, dtype = float][, order = 'C'])
```

The above constructor takes the following parameters:

Parameters	Description
shape	Shape of an empty array in int or tuple of int
dtype	Desired data type of array, optional parameter
order	'C' for C-style row-major array, 'F' for FORTRAN style column-major array

Example:

```
import numpy as np
c = np.ones(3)
print(c)
c1 = np.ones((3,3)) #have to give as tuple
print(c1)
```

Output:

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

numpy.arange

- it will create NumPy array starting from start till the end (not included) with specified steps.

Syntax

```
numpy.arange(start, stop[, step][, dtype])
```

The above constructor takes the following parameters:

Parameters	Description
start	The start of an interval. If omitted, defaults to 0
stop	The end of an interval (not including this number)
step	Spacing between values, default is 1
dtype	Desired data type of array, optional parameter

Example:

```
import numpy as np
b = np.arange(0,10,1)
print(b)
```

Output:

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

numpy.linspace

- linspace function will return evenly spaced numbers over a specified interval.

Syntax

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

The above constructor takes the following parameters:

Parameters	Description
start	The starting value of the sequence
stop	The end value of the sequence, included in the sequence if endpoint set to true
num	The number of evenly spaced samples to be generated. Default is 50

endpoint	True by default, hence the stop value is included in the sequence. If false, it is not included
retstep	If true, returns samples and step between the consecutive numbers
dtype	Data type of output ndarray

Example:

```
import numpy as np
c = np.linspace(0,1,11)
print(c)
```

Output:

```
[0. 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.]
```

numpy.logspace

- logspace function will return evenly spaced numbers over a specified interval on a log scale. The scale indicates of base usually 10.

Syntax

```
numpy.logspace(start, stop, num, endpoint, base, dtype)
```

The above constructor takes the following parameters:

Parameters	Description
start	The starting value of the sequence
stop	The end value of the sequence, included in the sequence if endpoint set to true
num	The number of evenly spaced samples to be generated. Default is 50
endpoint	True by default, hence the stop value is included in the sequence. If false, it is not included
base	Base of log space, default is 10
dtype	Data type of output ndarray

Example:

```
import numpy as np
c = np.logspace(1.0,2.0, num = 10)
print(c)
```

Output:

```
[10.  12.91549665  16.68100537   21.5443469    27.82559402
   35.93813664   46.41588834   59.94842503   77.42636827  100.]
```

Attributes of NumPy

- this section describes attributes of NumPy

Attribute	Description	Example	Output

ndarray.shape	This array attribute returns a tuple consisting of array dimensions.	`import numpy as np`
		`a=np.array([[1,3],[4,6]])`
		`print a.shape`
			(2, 2)

ndarray.ndim	This array attribute returns the number of array dimensions.	`import numpy as np`
		`a = np.arange(24)`
		`print(a.ndim)`
			1

numpy.itemsize	This array attribute returns the length of each element of array in bytes.	`import numpy as np`
		`x = np.array([1,2,3,4,5])`
		`print x.itemsize`
			8

numpy.flags	This function returns the current values of a set of flags.	`import numpy as np`
		`x = np.array([1,2,3,4,5])`
		`print x.flags`
			`C_CONTIGUOUS : True`
			`F_CONTIGUOUS : True`
			`OWNDATA : True`
			`WRITEABLE : True`
			`ALIGNED : True`
			`UPDATEIFCOPY : False`

Random Numbers in NumPy

- It is used to generate a random number for the array.
- For generating random numbers NumPy offers the random module to work with random numbers.

```
from numpy import random
--
--
```

numpy.random.rand()

- Rand() function is used for generating random float numbers.

Syntax

```
rand(d0, d1, …, dn)
```

- Here d specifies dimensions.
- Rand function will create an n-dimensional array with random data using a uniform distribution, if we do not specify any parameter it will return a random float number.

Example:

```
import numpy as np
r1 = np.random.rand()
print(r1)
r2 = np.random.rand(3,2) # no tuple
print(r2)
```

Output:

```
0.23937253208490505

[[0.58924723 0.09677878] [0.97945337 0.76537675] [0.73097381 0.51277276]]
```

numpy.random.randint()

- It is used for generating random int numbers.
- randint(low, high, num) function will create a one-dimensional array with num random integer data between low and high.

```
randint(low[, high=None][, size=None][, dtype='l'])
```

- following are the parameters of function

Parameters	Description
low	Lowest integer number for random generation
high	Highest integer number for random generation
size	Total number of random number generation
dtype	Desired dtype of the array

Example:

```
import numpy as np
r3 = np.random.randint(1,100,10)
print(r3)
```

Output:

```
[78 78 17 98 19 26 81 67 23 24]
```

numpy.random.randn()

- Randn() function is used for generating random float numbers.

Syntax

```
randn(d0, d1, …, dn)
```

- Here d specifies dimensions.
- Randn() function will create an n-dimensional array with random data using the normal distribution, if we do not specify any parameter it will return a random float number.

Example:

```
import numpy as np
r1 = np.random.randn()
print(r1)
r2 = np.random.randn(3,2) # no tuple
print(r2)
```

Output:

```
-0.15359861758111037
[[ 0.40967905 -0.21974532]  [-0.90341482 -0.69779498] [ 0.99444948 -1.45308348]]
```

Multidimensional array in NumPy

- In Numpy, the number of dimensions of the array is called the rank of the array.
- A tuple of integers giving the size of the array along each dimension is known as shape of the array
- Elements in Numpy arrays are accessed by using square brackets and can be initialized by using nested Python Lists
- Now let's create 2d array.

Example:

```
arr = np.array([['a','b','c'],['d','e','f'],['g','h','i']])
print('double = ',arr[2][1]) # double bracket notaion
print('single = ',arr[2,1]) # single bracket notation
```

Output:

```
double = h
single = h
```

- There are two ways in which you can access element of multi-dimensional array, example of both the method is given below
- Both methods are valid and provide the same answer, but single bracket notation is recommended as in double bracket notation it will create a temporary sub-array of the third row and then fetch the second column from it.
- Single bracket notation will be easy to read and write while programming

Aggregations in NumPy

- NumPy provides many aggregate functions or statistical functions to work with a single-dimensional or multi-dimensional array.
- Following are the aggregation function of NumPy.

Function	Description
np.mean()	Compute the arithmetic mean along the specified axis.
np.std()	Compute the standard deviation along the specified axis.

np.var()	Compute the variance along the specified axis.
np.sum()	Sum of array elements over a given axis.
np.prod()	Return the product of array elements over a given axis.
np.cumsum()	Return the cumulative sum of the elements along a given axis.
np.cumprod()	Return the cumulative product of elements along a given axis.
np.min()	Return the minimum of an array or minimum along an axis.
np.max()	Return the maximum of an array or minimum along an axis.
np.argmin()	Returns the indices of the minimum values along an axis
np.argmax()	Returns the indices of the maximum values along an axis

- **Example**

```
l = [7,5,3,1,8,2,3,6,11,5,2,9,10,2,5,3,7,8,9,3,1,9,3]
a = np.array(l)
print('Min = ',a.min())
print('ArgMin = ',a.argmin())
print('Max = ',a.max())
print('ArgMax = ',a.argmax())
print('Sum = ',a.sum())
print('Mean = ',a.mean())
print('Std = ',a.std())
```

Output:

```
Min = 1
ArgMin = 3
Max = 11
ArgMax = 8
Sum = 122
Mean = 5.304347826086956
Std = 3.042235771223635
```

- When we apply aggregate functions with multidimensional ndarray, it will apply an aggregate function to all its dimensions (axis).

Example

```
import numpy as np
array2d = np.array([[1,2,3],[4,5,6],[7,8,9]])
print('sum = ',array2d.sum())
```

Output:

```
sum = 45
```

- If we want to get sum of rows or cols we can use the axis argument with the aggregate functions.

Example

```
import numpy as np
array2d = np.array([[1,2,3],[4,5,6],[7,8,9]])
print('sum (cols)= ',array2d.sum(axis=0)) #Vertical
print('sum (rows)= ',array2d.sum(axis=1)) #Horizontal
```

Output:

```
sum (cols) = [12 15 18]
sum (rows) = [6 15 24]
```

Indexing & Slicing in NumPy

- The Contents of ndarray object can be accessed and modified by indexing or slicing, just like Python's in-built container objects.
- Indexing means accessing any element from ndarray.
- Slicing in python means taking elements from one given index to another given index.
- Similar to Python List, we can use the same syntax to slice ndarray.

  Syntax:

  ```
  array[start:end:step]
  ```

  - Start – it represents the starting of slicing. Default value is 0
  - End – it represents the ending of slicing. Default value is length of array
  - Step – it represents the step of the slicing:
- Let's take one example of slicing a single dimensional array. Consider the following example for data.

  Example:

  ```
  import numpy as np
  arr = np.array(['a','b','c','d','e','f','g','h'])
  ```

Expression	Output
arr[::]	['a' 'b' 'c' 'd' 'e' 'f' 'g' 'h']
arr[::2]	['a' 'c' 'e' 'g']
arr[:7:]	['a' 'b' 'c' 'd' 'e' 'f' 'g']
arr[:7:2]	['a' 'c' 'e' 'g']
arr[2::]	['c' 'd' 'e' 'f' 'g' 'h']
arr[2::2]	['c' 'e' 'g']
arr[2:7:]	['c' 'd' 'e' 'f' 'g']
arr[2:7:2]	['c' 'e' 'g']
arr[2:5]	['c' 'd' 'e']
arr[:5]	['a' 'b' 'c' 'd' 'e']
arr[5:]	['f' 'g' 'h']
arr[::-1]	['h' 'g' 'f' 'e' 'd' 'c' 'b' 'a']

- Slicing a multi-dimensional array would be the same as a single dimensional array with the help of the single bracket notation we learn earlier.
- Let's take one example of slicing a multidimensional dimensional array. Consider the following example for data.

Example:

```
import numpy as np
arr = np.array([['a','b','c'],['d','e','f'],['g','h','i']])
```

Expression	Description	Output
arr[0:2 , 0:2]	first two rows and cols	[['a' 'b'] ['d' 'e']]
arr[::-1]	reversed rows	[['g' 'h' 'i'] ['d' 'e' 'f'] ['a' 'b' 'c']]
arr[: , ::-1]	reversed cols	[['c' 'b' 'a'] ['f' 'e' 'd'] ['i' 'h' 'g']]
arr[::-1,::-1]	complete reverse	[['i' 'h' 'g'] ['f' 'e' 'd'] ['c' 'b' 'a']]

- When we slice an array and apply some operation on them, it will also make changes in the original array, as it will not create a copy of an array while slicing.
- Array slicing is mutable.

Example:

```
import numpy as np
arr = np.array([1,2,3,4,5])
arrsliced = arr[0:3]

arrsliced[:] = 2 # Broadcasting

print('Original Array = ', arr)
```

Output:

```
Original Array = [2 2 2 4 5]
Sliced Array = [2 2 2]
```

NumPy Arithmetic Operators

- In NumPy array, we can perform a mathematical operation on every element with a single command.

Example:

```
import numpy as np
arr1 = np.array([[1,2,3],[1,2,3],[1,2,3]])
arr2 = np.array([[4,5,6],[4,5,6],[4,5,6]])
```

Expression	Description	Output
arr1 + 2	addition of matrix with scalar	[[3 4 5] [3 4 5] [3 4 5]]
arr1 + arr2	addition of two matrices	[[5 7 9] [5 7 9] [5 7 9]]
arr1 - 2	subtraction of matrix with scalar	[[-1 0 1] [-1 0 1] [-1 0 1]]
arr1 - arr2	subtraction of two matrices	[[-3 -3 -3] [-3 -3 -3] [-3 -3 -3]]
arr1 / 2	Division of matrix with *scalar*	[[0.5 1. 1.5] [0.5 1. 1.5] [0.5 1. 1.5]]
arr1 / arr2	Division of two matrix	[[0.25 0.4 0.5] [0.25 0.4 0.5] [0.25 0.4 0.5]]
arr1 * 2	multiply matrix with scalar	[[2 4 6] [2 4 6] [2 4 6]]
arr1 * arr2	multiply two matrices	[[4 10 18] [4 10 18] [4 10 18]]
np.matmul(arr1 ,arr2)	Matrix Multiplication	[[24 30 36] [24 30 36] [24 30 36]]
arr1.dot(arr2)	Dot	[[24 30 36] [24 30 36] [24 30 36]]

Sorting array in NumPy

- Sorting means putting elements in an ordered sequence.
- Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.
- The NumPy ndarray object has a function called sort(), which will sort a specified array.

- The sort method returns a copy of the array, leaving the original array unchanged.

Syntax:

```
import numpy as np
--
np.sort(arr,axis,kind,order)
```

- The above sort function takes the following parameters:

Parameters	Description
arr	array to sort (inplace)
axis	axis to sort (default=0)
kind	kind of algo to use, the default value is 'quicksort', we can also specify other algorithms like 'mergesort', 'heapsort'.
order	on which field we want to sort if multiple fields available

Example:

```
import numpy as np
arr = np.array(['Darshan','Rajkot','Insitute','of','Engineering'])
print("Before Sorting = ", arr)
arr.sort() # or np.sort(arr)
print("After Sorting = ",arr)
```

Output:

```
Before Sorting =  ['Darshan' 'Rajkot' 'Insitute' 'of' 'Engineering']
After Sorting =  ['Darshan' 'Engineering' 'Insitute' 'Rajkot' 'of']
```

- Sorting a multidimensional array using the sort function.

Example:

```
import numpy as np
dt = np.dtype([('name', 'S10'),('age', int)])
arr2 = np.array([('Darshan',200),('ABC',300),('XYZ',100)],dtype=dt)
arr2.sort(order='name')
print(arr2)
```

Output:

```
[(b'ABC', 300) (b'Darshan', 200) (b'XYZ', 100)]
```

Conditional Selection in NumPy

- Similar to arithmetic operations when we apply any comparison operator to Numpy Array, then it will be applied to each element in the array and a new bool Numpy Array will be created with values True or False.

Example:

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7,8,9,10])
print(arr)
boolArr = arr > 5
print(boolArr)
newArr = arr[arr > 5]
print(newArr)
```

Output:

```
[ 1  2  3  4  5  6  7  8  9 10]
[False False False False False  True  True  True  True  True]
[ 6  7  8  9 10]
```

- If we pass this bool Numpy Array to subscript operator [] of an original array then it will return a new Numpy Array containing elements from Original array for which there was True in bool Numpy Array.
- Here are some of the examples of conditional selections.

Description	Expression
divisible by 3	arr[arr%3==0]
even numbers	arr[arr%2==0]
Odd numbers	arr[arr%2!=0]
greater than 5 and less than 20	arr[(arr > 5) & (arr < 20)]

Pandas

- In 2008, developer Wes McKinney started developing pandas when in need of high performance, flexible tool for analysis of data.
- Before Pandas, Python was majorly used for data munging and preparation. It had very little contribution to data analysis. Pandas solved this problem.
- Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data — load, prepare, manipulate, model, and analyze.
- Pandas is an open-source library built on top of NumPy.
- It allows for fast data cleaning, preparation, and analysis.
- It excels in performance and productivity.
- It also has built-in visualization features.
- It can work with data from a wide variety of sources.
- It has the following features.
    - Time Series functionality.
    - Fast and efficient DataFrame object with the default and customized indexing.
    - Tools for loading data into in-memory data objects from different file formats.
    - Data alignment and integrated handling of missing data.
    - Reshaping and pivoting of data sets.
    - Label-based slicing, indexing, and subsetting of large data sets.

- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High-performance merging and joining of data.

- **Install :**
  - conda install pandas *or*
  - pip install pandas
- Once we installed Pandas, import it in your applications by adding the import keyword

```
import pandas as pd
--
```

- Pandas is usually imported under the pd alias. Alias is an alternate name for referring to the same thing. now Pandas package can be referred to as pd instead of Pandas.

Series in Pandas

- Series is a one-dimensional* array with axis labels.
- It supports both integer and label-based index but the index must be of hashable type.
- If we do not specify an index it will assign an integer zero-based index.
- Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

Syntax:

```
import pandas as pd
s = pd.Series(data,index,dtype,copy=False)
```

- The above constructor takes the following parameters :

Parameters	Description
data	array like Iterable, dict, or scalar value
index	array-like or Index (1d)
dtype	The data type for the output Series. If not specified, this will be inferred from data. Optional argument
copy	Copy input data. Default is False

- Let's takes one example of pandas.

Example:

```
import pandas as pd
s = pd.Series([1, 3, 5, 7, 9, 11])
print(s)
```

Output:

```
0     1
1     3
2     5
3     7
4     9
5    11
dtype: int64
```

- We can then access the elements inside Series just like the array using square brackets notation.
- We can specify the data type of Series using dtype parameter

Example:

```
import pandas as pd
s = pd.Series([1, 3, 5, 7, 9, 11], dtype='str')
print("S[0] = ", s[0])
b = s[0] + s[1]
print("Sum = ", b)
```

Output:

```
S[0] = 1
Sum = 13
```

- We can specify an index to Series with the help of the index parameter.

Example:

```
import numpy as np
import pandas as pd
i = ['name','address','phone','email','website']
d = ['darshan','rj',’123','d@d.com','darshan.ac.in']
s = pd.Series(data=d,index=i)
print(s)
```

Output:

```
name              darshan
address                rj
phone                 123
email             d@d.com
website    darshan.ac.in
dtype:object
```

Creating Time Series

- We can use some of the pandas inbuilt date functions to create a time series.

Example:

```
import numpy as np
import pandas as pd
dates = pd.to_datetime("27th of July, 2020")
i = dates + pd.to_timedelta(np.arange(5), unit='D')
d = [50,53,25,70,60]
time_series = pd.Series(data=d,index=i)
print(time_series)
```

Output:

```
2020-07-27    50
2020-07-28    53
2020-07-29    25
2020-07-30    70
2020-07-31    60
dtype: int64
```

Data Frames in Pandas

- Data frames are two-dimensional data structures, i.e. data is aligned in a tabular format in rows and columns.
- The Data frame also contains labeled axes on rows and columns.
- Features of Data Frame :
  - It is size-mutable.
  - Has labeled axes.
  - Columns can be of different data types.
  - We can perform arithmetic operations on rows and columns.

Structure :

	PDS	Algo	SE	INS
101				
102				
103				
....				
160				

- A pandas DataFrame can be created using the following constructor

Syntax:

```
import pandas as pd
df = pd.DataFrame(data,index,columns,dtype,copy=False)
```

- The above constructor takes the following parameters :

Parameters	Description
data	ndarray (structured or homogeneous), Iterable, dict, or DataFrame
index	array like row index
columns	array like col index
dtype	The data type for the output dataframe. If not specified, this will be inferred from data. Optional argument
copy	Copy data from inputs. Default False

- Let's take one example of pandas.

Example:

```
import numpy as np
import pandas as pd
randArr = np.random.randint(0,100,20).reshape(5,4)
df =
pd.DataFrame(randArr,np.arange(101,106,1),['PDS','Algo','SE','INS'])
print(df)
```

Output:

```
     PDS  Algo  SE  INS
101    0    23  93   46
102   85    47  31   12
103   35    34   6   89
104   66    83  70   50
105   65    88  87   87
```

- We can access columns, rows, and perform an operation on the data frame. Consider the below examples.

Action	Example	Output
Grabbing the column	`df['PDS']`	101 0 102 85 103 35 104 66 105 65 Name: PDS, dtype: int32
Grabbing the multiple column	`df['PDS', 'SE']`	PDS SE 101 0 93 102 85 31 103 35 6 104 66 70 105 65 87
Grabbing a row – for accessing row loc() function is used	`df.loc[101] Or` `df.iloc[0]`	PDS 0 Algo 23 SE 93 INS 46 Name: 101, dtype: int32

Grabbing Single Value	`df.loc[101, 'PDS']`	0
Deleting Row	`df.drop('103',inplace=True)`	PDS Algo SE INS 101 0 23 93 46 102 85 47 31 12 104 66 83 70 50 105 65 88 87 87
Creating new column and column Sum	`df['total'] = df['PDS'] + df['Algo'] + df['SE'] + df['INS']` `print(df)`	PDS Algo SE INS total 101 0 23 93 46 162 102 85 47 31 12 175 103 35 34 6 89 164 104 66 83 70 50 269 105 65 88 87 87 327
Deleting Column and Row	`df.drop('total',axis=1,inplace=True)`	PDS Algo SE INS 101 0 23 93 46 102 85 47 31 12 103 35 34 6 89 104 66 83 70 50 105 65 88 87 87
Getting Subset of Data Frame	`df.loc[[101,104], [['PDS','INS']]`	PDS INS 101 0 46 104 66 50
Selecting all cols except one	`df.loc[:, df.columns != 'Algo' ]`	PDS SE INS 101 0 93 46 102 85 31 12 103 35 6 89 104 66 70 50 105 65 87 87

Conditional Selection in Pandas

- Similar to NumPy we can do conditional selection in pandas.
- We can then use this boolean DataFrame to get associated values.
- It will set NaN (Not a Number) in case of False
- We can apply conditions to a specific column.

Example:

```
import numpy as np
import pandas as pd
np.random.seed(121)
randArr = np.random.randint(0,100,20).reshape(5,4)
df =
pd.DataFrame(randArr,np.arange(101,106,1),['PDS','Algo','SE','INS'])
print(df)
dfBool = df > 50
print(dfBool)
print(df[dfBool])
dfBool1 = df['PDS'] > 50
print(df[dfBool1])
```

Output:

```
       PDS   Algo     SE   INS
101     66     85      8    95
102     65     52     83    96
103     46     34     52    60
104     54      3     94    52
105     57     75     88    39

       PDS   Algo     SE    INS
101   True   True  False   True
102   True   True   True   True
103  False  False   True   True
104   True  False   True   True
105   True   True   True  False

       PDS   Algo     SE   INS
101     66     85    NaN    95
102     65     52     83    96
103    NaN    NaN     52    60
104     54    NaN     94    52
105     57     75     88   NaN

       PDS   Algo     SE   INS
101     66     85      8    95
102     65     52     83    96
104     54      3     94    52
105     57     75     88    39
```

Read CSV in Pandas

- read_csv() is used to read the Comma Separated Values (CSV) file into a pandas DataFrame.

Syntax:

```
pd.read_csv(filepath, sep, header, index_col)
```

- Some of important Parameters of above method are as follow :

Parameters	Description
filePath	Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected
sep	Delimiter to use. (Default is comma)
header	Row number(s) to use as the column names.
index_col	index column(s) of the data frame.

Example:

```
dfINS = pd.read_csv('Marks.csv',index_col=0,header=0)
print(dfINS)
```

Output:

```
      PDS  Algo  SE   INS
101   50    55  60  55.0
102   70    80  61  66.0
103   55    89  70  77.0
104   58    96  85  88.0
201   77    96  63  66.0
```

Read Excel in Pandas

- Read an Excel file into a pandas DataFrame.
- Supports xls, xlsx, xlsm, xlsb, odf, ods and odt file extensions read from a local filesystem or URL. Supports an option to read a single sheet or a list of sheets.
- real_excel () function is used for reading excel file.

Syntax:

```
pd. read_excel(excelFile, sheet_name, header, index_col)
```

- Some of important Parameters of above method are as follow :

Parameters	Description
excelFile	str, bytes, ExcelFile, xlrd.Book, path object, or file-like object
sheet_name	Sheet no in integer or the name of the sheet, can have a list of sheets.
index_col	Index column of the data frame.

Example:

```
df = pd.read_excel('records.xlsx', sheet_name='Employees')
print(df)
```

Output:

```
   EmpID    EmpName EmpRole
0      1     abc      CEO
1      2     xyz    Editor
2      3     pqr    Author
```

Read from MySQL Database

- We need two libraries for that
  - conda install sqlalchemy
  - conda install pymysql
- After installing both the libraries, import create_engine from sqlalchemy and import pymysql

```
from sqlalchemy import create_engine
import pymysql
```

- Then, create a database connection string and create engine using it.

```
db_connection_str = 'mysql+pymysql://username:password@host/dbname'
db_connection = create_engine(db_connection_str)
```

- After getting the engine, we can fire any sql query using pd.read_sql method.
- read_sql is a generic method that can be used to read from any sql (MySQL,MSSQL, Oracle etc…)

```
df = pd.read_sql('SELECT * FROM cities', con=db_connection)
print(df)
```

Output:

```
   CityID   CityName      City  Description    CityCode
0       1      Rajkot     Rajkot Description here       RJT
1       2  Ahemdabad  Ahemdabad Description here       ADI
2       3      Surat      Surat Description here       SRT
```

Setting/Resetting index in Pandas

- We have seen index does not have a name, if we want to specify a name to an index we can specify it using DataFrame.index.name property.

Syntax:

```
df.index.name('Index name')
```

- We can use pandas built-in methods to set or reset the index
- pd.set_index('NewColumn',inplace=True), will set new column as index,
- pd.reset_index(), will reset index to zero based numberic index.

Setting/Resetting index in Pandas

- Hierarchical indexes (AKA multi indexes) help us to organize, find, and aggregate information faster at almost no cost.
- Example where we need Hierarchical indexes

Numeric Index/Single Index:

```
      Col Dep  Sem   RN  S1  S2  S3
0       ABC  CE    5  101  50  60  70
1       ABC  CE    5  102  48  70  25
2       ABC  CE    7  101  58  59  51
3       ABC  ME    5  101  30  35  39
4       ABC  ME    5  102  50  90  48
5  Darshan  CE    5  101  88  99  77
6  Darshan  CE    5  102  99  84  76
7  Darshan  CE    7  101  88  77  99
8  Darshan  ME    5  101  44  88  99
```

Multi Index Index:

```
                RN  S1  S2  S3
Col     Dep Sem
ABC     CE  5    101  50  60  70
            5    102  48  70  25
            7    101  58  59  51
        ME  5    101  30  35  39
            5    102  50  90  48
Darshan CE  5    101  88  99  77
            5    102  99  84  76
            7    101  88  77  99
        ME  5    101  44  88  99
```

- Creating multi indexes is as simple as creating a single index using set_index method, only difference is in case of multi indexes we need to provide list of indexes instead of a single string index, let's see an example for that

Example:

```
dfMulti = pd.read_csv('MultiIndexDemo.csv')
dfMulti.set_index(['Col','Dep','Sem'],inplace=True)
print(dfMulti)
```

Output:

```
                RN  S1  S2  S3
Col     Dep Sem
ABC     CE  5    101  50  60  70
            5    102  48  70  25
            7    101  58  59  51
        ME  5    101  30  35  39
            5    102  50  90  48
Darshan CE  5    101  88  99  77
            5    102  99  84  76
            7    101  88  77  99
        ME  5    101  44  88  99
```

- Now we have a multi-indexed DataFrame from which we can access data using multiple indexes.

Example:

```
print(dfMulti.loc['Darshan']) # Sub DataFrame for all the students
of Darshan
print(dfMulti.loc['Darshan','CE']) # Sub DataFrame for Computer
Engineering students from Darshan
```

Output:

```
          RN  S1  S2  S3
Dep Sem
CE  5    101  88  99  77
    5    102  99  84  76
    7    101  88  77  99
ME  5    101  44  88  99

      RN  S1  S2  S3
Sem
5    101  88  99  77
5    102  99  84  76
7    101  88  77  99
```

Reading in Multiindexed DataFrame directly from CSV

- read_csv function of pandas provides an easy way to create multi-indexed DataFrame directly while fetching the CSV file.
- Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int/str is given, a MultiIndex is used.

Syntax:

```
pd.read_csv('MultiIndexDemo.csv',index_col=[{Comma separated column
index}])
```

Example:

```
dfMultiCSV = pd.read_csv('MultiIndexDemo.csv',index_col=[0,1,2])
print(dfMultiCSV)
```

Output:

```
                  RN  S1  S2  S3
Col     Dep Sem
ABC     CE  5    101  50  60  70
            5    102  48  70  25
            7    101  58  59  51
        ME  5    101  30  35  39
            5    102  50  90  48
Darshan CE  5    101  88  99  77
            5    102  99  84  76
            7    101  88  77  99
        ME  5    101  44  88  99
```

Cross Section in DataFrame

- The xs() function is used to get cross-section from the Series/DataFrame.

- This method takes a key argument to select data at a particular level of a MultiIndex.

Syntax:

```
DataFrame.xs(key, axis=0, level=None, drop_level=True)
```

- Some of important Parameters of above method are as follow :

Parameters	Description
key	label
axis	Axis to retrieve cross-section
level	level of key
drop_level	False if you want to preserve the level

Example:

```
dfMultiCSV = pd.read_csv('MultiIndexDemo.csv', index_col=[0,1,2])
print(dfMultiCSV)
print(dfMultiCSV.xs('CE',axis=0,level='Dep'))
```

Output:

```
                  RN  S1  S2  S3
Col     Dep Sem
ABC     CE  5    101  50  60  70
            5    102  48  70  25
            7    101  58  59  51
        ME  5    101  30  35  39
            5    102  50  90  48
Darshan CE  5    101  88  99  77
            5    102  99  84  76
            7    101  88  77  99
        ME  5    101  44  88  99

RN  S1  S2  S3
Col     Sem
ABC     5    101  50  60  70
        5    102  48  70  25
        7    101  58  59  51
Darshan 5    101  88  99  77
        5    102  99  84  76
        7    101  88  77  99
```

Dealing with Missing Data

- There are many methods by which we can deal with the missing data, some of most commons are listed below,

dropna

- it will drop(delete) the missing data(rows/cols)

Syntax:

```
DataFrame.dropna(axis, how, inplace)
```

- The above method takes the following parameters :

Parameters	Description
axis	Determine if rows or columns which contain missing values are removed.
how	Determine if a row or column is removed from DataFrame when we have at least one NA or all NA. • 'any': If any NA values are present, drop that row or column. • 'all': If all values are NA, drop that row or column.
inplace	If True, do the operation in place and return None.

fillna

- It will fill specified values in place of missing data.

  Syntax:

```
DataFrame.interpolate(self, method='linear', axis=0, limit=None,
inplace=False, limit_direction='forward', limit_area=None,
downcast=None)
```

- The above method takes the following parameters :

Parameters	Description
value	Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). Values not in the dict/Series/DataFrame will not be filled. This value cannot be a list.
method	Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use next valid observation to fill gap.
axis	Axis along which to fill missing values.
inplace	If True, fill in-place. Note: this will modify any other views on this object (e.g., a no-copy slice for a column in a DataFrame).
limit	If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill.
downcast	A dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type

Interpolate

- it will interpolate missing data and fill interpolated values in place of missing data.

  Syntax:

```
DataFrame.interpolate(method='linear', axis=0, limit=None,
inplace=False, limit_direction='forward', limit_area=None,
downcast=None)
```

- The above method takes the following parameters :

Parameters	Description
method	Interpolation technique to use. One of:

	Linear, time, index, pad, nearest, zero, slinear, quadratic, cubic, spine, barycentric, krogh, from_derivatives.
axis	Axis to interpolate along.
inplace	Update the data in place if possible.
limit	The maximum number of consecutive NaNs to fill. Must be greater than 0.
limit_direction	If the limit is specified, consecutive NaNs will be filled in this direction.
limit_area	If the limit is specified, consecutive NaNs will be filled with this restriction.
downcast	Downcast dtypes if possible.

Groupby in Pandas

- Any groupby operation involves one of the following operations on the original object. They are
  - Splitting the Object
  - Applying a function
  - Combining the results
- In many situations, we split the data into sets and we apply some functionality on each subset.
- we can perform the following operations
  - Aggregation – computing a summary statistic
  - Transformation – perform some group-specific operation
  - Filtration – discarding the data with some condition
- Basic ways to use of groupby method
  - df.groupby('key')
  - df.groupby(['key1','key2'])
  - df.groupby(key,axis=1)
- Consider IPL database Example

Example:

```
import pandas as pd
ipl_data = {'Team': ['Riders', 'Riders', 'Devils', 'Devils',
'Kings', 'kings', 'Kings', 'Kings', 'Riders', 'Royals', 'Royals',
'Riders'],'Rank': [1, 2, 2, 3, 3,4 ,1 ,1,2 , 4,1,2],'Year':
[2014,2015,2014,2015,2014,2015,2016,2017,2016,2014,2015,2017],
'Points':[876,789,863,673,741,812,756,788,694,701,804,690]}
df = pd.DataFrame(ipl_data)
print df.groupby('Team').groups
groupIPL = df.groupby('Year')
for name,group in groupIPL :
  print(name)
  print(group)
```

Output:

```
{'Kings': Int64Index([4, 6, 7], dtype='int64'),
 'Devils': Int64Index([2, 3], dtype='int64'),
 'Riders': Int64Index([0, 1, 8, 11], dtype='int64'),
 'Royals': Int64Index([9, 10], dtype='int64'),
 'kings': Int64Index([5], dtype='int64')}
2014
   Points  Rank    Team  Year
0     876     1  Riders  2014
2     863     2  Devils  2014
4     741     3   Kings  2014
9     701     4  Royals  2014
2015
    Points  Rank    Team  Year
1      789     2  Riders  2015
3      673     3  Devils  2015
5      812     4   kings  2015
10     804     1  Royals  2015
2016
   Points  Rank    Team  Year
6     756     1   Kings  2016
8     694     2  Riders  2016
2017
    Points  Rank    Team  Year
7      788     1   Kings  2017
11     690     2  Riders  2017
```

Concatenation in Pandas

- Concatenation basically glues together DataFrames.
- Keep in mind that dimensions should match along the axis you are concatenating on.
- You can use pd.concat and pass in a list of DataFrames to concatenate together:
- We can use the axis=1 parameter to concatenate columns.

Example:

```
dfCX = pd.read_csv('CX_Marks.csv',index_col=0)
dfCY = pd.read_csv('CY_Marks.csv',index_col=0)
dfCZ = pd.read_csv('CZ_Marks.csv',index_col=0)
dfAllStudent = pd.concat([dfCX,dfCY,dfCZ])
print(dfAllStudent)
```

Output:

```
      PDS   Algo SE
101    50    55  60
102    70    80  61
103    55    89  70
104    58    96  85
201    77    96  63
202    44    78  32
203    55    85  21
204    69    66  54
301    11    75  88
302    22    48  77
303    33    59  68
304    44    55  62
```

Join in Pandas

- join() method will efficiently join multiple DataFrame objects by index(or column specified).

Syntax:

```
df.join(dfOther, on, header, how)
```

- Some of important Parameters of above method are as follow :

Parameters	Description
dfOther	Right Data Frame
on	specify the column on which we want to join (Default is index)
how	• left - use calling frame's index (Default). • right - use dfOther index. • outer - form union of calling frame's index with other's index (or column if on is specified), and sort it. lexicographically. • inner - form intersection of calling frame's index (or column if on is specified) with other's index, preserving the order of the calling's one.

Example:

```
dfINS = pd.read_csv('INS_Marks.csv',index_col=0)
dfLeftJoin = allStudent.join(dfINS)
print(dfLeftJoin)
```

Output:

```
      PDS  Algo  SE   INS
101   50    55  60  55.0
102   70    80  61  66.0
103   55    89  70  77.0
104   58    96  85  88.0
201   77    96  63  66.0
202   44    78  32   NaN
203   55    85  21  78.0
204   69    66  54  85.0
301   11    75  88  11.0
302   22    48  77  22.0
303   33    59  68  33.0
304   44    55  62  44.0
```

Merge in in Pandas

- Merge DataFrame or named Series objects with a database-style join.
- Similar to the join method, but used when we want to join/merge with the columns instead of index.

Syntax:

```
object.merge(dfOther, on, left_on, right_on, how)
```

- Some of important Parameters of above method are as follow :

Parameters	Description
dfOther	Right Data Frame
on	specify the column on which we want to join (Default is index)
left_on	specify the column of left Dataframe
right_on	specify the column of right Dataframe
how	• left - use calling frame's index (Default). • right - use dfOther index. • outer - form union of calling frame's index with other's index (or column if on is specified), and sort it. lexicographically. • inner - form intersection of calling frame's index (or column if on is specified) with other's index, preserving the order of the calling's one.

Example:

```
m1 = pd.read_csv('Merge1.csv')
print(m1)
m2 = pd.read_csv('Merge2.csv')
print(m2)
m3 = m1.merge(m2,on='EnNo')
print(m3)
```

Output:

```
   RollNo      EnNo Name
0     101  11112222  Abc
1     102  11113333  Xyz
2     103  22224444  Def

       EnNo  PDS  INS
0  11112222   50   60
1  11113333   60   70

   RollNo      EnNo Name  PDS  INS
0     101  11112222  Abc   50   60
1     102  11113333  Xyz   60   70
```

NumPy v/s Pandas

NumPy	Pandas
NumPy module works with numerical data.	Pandas module works with the tabular data.
NumPy has a powerful tool like Arrays.	Pandas has powerful tools like Series, DataFrame etc.
NumPy consumes less memory as compared to Pandas.	Pandas consume large memory as compared to NumPy.
NumPy provides a multi-dimensional array.	Pandas provide 2d table object called DataFrame.
Indexing of numpy Arrays is very fast.	Indexing of the pandas series is very slow as compared to numpy arrays.
if we want ease of coding we should use pandas.	if we want performance we should use NumPy,

Web Scrapping using Beautiful Soup

- Beautiful Soup is a library that makes it easy to scrape information from web pages.
- It sits atop an HTML or XML parser, providing Pythonic idioms for iterating, searching, and modifying the parse tree.

Example:

```
import requests
import bs4
req = requests.get('https://www.darshan.ac.in/DIET/CE/Faculty')
soup = bs4.BeautifulSoup(req.text,'lxml')
allFaculty = soup.select('body > main > section:nth-child(5) > div >
div > div.col-lg-8.col-xl-9 > div > div')
for fac in allFaculty :
    allSpans = fac.select('h2>a')
    print(allSpans[0].text.strip())
```

Output:

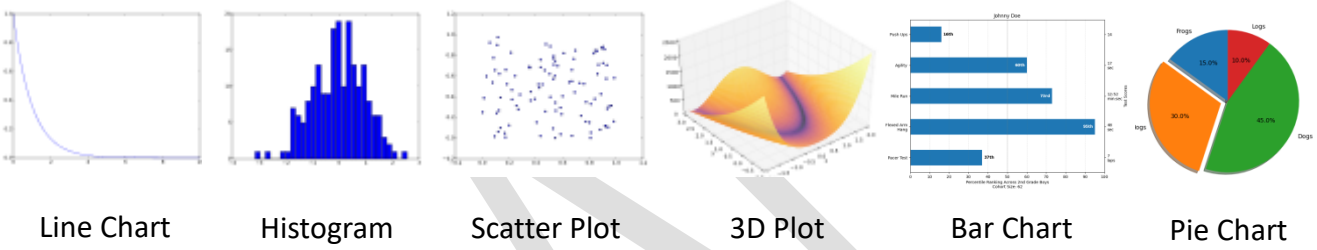
```
Dr. Gopi Sanghani
Dr. Nilesh Gambhava
Dr. Pradyumansinh Jadeja
---
```

Bag of Words model

- Before we can perform analysis on textual data, we must tokenize every word within the dataset.
- The bag of Words model is used to preprocess the text by converting it into a bag of words, which keeps a count of the total occurrences of most frequently used words.
- This model can be visualized using a table, which contains the count of words corresponding to the word itself.
- A bag of words is a representation of text that describes the occurrence of words within a document.
- We just keep track of word counts and disregard the grammatical details and the word order.
- It is called a "bag" of words because any information about the order or structure of words in the document is discarded.
- The model is only concerned with whether known words occur in the document, not wherein the document.
- With the bag-of-Words technique, we can convert variable-length texts into a fixed-length vector.

Introduction to Matplotlib.

- Most people visualize information better when they see it in graphic versus textual format.
- Graphics help people see relationships and make comparisons with greater ease.
- Fortunately, Python makes the task of converting textual data into graphics relatively easy using libraries, one of the most commonly used library for this is Matplotlib.
- Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.
- Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, bar charts, scatterplots, etc., with just a few lines of code.
- A Graph or chart is simply a visual representation of numeric data, Matplotlib makes a large number of graph and chart types.
- Some of graphs/charts are shown below,

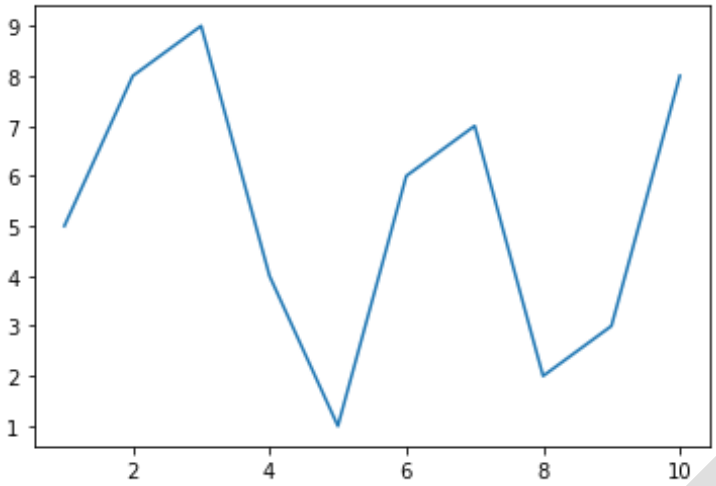


Line chart

- A line chart (also known as a line plot or line graph) is a graph that uses lines to connect individual data points that display quantitative values over specified x-axis values.
- Line graphs use data point "**markers**" that are connected by straight lines to aid in visualization.
- In finance, line graphs are the most frequently used visual representation of values over time.
- To create a line chart in Matplotlib we need some values and the **matplotlib.pyplot** module.
- Example:

```
import matplotlib.pyplot as plt
%matplotlib inline
values = [5,8,9,4,1,6,7,2,3,8]
plt.plot(range(1,11),values)
plt.show()
```

Output:

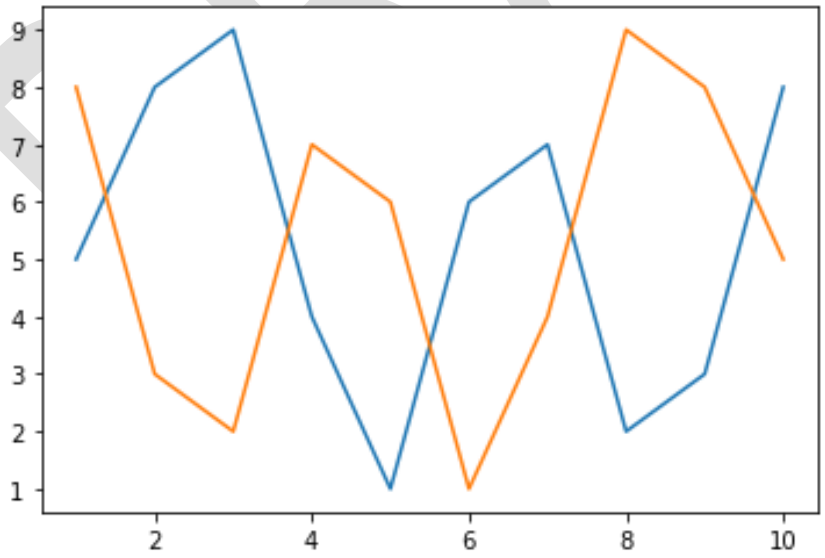


Drawing multiple lines

- We can draw multiple lines in a plot by making multiple plt.plot() calls.
- Example:

```
import matplotlib.pyplot as plt
%matplotlib inline
values1 = [5,8,9,4,1,6,7,2,3,8]
values2 = [8,3,2,7,6,1,4,9,8,5]
plt.plot(range(1,11),values1)
plt.plot(range(1,11),values2)
plt.show()
```

Output:



Saving/Exporting Graphs

- We can export/save our plots on a drive using savefig() method.

- Example:

```
import matplotlib.pyplot as plt
%matplotlib inline
values1 = [5,8,9,4,1,6,7,2,3,8]
values2 = [8,3,2,7,6,1,4,9,8,5]
plt.plot(range(1,11),values1)
plt.plot(range(1,11),values2)
#plt.show()
plt.savefig('SaveToPath.png',format='png')
```

Output: stored image file named *SaveToPath.png*

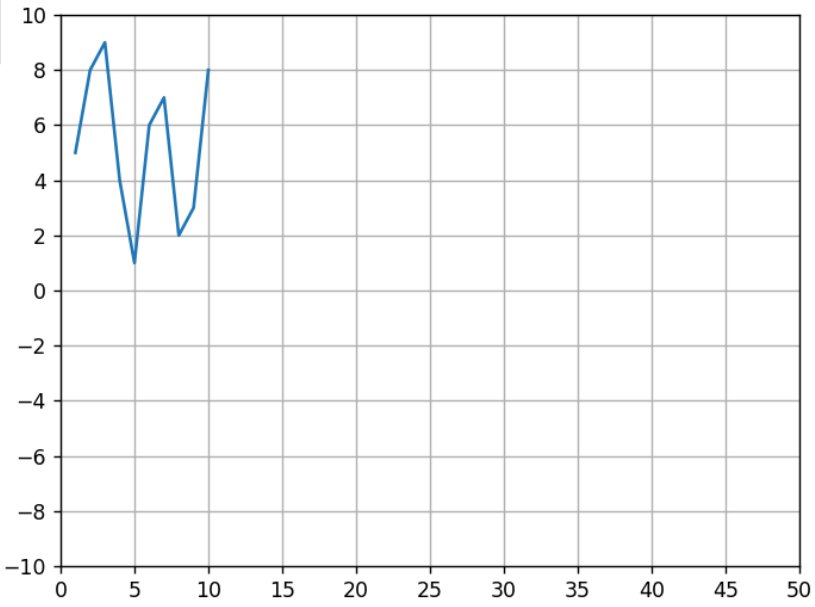
Note: some of the possible values for the *format* is png, svg, pdf etc…

Axis, Ticks and Grid

- We can access and format the axis, ticks, and grid on the plot using the axis() method of the matplotlib.pyplot.plt
- Example:

```
import matplotlib.pyplot as plt
%matplotlib notebook
values = [5,8,9,4,1,6,7,2,3,8]
ax = plt.axes()
ax.set_xlim([0,50])
ax.set_ylim([-10,10])
ax.set_xticks([0,5,10,15,20,25,30,35,40,45,50])
ax.set_yticks([-10,-8,-6,-4,-2,0,2,4,6,8,10])
ax.grid()
plt.plot(range(1,11),values)
```

Output:

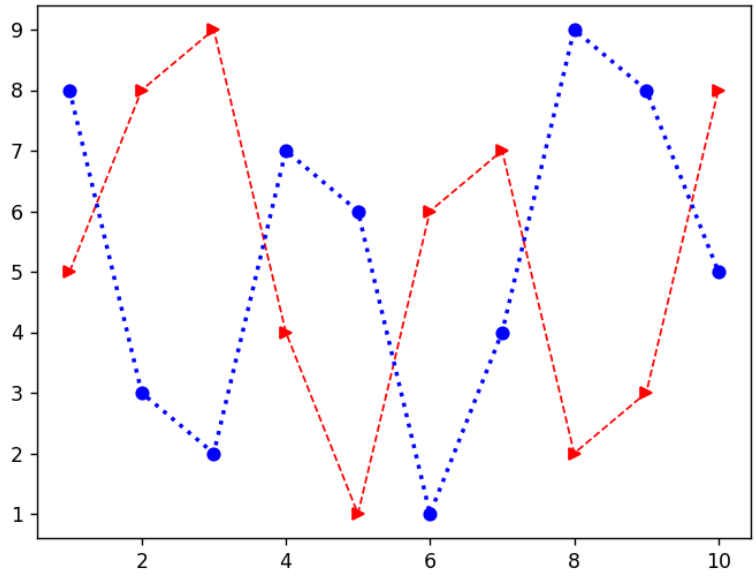


Line Appearance

- We need different line styles to differentiate when having multiple lines in the same plot, we can achieve this using many parameters, some of them are listed below.
  - Line style (*linestyle* or *ls*)
  - Line width (*linewidth* or *lw*)
  - Line color (*color* or *c*)
  - Markers (*marker*)
- Example:

```
import matplotlib.pyplot as plt
%matplotlib inline
values1 = [5,8,9,4,1,6,7,2,3,8]
values2 = [8,3,2,7,6,1,4,9,8,5]
plt.plot(range(1,11),values1,c='r',lw=1,ls='--',marker='>')
plt.plot(range(1,11),values2,c='b',lw=2,ls=':',marker='o')
plt.show()
```

Output:



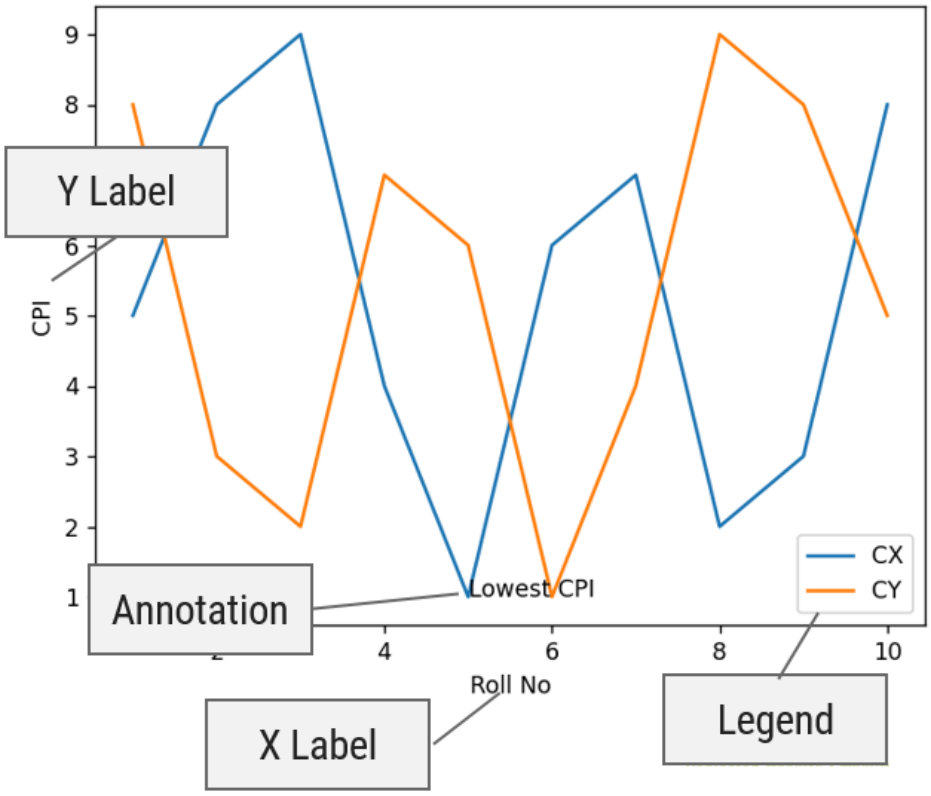
- Possible Values for each parameter are,

Values	Line Style		Values	Colors		Values	Markers
'-'	Solid line		'b'	Blue		'.'	Point
'--'	Dashed line		'g'	Green		','	Pixel
'-.'	Dash-dot line		'r'	Red		'o'	Circle
':'	Dotted line		'c'	Cyan		'v'	Triangle down
			'm'	Magenta		'^'	Triangle up
			'y'	Yellow		'>'	Triangle right
			'k'	Black		'<'	Triangle left
			'w'	White		'*'	Star

			'+'	Plus
			'x'	X
			Etc....	

Labels, Annotation and Legends

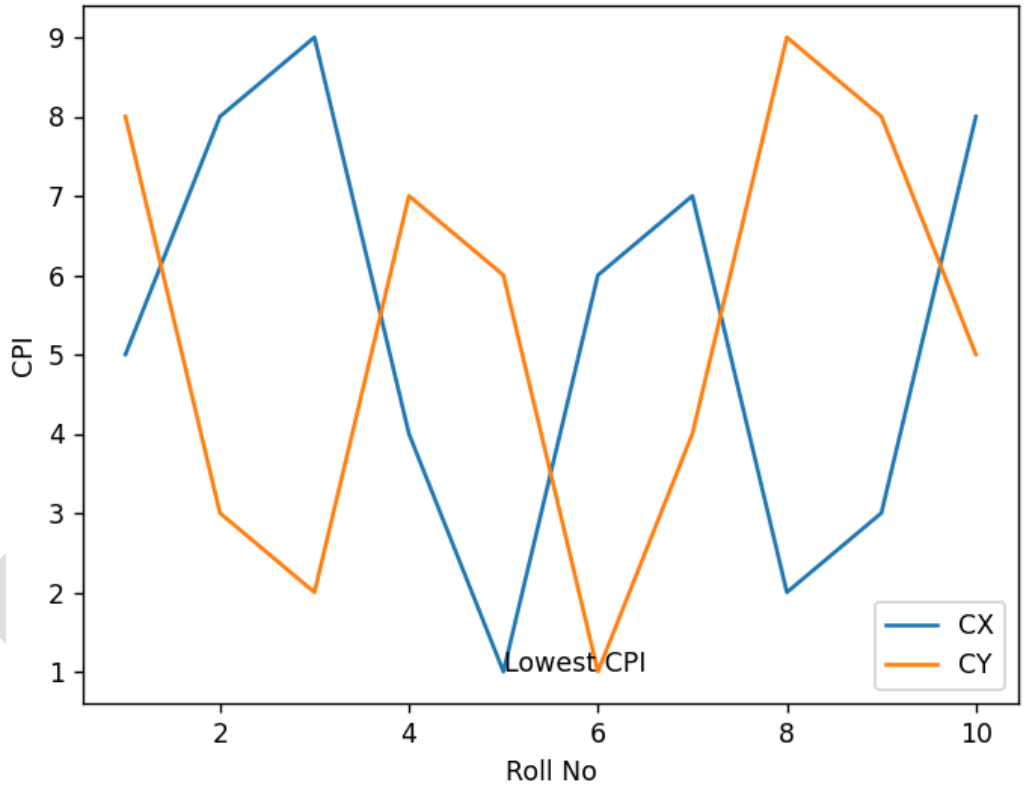
- To fully document our graph, we have to resort the labels, annotation, and legends.
- Each of this element have a different purpose as follows,
  - **Label:** provides identification of a particular data element or grouping, it will make it easy for the viewer to know the name or kind of data illustrated.
  - **Annotation:** augments the information the viewer can immediately see about the data with notes, sources, or other useful information.
  - **Legend:** presents a listing of the data groups within the graph and often provides cues (such as line type or color) to identify the line with the data.



- Example:

```
import matplotlib.pyplot as plt
%matplotlib inline
values1 = [5,8,9,4,1,6,7,2,3,8]
values2 = [8,3,2,7,6,1,4,9,8,5]
plt.plot(range(1,11),values1)
plt.plot(range(1,11),values2)
plt.xlabel('Roll No')
plt.ylabel('CPI')
plt.annotate(xy=[5,1],s='Lowest CPI')
plt.legend(['CX','CY'],loc=4)
plt.show()
```

Output:



Choosing the Right Graph

The kind of graph we choose determines how people view the associated data, so choosing the right graph from the outset is important.

we should choose,

- **Pie Chart**, if we want to show how various data elements **contribute towards a whole.**
- **Bar Chart**, if we want to **compare data elements.**
- **Histograms**, if we want to show the **distribution of elements.**

- **Box Plot**, if we want to **depict groups in elements**.
- **Scatter Plot**, if we want to **find patterns in data**.
- **Line Chart,** if we want to display **trends over time.**
- **Basemap,** if we want to display **geographical data.**
- **NetworkX,** if we want to **display network.**

Pie Chart

A pie chart (or a circle chart) is a circular statistical graphic, which is divided into slices to illustrate numerical proportion.

In a pie chart, the arc length of each slice is proportional to the quantity it represents.

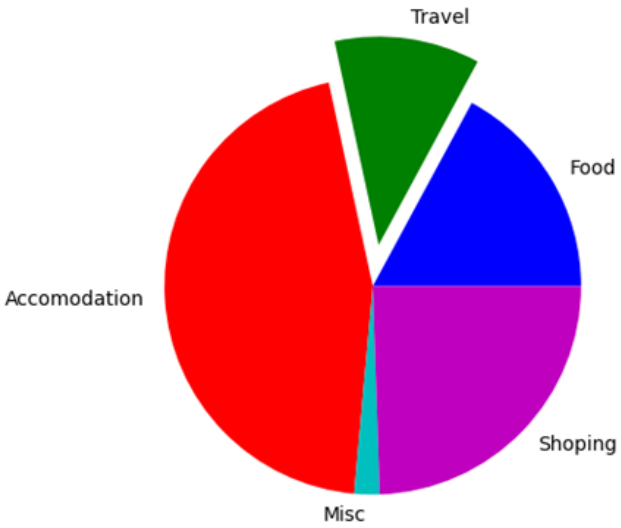
The pie chart focus on showing parts of a whole, the entire pie would be 100 percentages, the question is how much of that percentage each value occupies.

To draw a pie chart, we need to import Matplotlib library, and use its method named pie with the values and optional parameters like colors, labels, and explode.

Example:

```
import matplotlib.pyplot as plt
%matplotlib notebook
values = [305,201,805,35,436]
l = ['Food','Travel','Accomodation','Misc','Shoping']
c = ['b','g','r','c','m']
e = [0,0.2,0,0,0]
plt.pie(values,colors=c,labels=l,explode=e)
plt.show()
```

Output:



Bar Chart

A bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally. A vertical bar chart is sometimes called a column chart.

A bar graph shows comparisons among discrete categories. One axis of the chart shows the specific categories being compared, and the other axis represents a measured value.

Bar charts make comparing values easy, wide bars and segregated measurements emphasize the difference between values, rather than the flow of one value to another as a line graph.

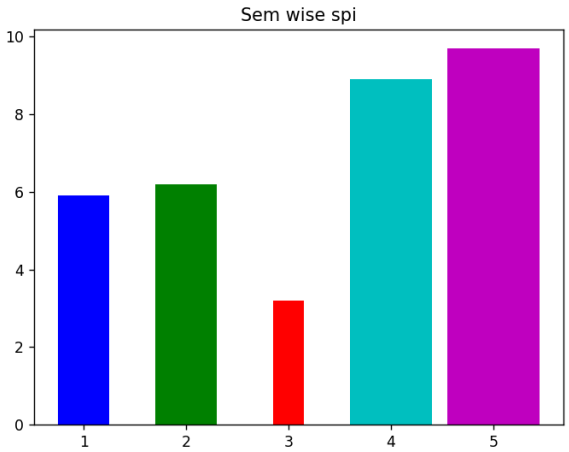
To draw a bar chart, we need to import Matplotlib library and use its bar/barh method with required parameters x and y, we can also supply other optional parameters like color, label, and width.

If we want to draw a vertical bar chart we can use **bar** method, and if we want to draw a horizontal bar chart we can use **barh** method.

Example:

```
import matplotlib.pyplot as plt
%matplotlib notebook
x = [1,2,3,4,5]
y = [5.9,6.2,3.2,8.9,9.7]
l = ['1st','2nd','3rd','4th','5th']
c = ['b','g','r','c','m']
w = [0.5,0.6,0.3,0.8,0.9]
plt.title('Sem wise spi')
plt.bar(x,y,color=c,label=l,width=w)
plt.show()
```

Output:



Histogram

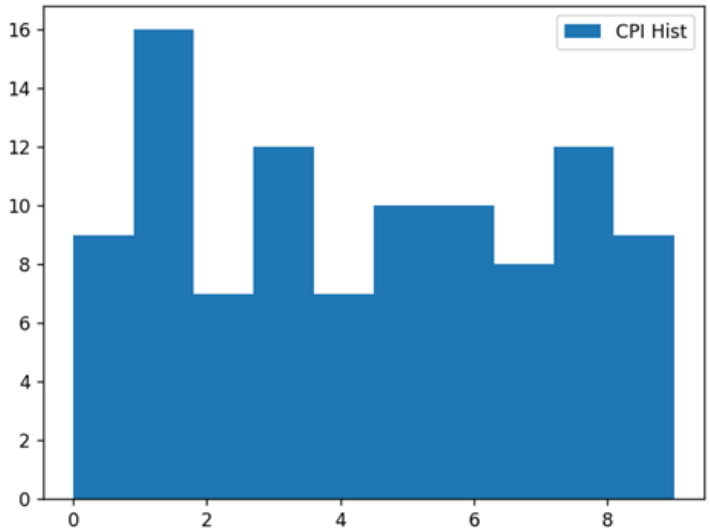
A histogram is an approximate representation of the distribution of numerical data. the first step is to divide the entire range of values into a series of intervals which is referred to as "bin" or "bucket", and then count how many values fall into each interval.

To draw a histogram in python, we can use hist method of Matplotlib library with the required x parameter and many optional parameters like bins, histtype, align, label etc..

Example:

```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib notebook
cpis = np.random.randint(0,10,100)
plt.hist(cpis,bins=10, histtype='stepfilled',align='mid',label='CPI
Hist')
plt.legend()
plt.show()
```

Output:



Boxplot

Boxplots provide a means of depicting groups of numbers through their quartiles, Quartiles means three points (25^{th} percentile, median, 75^{th} percentile) dividing a group into four equal parts.

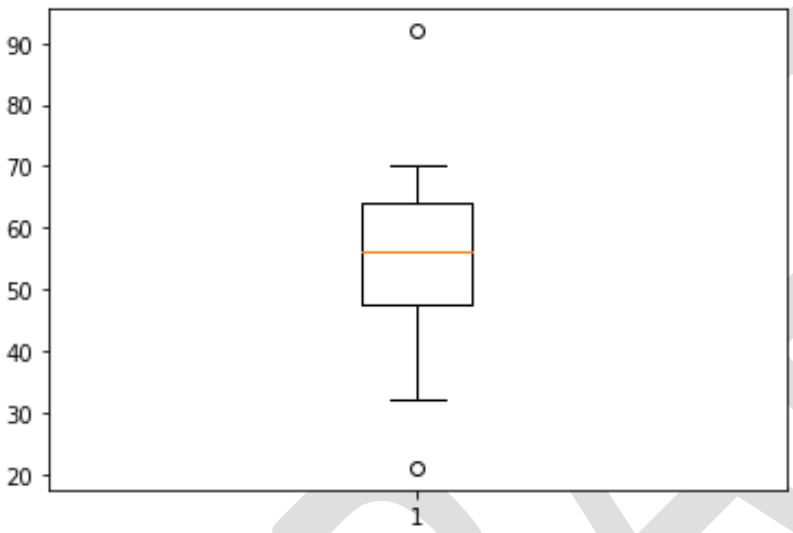
Boxplot basically used to detect **outliers** in the data, let's see an example where we need boxplot.

We have a dataset where we have time taken to check the paper, and we want to find the faculty which either takes more time or very little time to check the paper.

Example:

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
timetaken = pd.Series([50,45,52,63,70,21,56,68,54,57,35,62,65,92,32])
plt.boxplot(timetaken)
```

Output:



Scatter plot

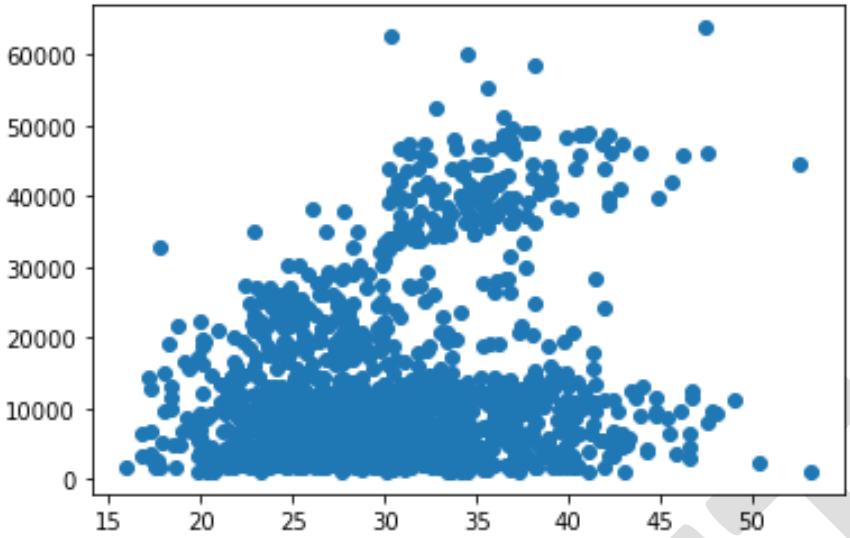
Scatter plot is a type of plot that shows the data as a collection of points, the position of a point depends on its two-dimensional value, where each value is a position on either the horizontal or vertical dimension, it is really useful in the study of the relationship or pattern between variables.

To draw scatter plot we can use Matplotlib library's scatter method with the x and y parameters.

Example:

```
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
df = pd.read_csv('insurance.csv')
plt.scatter(df['bmi'], df['charges'])
plt.show()
```

Output:



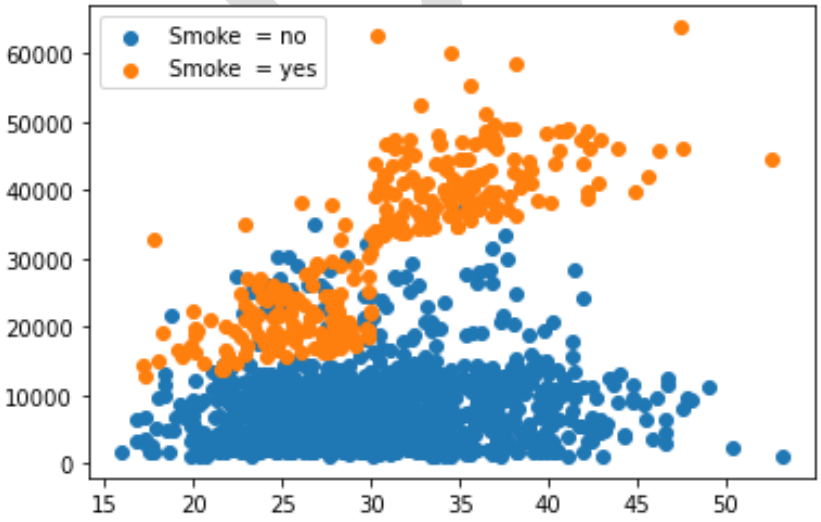
To find a specific pattern from the data, we can further divide the data and plot scatter plot.

We can do this with the help of groupby method of DataFrame and then using tuple unpacking while looping the group.

Example:

```
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
df = pd.read_csv('insurance.csv')
grouped = df.groupby(['smoker'])
for key, group in grouped:
   plt.scatter(group['bmi'], group['charges'], label='Smoke = '+key)
plt.legend()
plt.show()
```

Output:



we can specify marker, color, and size of the marker with the help of marker, color and s parameters respectively.

Time series

Observations over time can be considered as a Time Series, visualization plays an important role in time series analysis and forecasting, time Series plots can provide valuable diagnostics to identify temporal structures like trends, cycles, and seasonality.

To create a Time Series, we first need to get the date range, which can be created with the help of datetime and pandas library.

Example:

```
import pandas as pd
import datetime as dt
start_date = dt.datetime(2020,8,28)
end_date = dt.datetime(2020,9,05)
daterange = pd.date_range(start_date,end_date)
print(daterange)
```

Output:

```
DatetimeIndex(['2020-08-28', '2020-08-29', '2020-08-30', '2020-08-31', '2020-09-01',
'2020-09-02', '2020-09-03', '2020-09-04', '2020-09-05'],
dtype='datetime64[ns]', freq='D')
```

We can use some more parameters for date_range() function like

- freq, to specify the frequency at which we want the date range (default is 'D' for days)
- periods, number of periods to generate in between start/end or from start with freq.

Some of important possible values for the freq are

- D, for calendar day
- W, for week
- M, for month
- Y, for year
- H, for hour
- T/min, for minute
- S, for seconds
- L, for milliseconds
- B, for business day
- SM, for semi month-end
- Q, for quarter-end
- BQ, for business quarter-end

Basemap

One common type of visualization in data science is that of geographic data. Matplotlib's main tool for this type of visualization is the Basemap toolkit, which is one of several Matplotlib toolkits that lives under the mpl_toolkits namespace.

Admittedly, Basemap feels a bit clunky to use, and often even simple visualizations take much longer to render than we might hope. More modern solutions such as leaflet or the Google Maps API may be a better choice for more intensive map visualizations. Still, Basemap is a useful tool for Python users for some tasks.

Installation of Basemap is straightforward; if you're using conda you can execute "**conda install basemap**" command in the command prompt/terminal and the package will be downloaded.

NetworkX

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

Advantages of NetworkX,

- Data structures for graphs, digraphs, and multigraphs
- Many standard graph algorithms
- Network structure and analysis measures
- Generators for classic graphs, random graphs, and synthetic networks
- Nodes can be "anything" (e.g., text, images, XML records)
- Edges can hold arbitrary data (e.g., weights, time-series)
- Open-source
- Well tested with over 90% code coverage
- Additional benefits from Python include fast prototyping, easy to teach, and multi-platform

We can use networkx library in order to deal with any kind of networks, which includes a social network, railway network, road connectivity, etc….

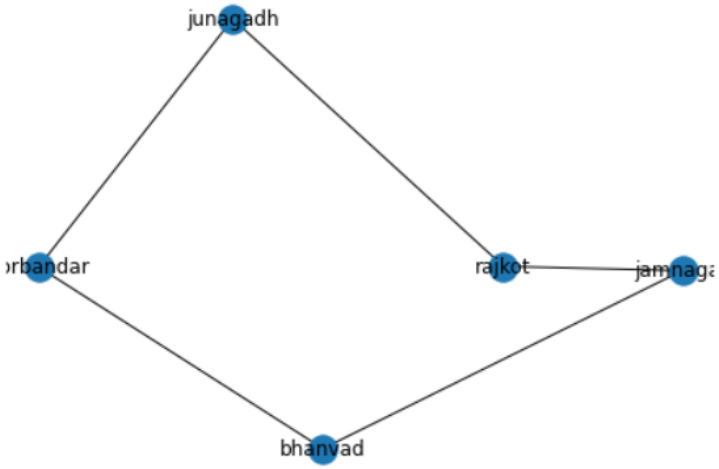
To install NetworkX we can use,

- pip install networkx **OR**
- conda install networkx

Example:

```
import networkx as nx
g = nx.Graph() # undirected graph
g.add_edge('rajkot','junagadh')
g.add_edge('junagadh','porbandar')
g.add_edge('rajkot','jamnagar')
g.add_edge('jamnagar','bhanvad')
g.add_edge('bhanvad','porbandar')
nx.draw(g,with_labels=True)
```

Output:

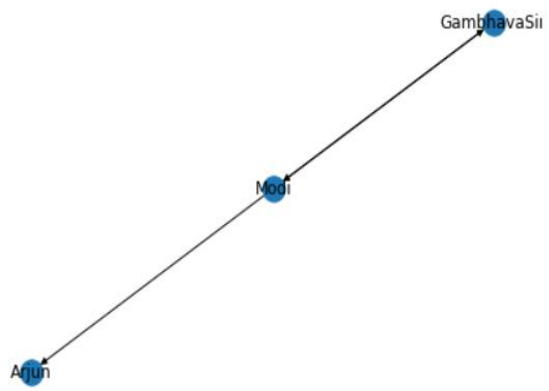


We can also use NetworkX's DiGraph to generate the directed graph,

Example:

```
import networkx as nx
gD = nx.DiGraph() # directed graph
gD.add_edge('Modi','Arjun')
gD.add_edge('Modi','GambhavaSir')
gD.add_edge('GambhavaSir','Modi')
nx.draw(gD, with_labels=True)
```

Output:



We can use many analysis functions available in NetworkX library, some of functions are as below,

- nx.shortest_path(g,'rajkot','porbandar')
  - Will return ['rajkot', 'junagadh', 'porbandar']
- nx.clustering(g)
  - Will return clustering value for each node
- nx.degree_centrality(g)
  - Will return the degree of centrality for each node, we can find the most popular/influential node using this method.
- nx.density(g)
  - Will return the density of the graph.
  - The density is 0 for a graph without edges and 1 for a complete graph.
- nx.info(g)
  - Return a summary of information for the graph G.
  - The summary includes the number of nodes and edges and their average degree.

Classes in Scikit-learn

- Understanding how classes work is an important prerequisite for being able to use the Scikit-learn package appropriately.
- Scikit-learn is the package for machine learning and data science experimentation favored by most data scientists.
- It contains wide range of well-established learning algorithms, error functions and testing procedures.
- Install : conda install scikit-learn
- There are four class types covering all the basic machine-learning functionalities,
  - Classifying
  - Regressing
  - Grouping by clusters
  - Transforming data
- Even though each base class has specific methods and attributes, the core functionalities for data processing and machine learning are guaranteed by one or more series of methods and attributes called interfaces.
- The interfaces provide a uniform Application Programming Interface (API) to enforce similarity of methods and attributes between all the different algorithms present in the package. There are four Scikit-learn object-based interfaces:
  - **estimator**: For fitting parameters, learning them from data, according to the algorithm.
  - **predictor**: For generating predictions from the fitted parameters
  - **transformer**: For transforming data, implementing the fitted parameters
  - **model**: For reporting goodness of fit or other score measures
- The package groups the algorithms built on base classes and one or more object interfaces into modules, each module displaying a specialization in a particular type of machine-learning solution. For example, the "**linear_model**" module is for linear modeling, and "**metrics**" is for score and loss measure.

Supervised V/S Unsupervised Learning

Supervised Learning	Unsupervised Learning
Supervised learning algorithms are trained using labeled data.	Unsupervised learning algorithms are trained using unlabeled data.

Supervised learning model takes direct feedback to check if it is predicting correct output or not.	Unsupervised learning model does not take any feedback.
The goal of supervised learning is to train the model so that it can predict the output when it is given new data.	The goal of unsupervised learning is to find the hidden patterns and useful insights from the unknown dataset.
Supervised learning can be categorized in **Classification** and **Regression** problems .	Unsupervised Learning can be classified in **Clustering** and **Associations** problems .
Supervised learning model produces an accurate result.	Unsupervised learning model may give less accurate result as compared to supervised learning.
It includes various algorithms such as Linear Regression, Logistic Regression, Support Vector Machine, Multi-class Classification, Decision tree, Bayesian Logic, etc.	It includes various algorithms such as Clustering, KNN, and Apriori algorithm.

Hashing Trick

- Most Machine Learning algorithms uses numeric inputs, if our data contains text instead we need to convert those text into **numeric values** first, this can be done using hashing tricks.
- For Example,

Id	Salary	Gender
1	10000	Male
2	15000	Female
3	12000	Male
4	11000	Female
5	20000	Male

Id	Salary	Gender	Gender_Num
1	10000	Male	1
2	15000	Female	0
3	12000	Male	1
4	11000	Female	0
5	20000	Male	1

- When dealing with text, one of the most useful solutions provided by the Scikit-learn package is the hashing trick.
- Example:

Actual Text

```
darshan is the best engineering college in rajkot
rajkot is famous for engineering
college is located in rajkot morbi road
```

Numeric Representation

```
1 2 3 4 5 6 7 8
8 2 9 10 5
6 2 11 7 8 12 13
```

Sparse Matrix

darshan	is	the	best	engineering	college	in	rajkot	famous	for	located	morbi	road
1	1	1	1	1	1	1	1	0	0	0	0	0
0	1	0	0	1	0	0	1	1	1	0	0	0
0	1	0	0	0	1	1	1	0	0	1	1	1

Example

```
a = ["darshan is the best engineering college in rajkot","rajkot
is famous for engineering","college is located in rajkot morbi
road"]
from sklearn.feature_extraction.text import *
countVector = CountVectorizer()
X_train_counts = countVector.fit_transform(a)
print(countVector.vocabulary_)
print(X_train_counts.toarray())
```

Output:

```
{'darshan': 2,
 'is': 7,
 'the': 12,
 'best': 0,
 'engineering': 3,
 'college': 1,
 'in': 6,
 'rajkot': 10,
 'famous': 4,
 'for': 5,
 'located': 8,
 'morbi': 9,
 'road': 11}
array([[1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1],
       [0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0],
       [0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0]], dtype=int64)
```

Considering Timing and Performance

- Managing the best use of machine resources is indeed an art, the art of optimization, and it requires time to master.
- However, we can start immediately becoming proficient in it by doing some accurate speed measurement and realizing what your problems really are.
- Profiling the time that operations require, measuring how much memory adding more data takes, or performing
- a transformation on your data can help you to spot the bottlenecks in your code and start looking for alternative solutions.
- IPython is the perfect environment for experimenting, tweaking, and improving your code.
- Working on blocks of code, recording the results and outputs, and writing additional notes and comments will help your data science solutions take shape in a controlled and reproducible way.

Benchmarking with timeit

- We can find the time taken to execute a statement or a cell in a Jupyter Notebook with the help of timeit magic command.
- This command can be used both as a line and cell magic:
  - In **line mode** you can time a single-line statement (though multiple ones can be chained with using semicolons).
    Syntax: %timeit [-n<N> -r<R> [-t|-c] -q -p<P> -o]
  - In **cell mode**, the statement in the first line is used as setup code (executed but not timed) and the body of the cell is timed. The cell body has access to any variables created in the setup code.
    Syntax: %%timeit [-n<N> -r<R> [-t|-c] -q -p<P> -o]
    Here, **-n** flag represents the number of loops and –**r** flag represents the number of repeats
    Example:

```
%%timeit -n 1000 -r 7

for i in range(2,1000) :
    listPrime  = []
    flag = 0
    for j in range(2,i) :
        if i % j == 0 :
            flag = 1
            break
    if flag == 0 :
        listPrime.append(i)
#print(listPrime)
```

Output:

```
5.17 ms ± 705 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Memory Profiler

- This is a python module for monitoring memory consumption of a process as well as line-by-line analysis of memory consumption for python programs.
- **Installation**: pip install memory_profiler
- **Usage**:
  - First Load the profiler by writing %load_ext memory_profiler in the notebook
  - Then write %memit on each statement you want to monitor memory.

Example:

```
%load_ext memory_profiler
from sklearn.feature_extraction.text import CountVectorizer
%memit countVector =
       CountVectorizer(stop_words='english',analyzer='word')
```

Output:

```
peak memory: 85.61 MiB, increment: 0.02 MiB
```

Running in parallel

- Most computers today are multicore (two or more processors in a single package), some with multiple physical CPUs.
- One of the most important limitations of Python is that it uses a single core by default. (It was created in a time when single cores were the norm.)
- Data science projects require quite a lot of computations.
- Part of the scientific aspect of data science relies on repeated tests and experiments on different data matrices.
- Also the data size may be huge, which will take lots of processing power.

- Using more CPU cores accelerates a computation by a factor that almost matches the number of cores.
- For example, having four cores would mean working at best four times faster, but practically we will not have four times faster processing as assigning the different cores to different processes will take some amount of time.
- So parallel processing will work best with huge datasets or where extreme processing power is required.
- In Scikit-learn library we need not to do any special programming in order to do the parallel (multiprocessing) processing, we can simply use **n_jobs** parameter to do so.
- If we set number grater than 1 in the **n_jobs** it will uses that much of the processor, if we set -1 to **n_jobs** it will use all available processor from the system.
- Let's see the Example how we can achieve parallel processing in Scikit-learn library.
- Example (Without parallel processing)

```
%%timeit -n 1 -r 1
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_info
rmative=15, n_redundant=5, random_state=3)
# define the model
model = RandomForestClassifier(n_estimators=500, n_jobs=1)
```

Output:

```
25.2 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

- Example (With parallel processing)

```
%%timeit -n 1 -r 1
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
# define dataset
X, y = make_classification(n_samples=10000, n_features=20, n_info
rmative=15, n_redundant=5, random_state=3)
# define the model
model = RandomForestClassifier(n_estimators=500, n_jobs=-1)
```

Output:

```
12.8 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Exploratory Data Analysis (EDA)

- Exploratory Data Analysis (EDA) refers to the critical process of performing initial investigations on data so as to discover patterns, to spot anomalies, to test hypothesis

and to check assumptions with the help of summary statistics and graphical representations.

- EDA was developed at Bell Labs by John Tukey, a mathematician and statistician who wanted to promote more questions and actions on data based on the data itself.
- In one of his famous writings, Tukey said:
  "*The only way humans can do **BETTER** than computers is to take a chance of doing **WORSE** than them.*"
- Above statement explains why, as a data scientist, your role and tools aren't limited to automatic learning algorithms but also to manual and creative exploratory tasks.
- Computers are unbeatable at optimizing, but humans are strong at discovery by taking unexpected routes and trying unlikely but very effective solutions.
- With EDA we can,
  - Describe data
  - Closely explore data distributions
  - Understand the relationships between variables
  - Notice unusual or unexpected situations
  - Place the data into groups
  - Notice unexpected patterns within the group
  - Take note of group differences

Measuring Central Tendency

We can use many inbuilt pandas functions in order to find central tendency of the numerical data, Some of the functions are,

- df.mean()
- df.median()
- df.std()
- df.max()
- df.min()
- df.quantile(np.array([0,0.25,0.5,0.75,1]))

Normality

We can define normality of the data using below measures,

- **Skewness** defines the asymmetry of data with respect to the mean.
  - It will be negative if the left tail is too long.
  - It will be positive if the right tail is too long.
  - It will be zero if left and right tail are same.

- **Kurtosis** shows whether the data distribution, especially the peak and the tails, are of the right shape.
  - It will be positive if distribution has marked peak.
  - It will be zero if distribution is flat.

Frequencies

We can use pandas built-in function value_counts() to obtain the frequency of the categorical variables

Describe

We also can use another pandas built-in function describe() to obtain insights of the data.

Visualization

We can use many types of graphs, some of them are listed below,

- if we want to show how various data elements contribute towards a whole, we should use pie chart.
- If we want to compare data elements, we should use bar chart.
- If we want to show distribution of elements, we should use histograms.
- If we want to depict groups in elements, we should use boxplots.
- If we want to find patterns in data, we should use scatterplots.
- If we want to display trends over time, we should use line chart.
- If we want to display geographical data, we should use basemap.
- If we want to display network, we should use networkx.

In addition to all these graphs, we also can use seaborn library to plot advanced graphs like,

- Countplot
- Heatmap
- Distplot
- Pairplot

Covariance

- Covariance is a measure used to determine how much two variables change in tandem.
- The unit of covariance is a product of the units of the two variables.
- Covariance is affected by a change in scale, The value of covariance lies between -∞ and +∞.
- Pandas does have built-in function to find covariance in DataFrame named cov()

Correlation

- The correlation between two variables is a normalized version of the covariance.
- The value of correlation coefficient is always between -1 and 1.
- Once we've normalized the metric to the -1 to 1 scale, we can make meaningful statements and compare correlations.