PYTHON PROGRAMMING

What is Python:

- Python is a general purpose programming language created by Guido Van Rossum.
- Python is most praised for its elegant syntax and readable code, if you are just beginning your programming career python suits you best.
- With python you can do everything from GUI development, Web application, System administration tasks, Financial calculation, Data Analysis, Visualization and list goes on.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-orientated way or a functional way.

Python Syntax compared to other programming languages

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Python is interpreted language

- Yes, python is interpreted language, when you run python program an interpreter will parse python program line by line basis, as compared to compiled languages like C or C++, where compiler first compiles the program and then start running.

Python is Dynamically Typed

- In python you don't need to define variable data type ahead of time, python automatically guesses the data type of the variable based on the type of value it contains.
- EG.
  - myvar = "Hello Python"
  - In the above line "Hello Python" is assigned to myvar , so the type of myvar is string.
    Note that in python you do not need to end a statement with a semicolon (;) .

- Suppose little bit later in the program we assign myvar a value of 1 i.e

  eg.

  myvar = 1

  now myvar is of type int.

Python is strongly typed

- If you have programmed in php or javascript. You may have noticed that they both convert data of one data type to other data type automatically.
- For e.g:
- in JavaScript

```
1 + "2“
```

will be '12'

- In Python automatic conversions are not allowed, so

  eg.

  myvar=1 + "2“

  will produce an error.

Write less code and do more

- Python codes are usually 1/3 or 1/5 of the java code. It means we can write less code in Python to achieve the same thing as in Java.
- In python to read a file you only need 2 lines:
- Eg.
    - with open("myfile.txt") as f:
      print(f.read())

Who uses python:

- Python is used by many large organization like Google, NASA, Quora, HortonWorks and many others.
  - GUI application.
  - Create Websites.
  - Scrape data from website.
  - Analyze Data.
  - System Administration Task.
  - Game Development.
  - and many more …

Multiple Assignment

- Python allows you to assign a single value to several variables simultaneously.

  a=b=c=1

  here , an integer object is created with the value 1,and all three variables are assigned to the same memory location.

- You can also assign multiple objects to multiple variables.
- For example:

  a,b,c=1,2,"John"

  Here two integer objects with values 1 and 2 are assigned to variables a and b respectively,and one string object with the value "john" is assigned to variable c.

Standard data types

- Python has various standard data types that are used to define the operation possible on them and the storage method for each of them.
    - Numbers
    - String
    - List
    - Tuple
    - Dictionary

Python numbers

- Number data type store numeric value. Number object are created when you assign a value to them.
- For ex.
  - Var1=10
  - Var2=20

- Python supports four different numerical data types:
  - Int(signed integer) (10)
  - Long(octal and hexadecimal)(51924361L)
  - Float(floating point real values)(15.20)
  - Complex number(3.14j)

Python string

- String in python are identified as a contiguous set of characters represented in quotation mark
    - Mystring=“hello world”

Python list

- A list contains items are separated by comma and enclosed within square brackets([]).
- To some extent list are similar to array.
- One difference is that array stores data of same type and list stores data of different type.
- Eg:
  - List[‘Mukesh’,10,20,’ramesh’,90]

Python Tuples

- A tuple is another sequence data type that is similar to list.
- A tuple consist of a number of values separated by commas.
- Tuples are enclosed with parenthesis().
- Tuples can not be updated it can be thought as read only data.

Strings in python are immutable.

- What this means to you is that once string is created it can't be modified. Let's take an example to illustrate this point.
  - >>> str1 = "welcome"
  - >>> str2 = "welcome“
  - here str1 and str2 refers to the same string object "welcome" which is stored somewhere in memory. You can test whether str1 refers to same object as str2 using id() function.

- **What is** id() : Every object in python is stored somewhere in memory. We can useid() to get that memory address.
  - >>> id(str1)
  - 78965411
  - >>> id(str2)
  - 78965411

Operations on string

- String index starts from o , so to access the first character in the string type:
  - >>> name[o]
  - T

  + operator is used to concatenate string and * operator is a repetition operator for string.
  - >>> s = "tom and " + "jerry"
  - >>> print(s)
  - tom and jerry

- >>> s = "this is bad spam " * 3
- >>> print(s)
- this is bad spam this is bad spam this is bad spam
- **Slicing string**
- You can take subset of string from original string by using [] operator also known as slicing operator.
- **Syntax**: s[start:end]
- this will return part of the string starting from index start to index end .

- Eg.
  - >>> s = "Welcome"
  - >>> s[1:3]
  - El
  - Eg.
  - >>> s = "Welcome"
  - >>> s[: 6]
  - 'Welcom'

  - >>> s[4 :]
  - 'ome'

  - >>> s[1 : -1]
    'elcom'

ord() and chr() Functions

- **ord**() – function returns the ASCII code of the character.
- **chr**() – function returns character represented by a ASCII number.
  - >>> ch = 'b'
  - >>> ord(ch)
  - 98
  - >>> chr(97)
  - 'a'
  - >>> ord('A')
  - 65

String Functions in Python

- FUNCTION NAME FUNCTION DESCRIPTION
- len() returns length of the string
- max() returns character having highest ASCII value
- min() returns character having lowest ASCII value

- Eg.
- >>> len("hello")
- 5
- >>> max("abc")
- 'c'
- >>> min("abc")
- 'a'

in **and** not in **operators**

- You can use in and not in operators to check existence of string in another string. They are also known as membership operator.
  - >>> s1 = "Welcome"
  - >>> "come" in s1
  - True
  - >>> "come" not in s1
  - False
  - >>>

String comparison

- You can use (> , < , <= , <= , == , !=) to compare two strings. Python compares string lexicographically i.e using ASCII value of the characters.
  - >>> "tim" == "tie"
  - False
  - >>> "free" != "freedom"
  - True
  - >>> "arrow" > "aron"
  - True
  - >>> "right" >= "left"
  - True

- >>> "teeth" < "tee"
- False
- >>> "yellow" <= "fellow"
- False
- >>> "abc" > ""
- True
- >>>

Iterating string using for loop

- String is a sequence type and also iterable using for loop.
- Eg

```
>>> s = "hello"
>>> for i in s:
... print(i, end="")
hello
```

Testing strings

- String class in python has various inbuilt methods which allows to check for different types of strings.
- **METHOD NAME** **METHOD DESCRIPTION**
- isalnum() Returns True if string is alphanumeric
- isalpha() Returns True if string contains only alphabets
- isdigit() Returns True if string contains only digits
- isidentifier() Return True is string is valid identifier
- islower() Returns True if string is in lowercase
- isupper() Returns True if string is in uppercase
- isspace() Returns True if string contains only whitespace

Searching for Substrings

- **METHOD NAME** **METHODS DESCRIPTION:**
- endswith(s1: str):bool Returns True if strings ends with substring s1
- startswith(s1: str):bool Returns True if strings starts with substring s1
- count(substring):int Returns number of occurrences of substring the string
- find(s1): int Returns lowest index from where s1 starts in the string, if string not found returns 1
- rfind(s1): int Returns highest index from where s1 starts in the string, if string not found returns -1

Converting Strings

METHOD NAME	METHOD DESCRIPTION
capitalize(): str	Returns a copy of this string with only the first character capitalized.
lower(): str	Return string by converting every character to lowercase
upper(): str	Return string by converting every character to uppercase
title(): str	This function return string by capitalizing first letter of every word in the string
swapcase(): str	Return a string in which the lowercase letter is converted to uppercase and uppercase to lowercase
replace(old, new): str	This function returns new string by replacing the occurrence of old string with new string

List Common Operations

METHOD NAME	DESCRIPTION
x in s	True if element x is in sequences.
x not in s	if element x is not in sequence
s1 + s2	Concatenates two sequences s1 and s2
s * n , n * s	n copies of sequence s concatenated
s[i]	ith element in sequences.
len(s)	Length of sequence s, i.e. the number of elements in s.
min(s)	Smallest element in sequences.
max(s)	Largest element in sequences.
sum(s)	Sum of all numbers in sequences.
for loop	Traverses elements from left to right in a for loop.

List slicing

- Slice operator ([start:end]) allows to fetch sublist from the list. It works similar to string.
- Eg.

  >>> list = [11,33,44,66,788,1]

  - >>> list[0:5] # this will return list starting from index 0 to index 4
  - [11,33,44,66,788]

+ and * operators in list

- + operator joins the two list
- Eg.
  - >>> list1 = [11, 33]
  - >>> list2 = [1, 9]
  - >>> list3 = list1 + list2
  - >>> list3
  - [11, 33, 1, 9]

- * operator replicates the elements in the list.
- Eg.
  - >>> list4 = [1, 2, 3, 4]
  - >>> list5 = list4 * 3
  - >>> list5
  - [1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]

in **or** not in **operator**

- in operator is used to determine whether the elements exists in the list. On success it returns True on failure it returns False .
- Eg.
  - >>> list1 = [11, 22, 44, 16, 77, 98]
  - >>> 22 in list1
  - True

- Similarly not in is opposite of in operator.
- Eg.
  - >>> 22 not in list1
  - False

Traversing list using for loop

- As already discussed list is a sequence and also iterable. Means you can use for loop to loop through all the elements of the list.
- Eg.
  - >>> list = [1,2,3,4,5]
  - >>> for i in list:
  - ... print(i, end=" ")
  - 1 2 3 4 5

Commonly used list methods with return type

METHOD NAME	DESCRIPTION
append(x:object):None	Adds an element x to the end of the list and returns None.
count(x:object):int	Returns the number of times element x appears in the list.
extend(l:list):None	Appends all the elements in l to the list and returns None.
index(x: object):int	Returns the index of the first occurrence of element x in the list

- insert(index: int, x:object): None Inserts an element x at a given index. Note that the first element in the list has index o and returns None..
- remove(x:object):None Removes the first occurrence of element x from the list and returns None
- reverse():None Reverse the list and returns None
- sort(): None Sorts the elements in the list in ascending order and returns None.
- pop(i): object Removes the element at the given position and returns it. The parameter i is optional. If it is not specified, pop() removes and returns the last element in the list.

List Comprehension

- List comprehension provides a concise way to create list. It consists of square brackets containing expression followed by for clause then zero or more for or if clauses.
- Ex.
- >>> list1 = [x for x in range(10)]
- >>> list1
- [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

- >>> list2 = [x + 1 for x in range(10)]
- >>> list2
- [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

- >>> list3 = [x for x in range(10) if x % 2 == 0]
- >>> list3
- [0, 2, 4, 6, 8]

- >>> list4 = [x *2 for x in range(10) if x % 2 == 0]
- [0, 4, 8, 12, 16]

Python Dictionaries

- Dictionary is a python data type that is used to store key value pairs. It enables you to quickly retrieve, add, remove, modify, values using key. Dictionary is very similar to what we call associative array or hash on other languages.
- **Note**: Dictionaries are mutable.
- **Creating Dictionary**
  - Dictionaries can be created using pair of curly braces ({}). Each item in the dictionary consist of key, followed by a colon, which is followed by value. And each item is separated using commas (,). Let's take an example.

- Eg.
  - friends = {
  - 'tom' : '111-222-333',
  - 'jerry' : '666-33-111'
  - }
  - here friends is a dictionary with two items. One point to note that key must be of hashable type, but value can be of any type. Each key in the dictionary must be unique.

Retrieving, modifying and adding elements in the dictionary

- To get an item from dictionary, use the following syntax:
- >>> dictionary_name['key']
- Eg.
  - >>> friends['tom']
  - '111-222-333‘

- To add or modify an item, use the following syntax:
- >>> dictionary_name['newkey'] = 'newvalue'
- Eg:
  - >>> friends['bob'] = '888-999-666'
  - >>> friends

    {'tom': '111-222-333', 'bob': '888-999-666', 'jerry': '666-33-111'}
- Deleting Items from dictionary.

  >>> del dictionary_name['key']

- **Looping items in the dictionary**
- You can use for loop to traverse elements in the dictionary.
- Eg.
  - >>> friends = {
  - ... 'tom' : '111-222-333',
  - ... 'jerry' : '666-33-111'
  - ...}
  - >>>
  - >>> for key in friends:
  - ... print(key, ":", friends[key])
  - ...
  - tom : 111-222-333
  - jerry : 666-33-111
  - >>>
  - >>>

- **Find the length of the dictionary**
- You can use len() function to find the length of the dictionary.
- Eg.
  - >>> len(friends)
  - 2

- **in or not in operators**
- in and not in operators to check whether key exists in the dictionary.

- Eg.
  - >>> 'tom' in friends
  - True
  - >>> 'tom' not in friends
  - False

- **Equality Tests in dictionary**
- == and != operators tells whether dictionary contains same items not.
- Eg.
  - >>> d1 = {"mike":41, "bob":3}
  - >>> d2 = {"bob":3, "mike":41}
  - >>> d1 == d2
  - True
  - >>> d1 != d2
  - False
  - >>>
  - **Note: You can't use other relational operators like < , > , >= , <= to compare dictionaries.**

- **Dictionary methods**
- Python provides you several built-in methods for working with dictionaries.

METHODS	DESCRIPTION
popitem()	Returns randomly select item from dictionary and also remove the selected
clear()	Delete everything from dictionary
keys()	Return keys in dictionary as tuples
values()	Return values in dictionary as tuples
get(key)	Return value of key, if key is not found it returns None, instead on throwing KeyError exception
pop(key)	Remove the item from the dictionary, if key is not found KeyError will be thrown

Python Tuples

- In Python Tuples are very similar to list but once a tuple is created, you cannot add, delete, replace, reorder elements.
- **Note**: Tuples are immutable.
- **Creating a tuple**
  - >>> t1 = () # creates an empty tuple with no data
  - >>> t2 = (11,22,33)
  - >>> t3 = tuple([1,2,3,4,4]) # tuple from array
  - >>> t4 = tuple("abc") # tuple from string

- **Tuples functions:**
- Functions like max , min , len , sum can also be used with Tuples.
- Eg.
  - >>> t1 = (1, 12, 55, 12, 81)
  - >>> min(t1)
  - 1
  - >>> max(t1)
  - 81
  - >>> sum(t1)
  - 161
  - >>> len(t1)
  - 5

- **Iterating through Tuples**
- Tuples are iterable using for loop
  - >>> t = (11,22,33,44,55)
  - >>> for i in t:
  - … print(i, end=" ")
  - >>> 11 22 33 44 55

- **Slicing Tuples**
- Slicing operators works same in tuples as in list and string.
  - >>> t = (11,22,33,44,55)
  - >>> t[0:2]
  - (11,22)

- in **and** not in **operator**
- You can use in and not in operators to check existence of item in tuples as follows.
  - >>> t = (11,22,33,44,55)
  - >>> 22 in t
  - True
  - >>> 22 not in t
  - False

Datatype conversion

- Once in a while you will want to convert data type of one type to another type. Data type conversion is also known as Type casting.
- **Converting int to float**
- To convert int to float you need to use float() function.
  - >>> i = 10
  - >>> float(i)
  - 10.0

- **Converting float to int**
- To convert float to int you need to use int() function.
  - >>> f = 14.66
  - >>> int(f)
  - 14
- **Converting string to int**
- You can also use int() to convert string to int
  - >>> s = "123"
  - >>> int(s)
  - 123
  - **Note**: If string contains non numeric character then int() will throw ValueError.

- **Converting number to string**
- To convert number to string you need to use str() function.
  - >>> i = 100
  - >>> str(i)
  - "100"
  - >>> f = 1.3
  - str(f)
  - '1.3‘

- **Rounding numbers:**
- To round numbers you need to use round() function.
- **Syntax**: round(number,ndigits)
  - >>> i = 23.97312
  - >>> round(i, 2)
  - 23.97

Python Control Statements

- It is very common for programs to execute statements based on some conditions. In this section we will learn about python **if .. else ...** statement.
- But before we need to learn about relational operators. Relational operators allows us to compare two objects.
- **SYMBOL** **DESCRIPTION**
  - <= smaller than or equal to
  - < smaller than
  - > greater than
  - >= greater than or equal to
  - == equal to
  - != not equal to

- The result of comparison will always be a boolean value i.e True or False . Remember True and False are python keyword for denoting boolean values.
- The syntax of If statement is:
  - **if boolean-expression:**

    #statements
  - **else:**

    #statements

Note: Each statements in the if block must be indented using the same number of spaces, otherwise it will lead to syntax error. This is very different from many other languages like Java, C, C# where curly braces ({}) is used.

- i = 10
- if i % 2 == 0:

  print("Number is even")
- else:

  print("Number is odd")

Note: else clause is optional you can use only if clause if you want, like this

- if today == "party":
- print("thumbs up!")

- If your programs needs to check long list of conditions then you need to use if-elif-else statements.
  - if boolean-expression:
  - #statements
  - elif boolean-expression:
  - #statements
  - elif boolean-expression:
  - #statements
  - elif boolean-expression:
  - #statements
  - else:
  - #statements

```
today = "monday"

if today == "monday":
    print("this is monday")
elif today == "tuesday":
    print("this is tuesday")
elif today == "wednesday":
    print("this is wednesday")
elif today == "thursday":
    print("this is thursday")
elif today == "friday":
    print("this is friday")
elif today == "saturday":
    print("this is saturday")
elif today == "sunday":
    print("this is sunday")
else:
    print("something else")
```

- **Nested if statements:**
- You can nest if statements inside another if statements as follows:

```
today = "holiday"
bank_balance = 25000
if today == "holiday":
    if bank_balance > 20000:
        print("Go for shopping")
    else:
        print("Watch TV")
else:
    print("normal working day")
```

Python Functions

- Functions are the re-usable pieces of code which helps us to organize structure of the code. We create functions so that we can run a set of statements multiple times during in the program without repeating ourselves.
- **Creating functions**
- Python uses **def** keyword to start a function, here is the syntax:
  - **def function_name(arg1, arg2, arg3, …. argN):**
  - **#statement inside function**

- **Note**: All the statements inside the function should be indented using equal spaces. Function can accept zero or more arguments(also known as parameters) enclosed in parentheses. You can also omit the body of the function using the **pass** keyword, like this:
- Eg.

```
def myfunc():
    pass
def sum(start, end):
    result = 0
    for i in range(start, end + 1):
        result += i
    print(result)

sum(10, 50)
```

- **Function with return value.**
- The above function simply prints the result to the console, what if we want to assign the result to a variable for further processing ? Then we need to use the return statement. The return statement sends a result back to the caller and exits the function.
  - def sum(start, end):
  - result = 0
  - for i in range(start, end + 1):
  - result += i
  - return result
  -
  - s = sum(10, 50)
  - print(s)

- You can also use the return statement without a return value.
  - def sum(start, end):
  - if(start > end):
  - print("start should be less than end")
  - return # here we are not returning any value so a special value None is returned
  - result = 0
  - for i in range(start, end + 1):
  - result += i
  - return result
  -
  - s = sum(110, 50)
  - print(s)

- In python if you do not explicitly return value from a function , then a special value None is always returned. Let's take an example
  - def test(): # test function with only one statement
  - i = 100
  - print(test())

Global variables vs local variables

- **Global variables**: Variables that are not bound to any function , but can be accessed inside as well as outside the function are called global variables.
- **Local variables**: Variables which are declared inside a function are called local variables.
- **Example 1**:
  - global_var = 12 # a global variable
  - def func():
  - local_var = 100 # this is local variable
  - print(global_var) # you can access global variables in side function
  - func() # calling function func()
  -
  - #print(local_var)

- **Argument with default values**
- To specify default values of argument, you just need to assign a value using assignment operator.
  - def func(i, j = 100):
  - print(i, j)

Keyword arguments

- There are two ways to pass arguments to method: **positional arguments** and **Keyword arguments**. We have already seen how positional arguments work in the previous section. In this section we will learn about keyword arguments.
- Keyword arguments allows you to pass each arguments using name value pairs like this name=value . Let's take an example:
  - def named_args(name, greeting):
  - print(greeting + " " + name)
  - named_args(name='jim', greeting='Hello')

- **Returning multiple values from Function**
- We can return multiple values from function using the return statement by separating them with a comma (,). Multiple values are returned as tuples.
  - def bigger(a, b):
  - if a > b:
  - return a, b
  - else:
  - return b, a
  -
  - s = bigger(12, 100)
  - print(s)
  - print(type(s))

Python Loops

- Python has only two loops: for loop and while loop
- **For loop**
- For loop Syntax:
  - for i in iterable_object:
  - # do something
- **Note**: all the statements inside for and while loop must be indented to the same number of spaces. Otherwise SyntaxError will be thrown.
- Let's take an example:
  - my_list = [1,2,3,4]
  - for i in my_list:
  - print(i)

- **range(a, b) Function**
- range(a, b) functions returns sequence of integers from a , a + 1 , a+ 2 …. , b -2 , b - 1 . For e.g
  - for i in range(1, 10):
  - print(i)

- You can also use range() function by supplying only one argument like this:
  - >>> for i in range(10):
  - … print(i)

- range(a, b) function has an optional third parameter to specify the step size. For e.g
  - for i in range(1, 20, 2):
  - print(i)
- **<u>While loop</u>**
- Syntax:
  - while condition:
  - # do something
- While loop keeps executing statements inside it until condition becomes false. After each iteration condition is checked and if its True then once again statements inside the while loop will be executed.

```
count = 0

while count < 10:
    print(count)
    count += 1
```

- **break statement**
- break statement allows to breakout out of the loop.

```
count = 0

while count < 10:
    count += 1
    if count == 5:
        break
    print("inside loop", count)

print("out of while loop")
```

- **continue statement**
- When continue statement encountered in the loop, it ends the current iteration and programs control goes to the end of the loop body.
  - count = 0
  - while count < 10:
  - count += 1
  - if count % 2 == 0:
  - continue
  - print(count)
-

Python Mathematical Function

METHOD	DESCRIPTION
round(number[ndigits])	rounds the number, you can also specify precision in the second argument
pow(a, b)	Returns a raise to the power of b
abs(x)	Return absolute value of x
max(x1, x2, …, xn)	Returns largest value among supplied arguments
min(x1, x2, …, xn)	Returns smallest value among supplied arguments

METHOD	DESCRIPTION
ceil(x)	This function rounds the number up and returns its nearest integer.
floor(x)	This function rounds the down up and returns its nearest integer
sqrt(x)	Returns the square root of the number
sin(x)	Returns sin of x where x is in radian
cos(x)	Returns cosine of x where x is in radian
tan(x)	Returns tangent of x where x is in radian

- **Python Generating Random numbers:**
- Python random module contains function to generate random numbers. So first you need to import random module using the following line.
- import random
- **random() Function**
- random() function returns random number r such that 0 <= r < 1.0
  - >>> import random
  - >>> for i in range(0, 10):
  - … print(random.random())

- randint(a, b) generate random numbers between a and b.
  - >>> import random
  - >>> for i in range(0, 10):
  - … print(random.randint(1, 10))

Python File Handling

- We can use File handling to read and write data to and from the file.
- **Opening a file**
- Before reading/writing you first need to open the file. Syntax of opening a file is.
- f = open(filename, mode)
- open() function accepts two arguments filename and mode . filename is a string argument which specify filename along with it's path and mode is also a string argument which is used to specify how file will be used i.e for reading or writing. And f is a file handler object also known as file pointer.

- **Closing a file**
- After you have finished reading/writing to the file you need to close the file using close() method like this,
- f.close()

- **Different modes of opening a file are**

MODES	DESCRIPTION
"r"	Open a file for read only
"w"	Open a file for writing. If file already exists its data will be cleared before opening. Otherwise new file will be created.
"a"	Opens a file in append mode i.e to write a data to the end of the file
"wb"	Open a file to write in binary mode
"rb"	Open a file to read in binary mode

- **Writing data to the file**
  - >>> f = open('myfile.txt', 'w')
  - >>> f.write('this first line\n')
  - >>> f.write('this second line\n')
  - >>> f.close()
- **Reading data from the file**
- To read data back from the file you need one of these three methods.

METHODS	DESCRIPTION
read([number])	Return specified number of characters from the file. if omitted it will read the entire contents of the file.
readline()	Return the next line of the file.
readlines()	Read all the lines as a list of strings in the file

- **Reading all the data at once.**
- >>> f = open('myfile.txt', 'r')
- >>> f.read()
- "this first line\nthis second line\n"
- >>> f.close()
- **Reading all lines as an array.**
- >>> f = open('myfile.txt', 'r')
- >>> f.readlines()
- ["this first line\n", "this second line\n"]
- >>> f.close()

- Reading only one line.
- >>> f = open('myfile.txt', 'r')
- >>> f.readline()
- "this first line\n"
- >>> f.close()
- **Appending data**
- To append the data you need open file in 'a' mode.
- >>> f = open('myfile.txt', 'a')
- >>> f.write("this is third line\n")
- 19
- >>> f.close()

- **Looping through the data using for loop**
- You can iterate through the file using file pointer.
  - >>> f = open('myfile.txt', 'r')
  - >>> for line in f:
  - … print(line)
  - …
  - this first line
  - this second line
  - this is third line
  -
  - >>> f.close()
-

- **Binary reading and writing**
- To perform binary i/o you need to use a module called pickle , pickle module allows you to read and write data using load and dump method respectively.
- **Writing data**
- >> import pickle
- >>> f = open('pick.dat', 'wb')
- >>> pickle.dump(11, f)
- >>> pickle.dump("this is a line", f)
- >>> pickle.dump([1, 2, 3, 4], f)
- >>> f.close()

- **Reading data**
- >> import pickle
- >>> f = open('pick.dat', 'rb')
- >>> pickle.load(f)
- 11
- >>> pickle.load(f)
- "this is a line"
- >>> pickle.load(f)
- [1,2,3,4]
- >>> f.close()
- If there is no more data to read from the file pickle.load() throws EOFError or end of file error.

- **Python Arbitrary Arguments**
- Sometimes, we do not know in advance the number of arguments that will be passed into a function.Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.
- In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument.
- def greet(*names):
  - print("Hello",name)
  - greet("Monica","Luke","Steve","John")

- **Python Recursive Function**
- a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.
- def calc_factorial(x):
  - if x == 1:
  - return 1
  - else:
  - return (x * calc_factorial(x-1))
- num = 4
- print("The factorial of", num, "is“,calc_factorial(num))

Advantages of Recursion

- Recursive functions make the code look clean and elegant.
- A complex task can be broken down into simpler sub-problems using recursion.
- Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

- Sometimes the logic behind recursion is hard to follow through.
- Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- Recursive functions are hard to debug.

Object-oriented vs Procedure-oriented Programming languages

Index	Object-oriented Programming	Procedural Programming
1.	Object-oriented programming is the problem-solving approach and used where computation is done by using objects.	Procedural programming uses a list of instructions to do computation step by step.
2.	It makes the development and maintenance easier.	In procedural programming, It is not easy to maintain the codes when the project becomes lengthy.
3.	It simulates the real world entity. So real-world problems can be easily solved through oops.	It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions.

4.	It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere.	Procedural language doesn't provide any proper way for data binding, so it is less secure.
5.	Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc.	Example of procedural languages are: C, Fortran, Pascal, VB etc.

Python Object and Classes

- **Creating object and classes**
- Python is an object-oriented language. In python everything is object i.e int , str ,bool even modules, functions are also objects.
- Object oriented programming use objects to create programs, and these objects stores data and behaviors.
- Objects are an encapsulation of variables and functions into a single entity. Objects get their variables and functions from classes. Classes are essentially a template to create your objects.

Creating a class

```
class MyClass:
    variable = "blah"
    def function(self):
        print("This is a message inside the class.")
```

Creating a object of a class

```
class MyClass:
    variable = "blah"
    def function(self):
        print("This is a message inside the class.")
    myobjectx = MyClass()
```

Accessing a variable of class

```
class MyClass:
    variable = "blah"
    def function(self):
        print("This is a message inside the class.")
    myobjectx = MyClass()
    myobjectx.variable
```

OOP

- The popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).
- An object has two characteristics:
  - attributes
  - Behavior
- The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

Class

- A class is a blueprint for the object.
- The example for class of parrot can be :
- class Parrot:
  - Pass

Object:

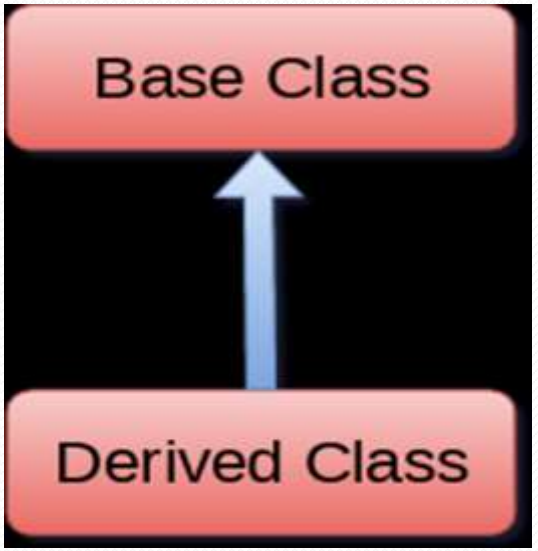
- An object is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.
- The example for object of parrot class can be:
- obj = Parrot()

- **Inheritance**
- Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).
- class BaseClass:

    Body of base class

  class DerivedClass(BaseClass):

    Body of derived class

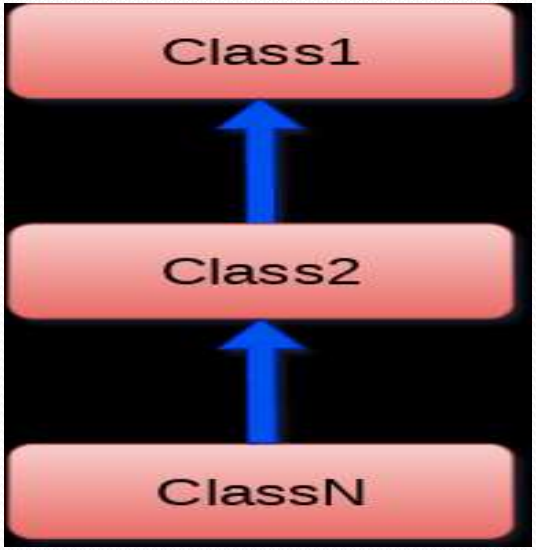


```
class Bird:
    def __init__(self):
        print("Bird is ready")
    def whoisThis(self):
        print("Bird")
    def swim(self):
        print("Swim faster")
Penguin(Bird):
    def __init__(self):
            super().__init__()
        print("Penguin is ready")
    def whoisThis(self):
        print("Penguin")
    def run(self):
        print("Run faster")
peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

Multilevel Inheritance

- Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
.
.
```



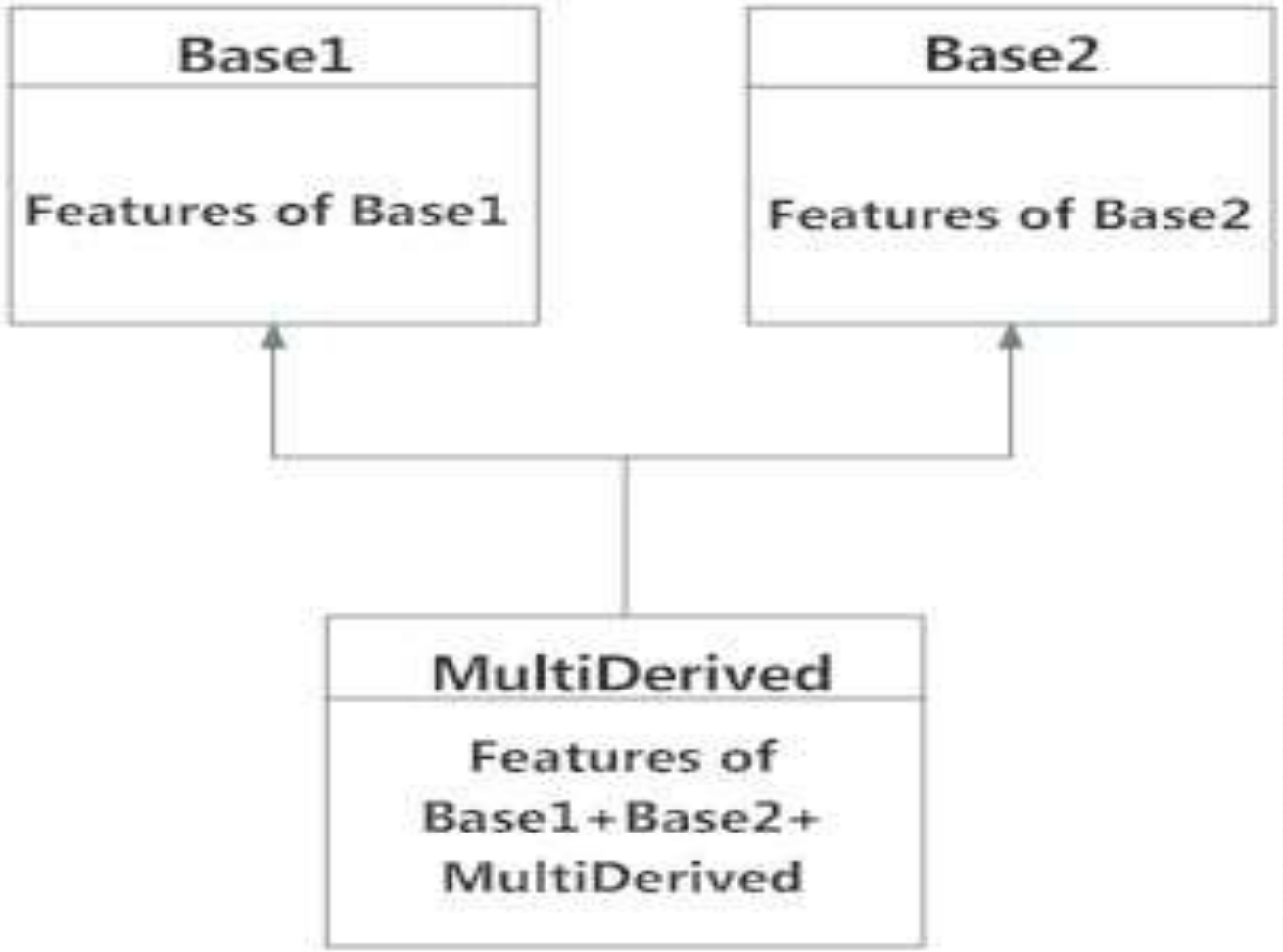
Multiple Inheritance in Python

- a **class** can be derived from more than one base classes in Python. This is called multiple inheritance.
- class Base1:

  pass
- class Base2:

  pass
- class MultiDerived(Base1, Base2):

  pass


Base1
Features of Base1
Base2
Features of Base2
MultiDerived
Features of
Base1+Base2+
MultiDerived

- **The issubclass(sub,sup) method**
- The issubclass(sub, sup) method is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise.

- **The isinstance (obj, class) method**
- The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

Encapsulation

- Using OOP in Python, we can restrict access to methods and variables.
- This prevent data from direct modification which is called encapsulation.
- In Python, we denote private attribute using underscore as prefix i.e single “ _ “ or double “ __“.

- **Hiding data fields**
- To hide data fields you need to define private data fields. In python you can create private data field using two leading underscores. You can also define a private method using two leading underscores.

```
class BankAccount:
    def __init__(self, name, money):
        self.__name = name
        self.__balance = money    # __balance is private now, so it is only accessible inside the class

    def deposit(self, money):
        self.__balance += money
    def withdraw(self, money):
        if self.__balance > money :
            self.__balance -= money
            return money
        else:
            return "Insufficient funds"

    def checkbalance(self):
        return self.__balance
```

Polymorphism

- Polymorphism is an ability (in OOP) to use common interface for multiple form.
- Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

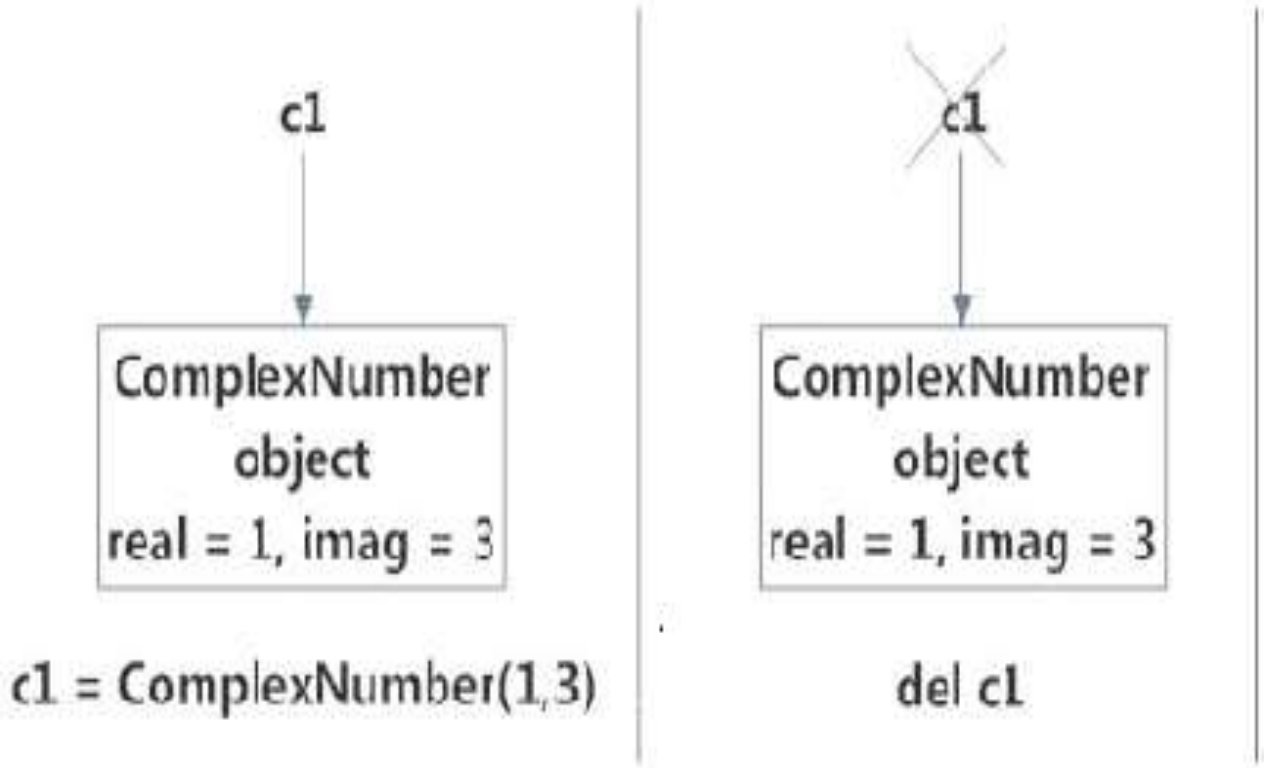
Method Overriding

- We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Constructors in Python

- Class functions that begins with double underscore (__) are called special functions as they have special meaning.
- Of one particular interest is the __init__() function. This special function gets called whenever a new object of that class is instantiated.
- This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

- **Deleting Attributes and Objects**
- Any attribute of an object can be deleted anytime, using the del statement.
- Automatic destruction of unreferenced objects in Python is also called garbage collection.


c1
ComplexNumber
object
real = 1, imag = 3
c1 = ComplexNumber(1,3)
c1
ComplexNumber
object
real = 1, imag = 3
del c1

File handling Methods

- **1) rename():**

  os.rename(existing_file_name, new_file_name)

  Eg.

  import os

  os.rename('mno.txt','pqr.txt')

- **2) remove():**

  Syntax:

  os.remove(file_name)

  Eg.

  import os

  os.remove('mno.txt')

- **3) mkdir()**

  Syntax:

  os.mkdir("file_name")

  eg:

  import os

  os.mkdir("new")

- 4) **getcwd()**

  Syntax:

  os.getcwd()

  Eg

  import os

  print os.getcwd()

- 5) **rmdir()**

  Syntax:

  os.rmdir("directory_name)

  Eg.

  import os

  os.rmdir("new")

Python Regular Expression

- Regular expressions are used to identify whether a pattern exists in a given sequence of characters (string) or not.
- They help in manipulating textual data, which is often a pre-requisite for data science projects that involve text mining.
- You must have come across some application of regular expressions: they are used at the server side to validate the format of email addresses or password during registration, used for parsing text data files to find, replace or delete certain string, etc.

Regular Expressions in Python

- In python regular expression are supported by re module. That means if you want to start using them in your python scripts, you need to import them .

- Syntax:
- Import re

Basic Patterns: Ordinary Characters

- You can easily tackle many basic patterns in Python using the ordinary characters. Ordinary characters are the simplest regular expressions.
- They match themselves exactly and do not have a special meaning in their regular expression syntax.
- Examples are 'A', 'a', 'X', '5'.

- The match function returns a match object if it match otherwise it returns none.
- The r is used before string it says that it is raw string literal. It changes how the string literal is interpreted. Such literals are stored as they appear.

Wild Card Characters: Special Characters

- Special characters are characters which do not match themselves as seen but actually have a special meaning when used in a regular expression.
- The most widely used special characters are:
- .-A period =matches any single character except new line character.
  - re.search(r'Co.k.e', 'Cookie').group()
  - Cookie
  - The group() function returns the string match by re.

- \w(lowercase w) =matches any single letter , digit or underscore.
- re.search(r'Co\wk\we', 'Cookie').group()
- Cookie
- \W(uppercase W)=Matches any character not part of \w (lowercase w).
- re.search(r"C\Wke","C@ke").group()
- C@ke

- \s(lowercase s)=Matches a single whitespace character like: space, newline, tab, return.
- re.search(r'eat\scake','eat cake').group())
- eat cake

- \S(uppercase S)=Matches any character not part of \s (lowercase s).
- re.search(r'cook\Se','cookie').group())
- cookie

- \t(Lower case t)=It matches tab in the string.
- re.search(r'eat\tcake','eat cake').group())
- Eat cake

- \n(lower case n)=it matches new line.
- re.search(r'eat\ncake','eat\ncake').group())
- Eat
- cake

- \d(lowercase d)=Lowercase d. Matches decimal digit 0-9.
- re.search(r'c\d\dkie','cookie').group())
- cookie

- ^(caret)=Matches a pattern at the start of the string.
- re.search(r'^eat','eat cake').group())
- eat

- $(dollar)=Matches a pattern at the end of string.
- re.search(r'cake$','eat cake').group())
- cake

- [a-zA-Z0-9]=Matches any letter from (a to z) or (A to Z) or (0 to 9).
- Characters that are not within a range can be matched by complementing the set.

- If the first character of the set is ^ ,all the characters that are not in the set will be matched.
- Re.search(r'number:[0-6]','number:5'.group())
- Number:5

- Matches any character except 5
- re.search(r'number:[^5]','number:0'.group())
- Number:0

- \A(Uppercase a):Matches only at the start of the string. Works across multiple lines as well.
- re.search(r'\A[A-E]ookie','Cookie').group())
- Cookie

- \b(lowercase b):Matches only the beginning or end of the word.
- re.search(r'\b[A-E]ookie','Cookie').group())
- Cookie

- \(Backslash):if the character following the backslash is recognized escape character then the special meaning of the term is taken.
- For example \n is considered as newline.

Repetitions

- The re module handles repetition using the following special characters.
- + : checks for one or more character to its left.
  - re.search('co+kie','cooookie').group()
  - Cooookie

- * : checks zero or more characters to its left.
  - re.search('co*okie','cookie').group()
  - cookie

- ? : checks exactly zero or one character to its left.
  - re.search('cooki?e','cookie')
  - cookie

- **What if you want to check for exact number of sequence repetition?**
- For example, checking the validity of a phone number in an application. **re** module handles this very gracefully as well using the following regular expressions:
- {x} - Repeat exactly x number of times.

- {x,} - Repeat at least x times or more.
- {x, y} - Repeat at least x times but no more than y times.
- Eg.
- re.search(r'\d{9,10}','0987654321').group()
- 0987654321

Groups and Grouping using Regular Expressions

- Suppose that, when you're validating email addresses and want to check the user name and host separately.
- the group feature of regular expression allows you to pick up parts of the matching text.
- The plain match.group() without any argument is still the whole matched text as usual.

Greedy vs Non-Greedy Matching

- When a special character matches as much of the search sequence (string) as possible, it is said to be a "Greedy Match".
- pattern = "cookie"
- sequence = "Cake and cookie"
- heading = r'<h1>TITLE</h1>'
- re.match(r'<.*>', heading).group()
- Output
- '<h1>TITLE</h1>'

- if you only wanted to match the first <h1> tag, you could have used the greedy qualifier *? that matches as little text as possible.
- Adding ? after the qualifier makes it perform the match in a non-greedy or minimal fashion.
- That is, as few characters as possible will be matched. When you run <.*>, you will only get a match with <h1>.
- heading = r'<h1>TITLE</h1>'
- re.match(r'<.*?>', heading).group()
- output
- '<h1>'

re Python Library

- **search(pattern, string, flags=0)**
- With this function, you scan through the given string/sequence looking for the first location where the regular expression produces a match.
- It returns a corresponding match object if found, else returns None if no position in the string matches the pattern.

- **match(pattern, string, flags=0)**
- Returns a corresponding match object if zero or more characters at the beginning of string match the pattern.
- Else it returns None, if the string does not match the given pattern.

search() versus match()

- The match() function checks for a match only at the beginning of the string (by default).
- The search() function checks for a match anywhere in the string.

- **findall(pattern, string, flags=0)**
- Finds all the possible matches in the entire sequence and returns them as a list of strings.
- Each returned string represents one match.

- **sub(pattern, repl, string, count=0, flags=0)**
- This is the substitute function.
- It returns the string obtained by replacing or substituting the leftmost non-overlapping occurrences of pattern in string by the replacement repl. If the pattern is not found then the string is returned unchanged.

Python Lambda Functions

- Python allows us to not declare the function in the standard manner, i.e., by using the def keyword. Rather, the anonymous functions are declared by using lambda keyword.
- However, Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.
- The anonymous function contains a small piece of code. It simulates inline functions of C and C++, but it is not exactly an inline function.

- The syntax to define an Anonymous function is given below.
- **lambda** args : expression

Python Modules

- A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.
- Modules in Python provides us the flexibility to organize the code in a logical way.
- To use the functionality of one module into another, we must have to import the specific module.

- **Loading the module in our python code**
- We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.
  - The import statement
  - The from-import statement

- **The import statement**
- The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.
- We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.
- The syntax to use the import statement is given below.
  - **import** module1,module2,…… module n

- **The from-import statement**
- Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement. The syntax to use the from-import statement is given below.
- **from** < module-name> **import** <name 1>, <name 2>..,<name n>
- **from** calculation **import** summation

- **Renaming a module**
- Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.
- The syntax to rename a module is given below.
- **import** <module-name> as <specific-name>

- **Using dir() function**
- The dir() function returns a sorted list of names defined in the passed module. This list contains all the sub-modules, variables and functions defined in this module.

Python Exceptions

- An exception can be defined as an abnormal condition in a program resulting in the disruption in the flow of the program.
- Whenever an exception occurs, the program halts the execution, and thus the further code is not executed. Therefore, an exception is the error which python script is unable to tackle with.
- Python provides us with the way to handle the Exception so that the other part of the code can be executed without any disruption. However, if we do not handle the exception, the interpreter doesn't execute all the code that exists after the that.

Common Exceptions

- A list of common exceptions that can be thrown from a normal python program is given below.
  - **ZeroDivisionError:** Occurs when a number is divided by zero.
  - **NameError:** It occurs when a name is not found. It may be local or global.
  - **IndentationError:** If incorrect indentation is given.
  - **IOError:** It occurs when Input Output operation fails.
  - **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

Exception handling in python

- If the python program contains suspicious code that may throw the exception, we must place that code in the try block. The try block must be followed with the except statement which contains a block of code that will be executed if there is some exception in the try block.

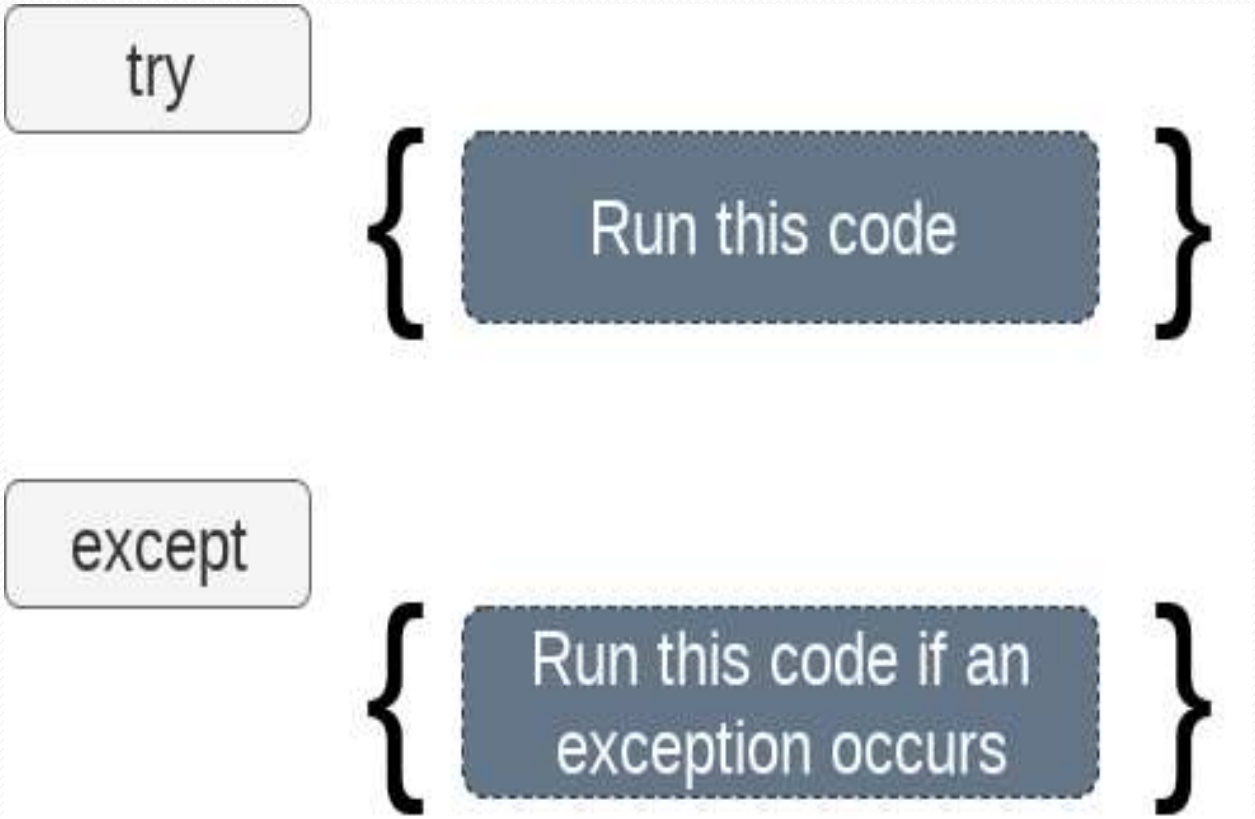
Syntax

```
try:
    #block of code

except Exception1:
    #block of code

except Exception2:
    #block of code

#other code
```


try
{
Run this code
}
except
{
Run this code if an exception occurs
}

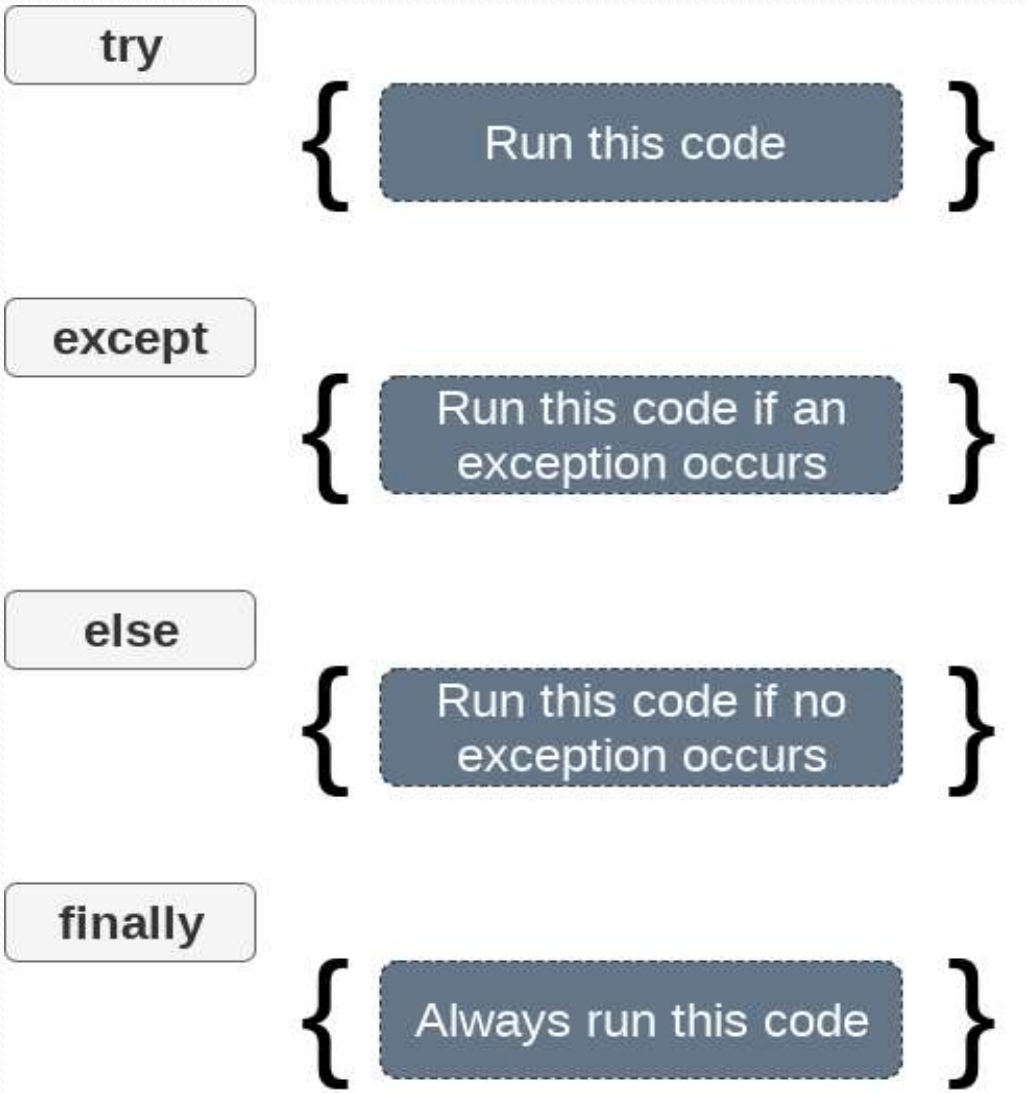
Points to remember

- Python facilitates us to not specify the exception with the except statement.
- We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.
- We can also specify an else block along with the try-except statement which will be executed if no exception is raised in the try block.
- The statements that don't throw the exception should be placed inside the else block.

The finally block

- We can use the finally block with the try block in which, we can pace the important code which must be executed before the try statement throws an exception.
- The syntax to use the finally block is given below.
- Syntax

```
try:
    # block of code
    # this may throw an exception
finally:
    # block of code
    # this will always be executed
```


try
{ Run this code }
except
{ Run this code if an exception occurs }
else
{ Run this code if no exception occurs }
finally
{ Always run this code }

Raising exceptions

- An exception can be raised by using the raise clause in python. The syntax to use the raise statement is given below.

- **syntax**
- raise Exception_class,<value>

- **Points to remember**
- To raise an exception, raise statement is used. The exception class name follows it.
- An exception can be provided with a value that can be given in the parenthesis.
- To access the value "as" keyword is used. "e" is used as a reference variable which stores the value of the exception.

Custom Exception

- The python allows us to create our exceptions that can be raised from the program and caught using the except clause.

```
class ErrorInCode(Exception):
    def __init__(self, data):
        self.data = data
    def __str__(self):
        return repr(self.data)


try:
    raise ErrorInCode(2000)
except ErrorInCode as ae:
    print("Received error:", ae.data)
```

Python Date and time

- In python, the date is not a data type, but we can work with the date objects by importing the module named with datetime, time, and calendar.
- Tick
- In python, the time instants are counted since 12 AM, 1st January 1970. The function time() of the module time returns the total number of ticks spent since 12 AM, 1st January 1970. A tick can be seen as the smallest unit to measure the time.

- Getting formatted time
  - The time can be formatted by using the asctime() function of time module. It returns the formatted time for the time tuple being passed.
  - **import** time;
  - **print**(time.asctime(time.localtime(time.time())))

- Python sleep time
- The sleep() method of time module is used to stop the execution of the script for a given amount of time. The output will be delayed for the number of seconds given as float.

The datetime Module

- The datetime module enables us to create the custom date objects, perform various operations on dates like the comparison, etc.
- To work with dates as date objects, we have to import datetime module into the python source code.

- The calendar module
- Python provides a calendar object that contains various methods to work with the calendars.

```
import calendar;
cal = calendar.month(2018,12)
#printing the calendar of December 2018
print(cal)
```

- Printing the calendar of whole year
- The prcal() method of calendar module is used to print the calendar of the whole year. The year of which the calendar is to be printed must be passed into this method.