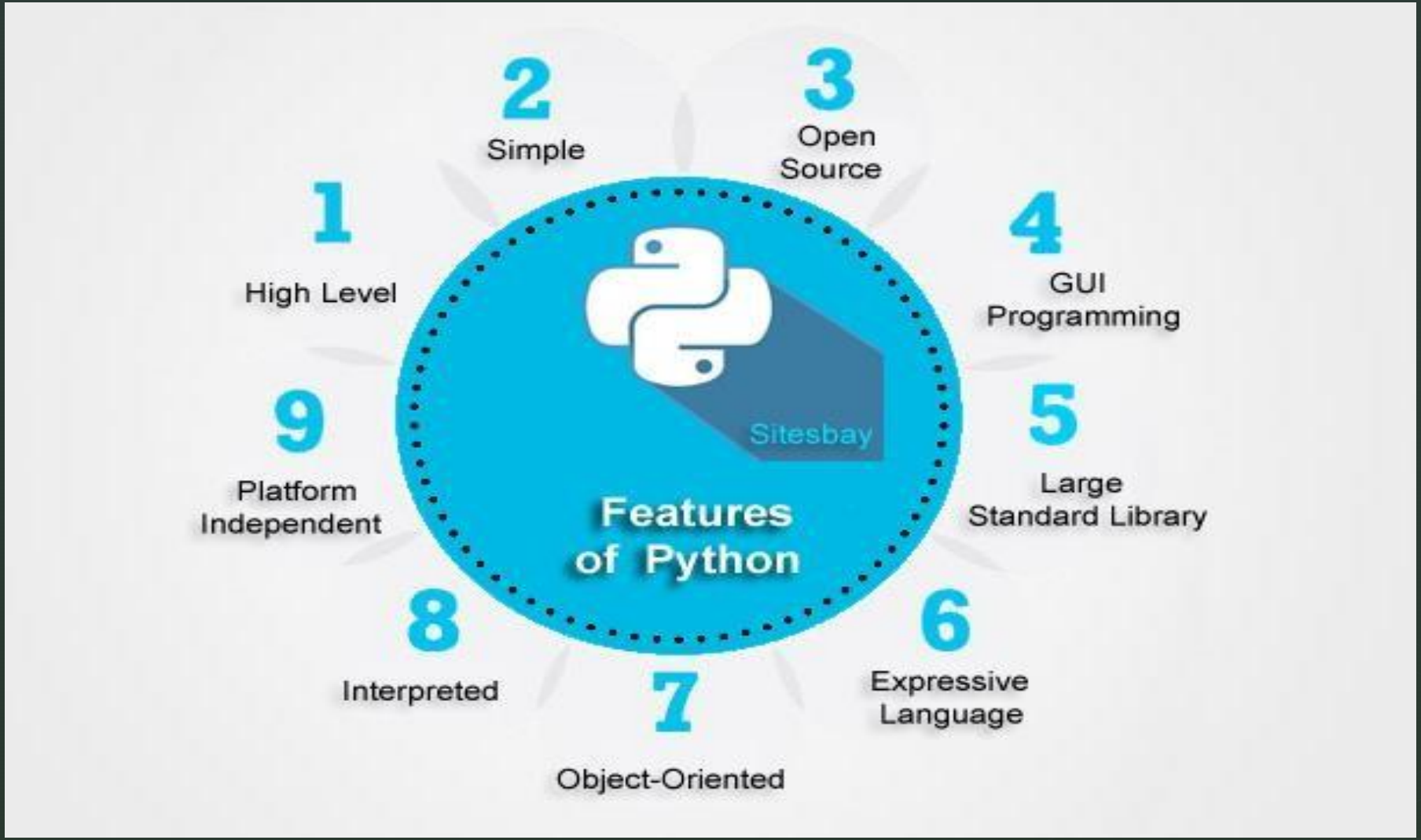
Python Introduction

- Python is a general purpose, dynamic, high level and interpreted programming language.
- It is a object-oriented language.
- Python’s elegant syntax and dynamic typing, together with its interpreted nature, make it and ideal language for scripting and rapid application development in many areas on most platforms.
- It was created by Guido van Rossum and reacted in 1991.

Why Python ?


1
High Level
2
Simple
3
Open Source
4
GUI Programming
5
Large Standard Library
6
Expressive Language
7
Object-Oriented
8
Interpreted
9
Platform Independent
Features of Python
Sitesbay

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc..).
- Python has syntax that allows developers to write programs that code can be executed as soon as it is written. This means that prototype can be very quick.
- Python can be treated in a procedural way, an object-oriented way.

What can Python do ?

- Python can be used on a server to create web application.
- Python can be connect to database system. 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 .

Installing Python ?

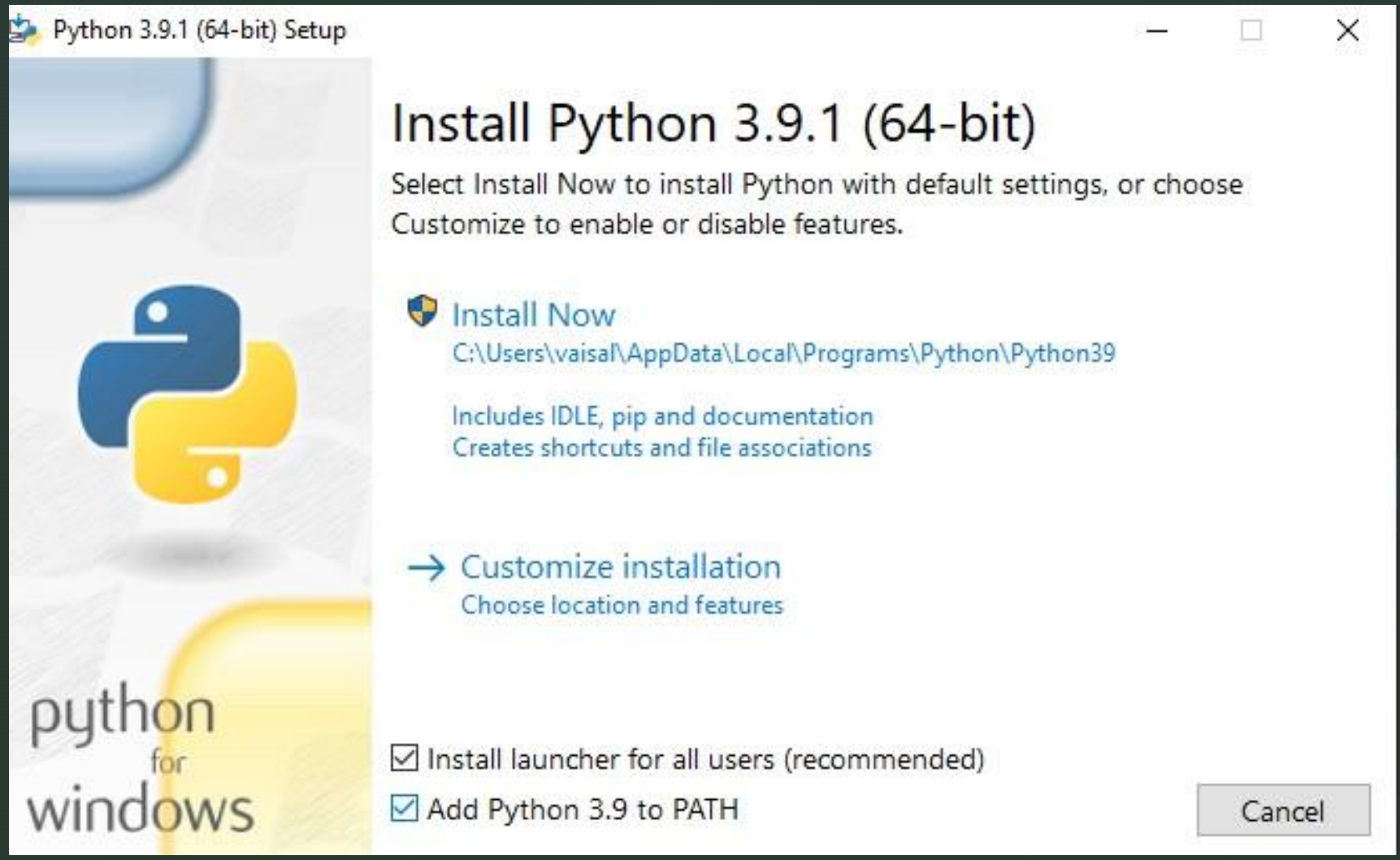
- Go to python official site (https://www.python.org/)
- Download the latest version (3.7.9).

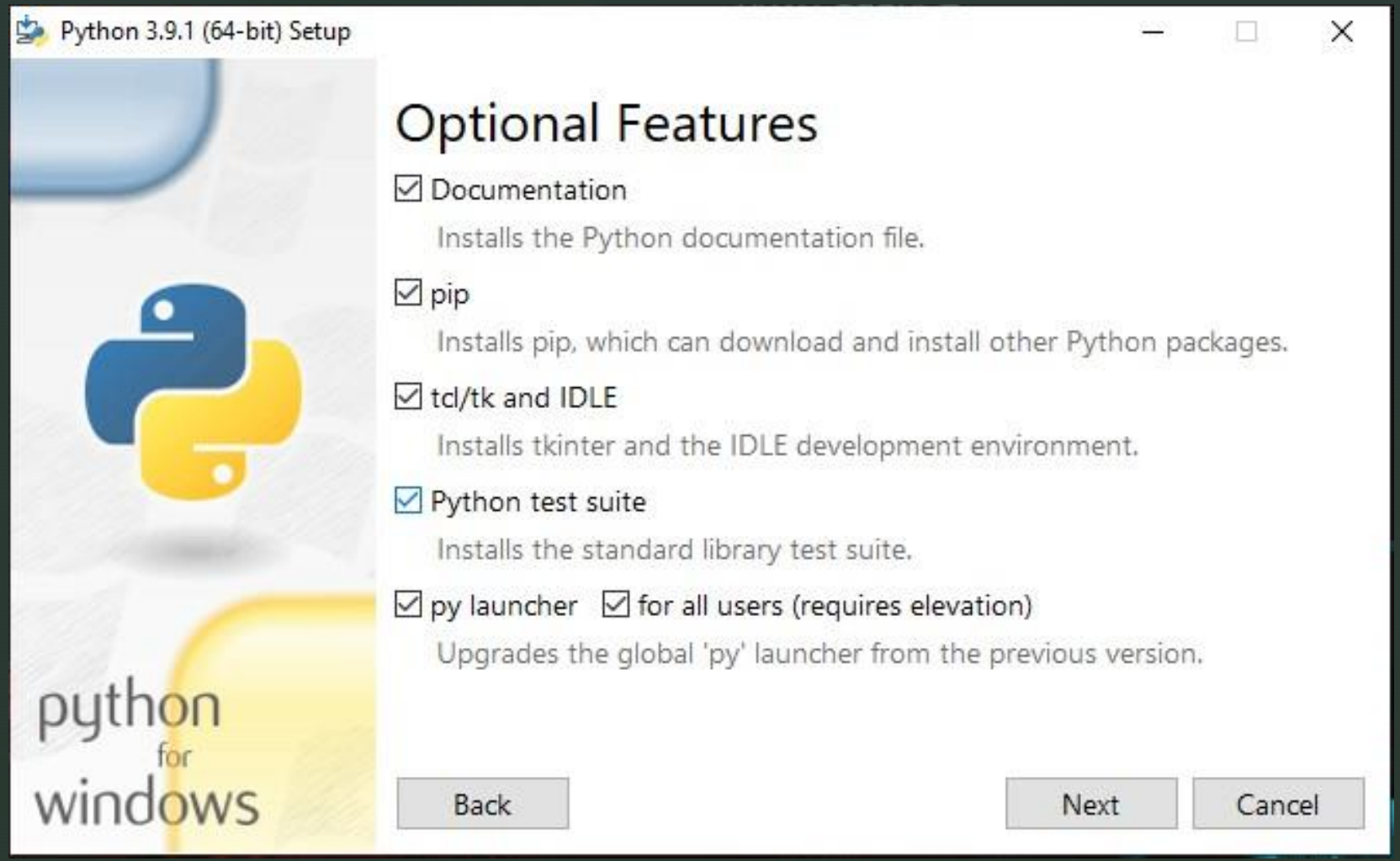
Editor:

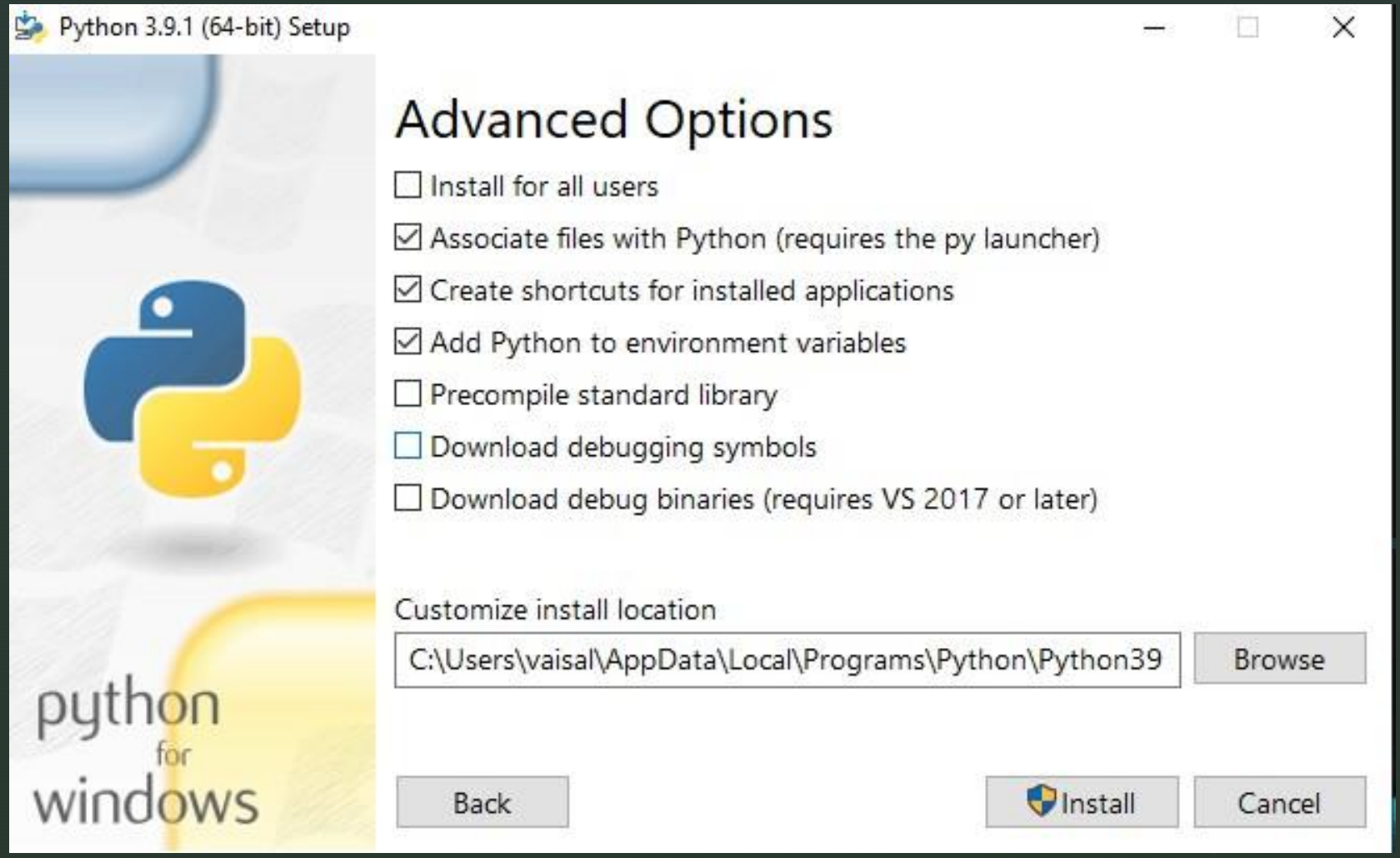
Notepad, Notepad++, Sublime, Atom, Visual Studio Code

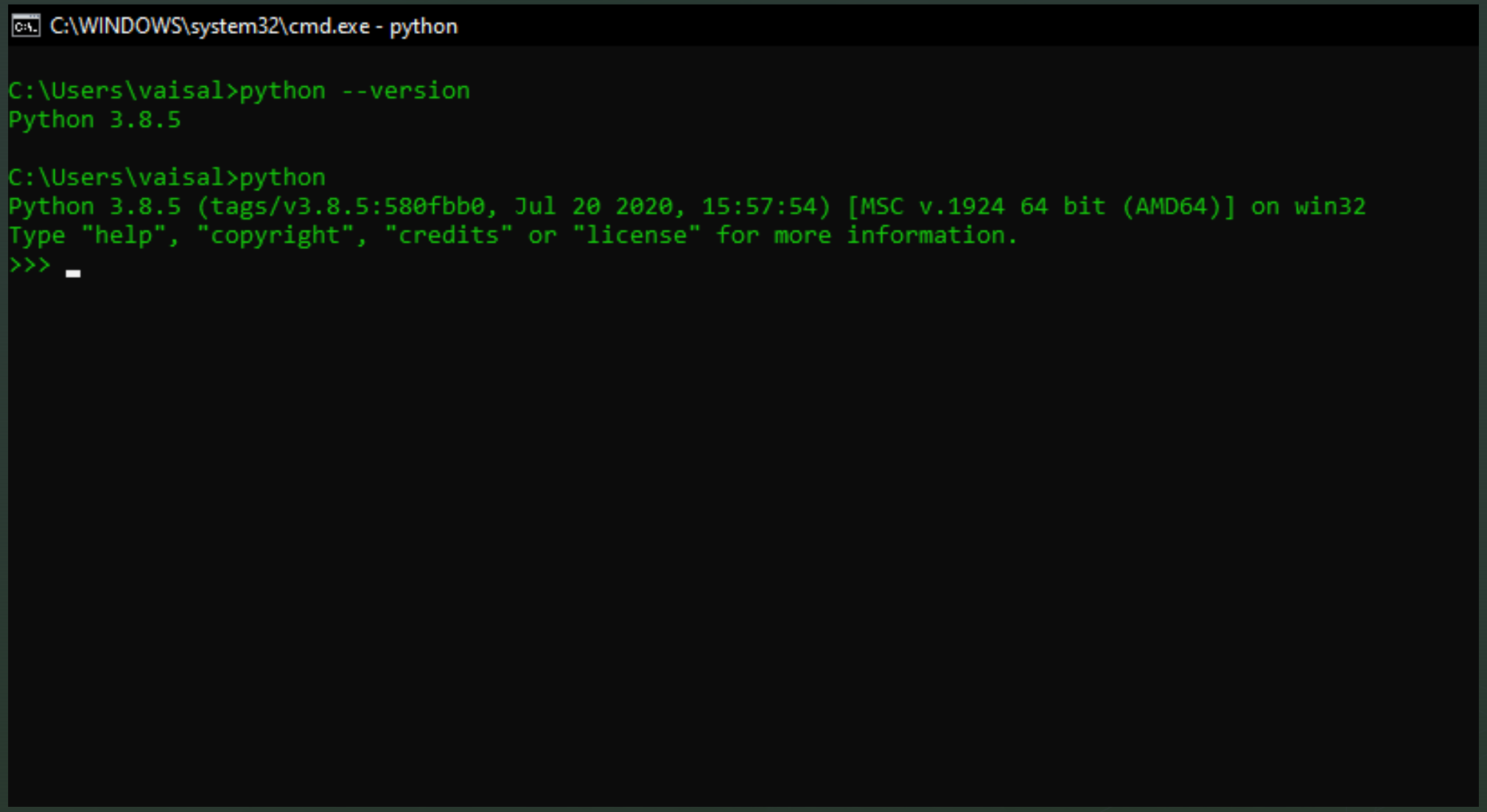
Ide:

Python IDLE , Pycharm


Python 3.9.1 (64-bit) Setup
Install Python 3.9.1 (64-bit)
Select Install Now to install Python with default settings, or choose
Customize to enable or disable features.
Install Now
C:\Users\vaisal\AppData\Local\Programs\Python\Python39
Includes IDLE, pip and documentation
Creates shortcuts and file associations
Customize installation
Choose location and features
python
for
windows
Install launcher for all users (recommended)
Add Python 3.9 to PATH
Cancel


Python 3.9.1 (64-bit) Setup
Optional Features
Documentation
Installs the Python documentation file.
pip
Installs pip, which can download and install other Python packages.
tcl/tk and IDLE
Installs tkinter and the IDLE development environment.
Python test suite
Installs the standard library test suite.
py launcher
for all users (requires elevation)
Upgrades the global 'py' launcher from the previous version.
python
for
windows
Back
Next
Cancel


Python 3.9.1 (64-bit) Setup
Advanced Options
Install for all users
Associate files with Python (requires the py launcher)
Create shortcuts for installed applications
Add Python to environment variables
Precompile standard library
Download debugging symbols
Download debug binaries (requires VS 2017 or later)
Customize install location
C:\Users\vaisal\AppData\Local\Programs\Python\Python39
Browse
python
for
windows
Back
Install
Cancel


C:\WINDOWS\system32\cmd.exe - python
C:\Users\vaisal>python --version
Python 3.8.5
C:\Users\vaisal>python
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

First Python program

1. Open python IDLE
2. File- new file

3. `print("Hello Python!!!!")`

4. Save .py
5. Run

PYTHON COMMENTS

Python has commenting capability for the purpose of in-code documents

1) Comments starts with a #, and Python will render the rest of the line as a comment:

```
#This is a comment

print("Hello Python!!!!")
```

2) Docstring can be on-line or multiline.

```
"""This is a

multiline docstring"""

print("Hello Python!!!!")
```

PYTHON TOKENS

VARIABLE

- Variables are containers for storing data values.
- Unlike other programming languages, Python has no command for declaring a variable.

```
x = 4 # x is of type int
y = “All" # y is now of type str
z = 500.23 # z is now of type float
print(x)
print("Hello " + y)
print(z)
```

MULTIPLE-VARIABLE ASSIGNMENT

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

```
a = b = c = 1
```

- can also assign multiple objects to multiple variables

```
a,b,c = 1,2,"john"
```

Taking input in Python

- input (prompt)

Example:

```
# Taking input from the user

Val = input("Enter your value: ")

Print(val)
```

Tokens

- Tokens are defined as the smallest individual unit in a programs or the very basic components of source code.
- In Python Tokens are classified into
  - Identifiers
  - Keyword
  - Literals
  - Operator

Identifiers

- Identifiers are names that you give to a variable, Class, or Function.
- There are certain rules for naming identifiers similar to the variable declaration rules, such as:

  An Identifier starts with a letter (A to Z or a to z), or an underscore(_) followed by zero or more letters or under-scores and digits (0 to 9)

- Case sensitive.

Keyword

- Keywords are the reserved words in Python.
- We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.
- In Python, keywords are case sensitive.

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

Literals

- Python Literals can be defined as data that is given in a variable or constant
- Python has different types of literals.

1. **String literals**
2. **Numeric literals**
3. **Boolean literals**
4. **Literal Collections**
5. **Special literals**

String literals

A string literal can be created by writing a text(a group of Characters) surrounded by the single(‘ '), double(“ ”), or triple quotes. By using triple quotes we can write multi-line strings or display in the desired way.

Character literal is a single character surrounded by single or double quotes

```
# in single quote
s = ‘hai python’
# in double quote
t= “hello all”
# in multi-line  string
m = ‘’’hai
            for
                all’’’
print(s)
print(t)
print(m)
```

Output

```
hai  python
hello all
hai
        for
            all
```

Numeric literals

- Numeric Literals are immutable. Numeric literals can belong to following 3 different numerical types.

1. **Integer-** Both positive and negative numbers including 0. There should not be any fractional part.
2. **Float-**These are real numbers having both integer and fractional parts.
3. **Complex.-**The numerals will be in the form of **a+bj**, where '**a**' is the real part and '**b**' is the complex part.

```
# integer literal
a = 50
# float
b = 24.5
# complex
c = 7 + 5j
# real part is 0 here.

print(a)
print(b)
print(c)
```

Output

```
50
24.5
(7+5j)
```

PYTHON DATA-TYPES

DATA TYPES

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data.

Python has five standard data types −

- ✓ **Numbers**
- ✓ **String**
- ✓ **List**
- ✓ **Tuple**
- ✓ **Dictionary**

NUMBERS

There are three numeric types in Python:

- ✓ **int**
- ✓ **float**
- ✓ **Complex**

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
print(type(x))       # output <class 'int'>
print(type(y))
print(type(z))
```

STRINGS

- **Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes**

- **Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.**

- **The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.**

```
str = 'Hello World!'

print str          # Prints complete string
print str[0]       # Prints first character of the string
print str[2:5]     # Prints characters starting from 3rd to 5th
print str[2:]      # Prints string starting from 3rd character
print str * 2      # Prints string two times
print str + "TEST" # Prints concatenated string
```

PYTHON COLLECTION (ARRAYS)

- Arrays are used to store multiple values in one single variable.
- An array is a special variable, which can hold more than one value at a time.

There are four collection data types in the Python programming language:

a) **List**

b) **Tuple**

c) **SET**

d) **Dictionary**

LIST

- **List is a collection which is ordered and changeable.**
- **Allows duplicate members.**
- **In Python lists are written with square brackets.**

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

LIST METHODS

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

TUPLES

- **Tuple is a collection which is ordered and unchangeable. Allows duplicate members.**
- **Allows duplicate members.**
- **In Python tuples are written with round brackets.**

Example :

```
thistuple = ("apple", "banana", "cherry“)
print(thistuple)
```

TUPLE METHODS

Method	*Description*
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

SET

- **Set is a collection which is unordered and unindexed.**
- **No duplicate members.**
- **In Python set are written with curly brackets.**

Example :

```
thisset = {"apple", "banana", "cherry“}

print(thisset)
```

SET METHODS

Method	*Description*
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
Difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
Intersection_update()	Removes the items in this set that are not present in other, specified set(s)
Isdisjoint()	Returns whether two sets have a intersection or not

Method	Description
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

DICTIONARY

- **Dictionary is a collection which is unordered,changeable and indexed.**
- **No duplicate members.**
- **In Python dictionary are written with curly brackets and they keys and values**

Example :

```
thisdict =          {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

DICTIONARY METHODS

Method	*Description*
formkeys()	Returns a dictionary with the specified keys and values
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
get()	Returns the value of the specified key
items()	Returns a list containing the a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

OPERATORS

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

1) Arithmetic operators
2) Assignment operators
3) Comparison operators
4) Logical operators
5) Identity operators
6) Membership operators
7) Bitwise operators

ARITHMETIC OPERATORS

Arithmetic operators are used with numeric values to perform common mathematical operations:

```
a=int(input("Enter the first number"))

        b=int(input("Enter the second number"))

        c = a + b

        print("Sum of {0} and {1} is {2}".format(a,b,c))
```

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

COMPARISON OPERATORS

Comparison operators are used to compare two values:

```
a=5

b=3

print( a == b )

# returns False because 5 is not equal to 3
```

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!= b) is true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

ASSIGNMENT OPERATORS

Assignment operators are used to assign values to variables:

```
a=5

print( a )
```

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

LOGICAL OPERATORS

Logical operators are used to combine conditional statements:

```
a=5

print( a > 3 and a < 10)


# returns True because 5 is greater than 3 AND 5 is less than 10
```

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is False.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is True.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is True.

MEMBERSHIP OPERATORS

Membership operators are used to test if a sequence is presented in an object:

```
a = ["apple", "banana"]

print( “banana” in a)

# returns True because a sequence with the value "banana" is in the list
```

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

IDENTITY OPERATORS

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

```
x = ["apple", "banana"]

y = ["apple", "banana"]

z = x

print (x is z)

# returns True because z is the same object as x

print(x is y)

# returns False because x is not the same object as y, even if they have thew same content

print(x == y)

# to demonstrate the difference betweeen "is" and "==": this comparison returns True because x is equal to y
```

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

PYTHON CONDITIONS
&
IF-STATEMENTS

Python supports the usual logical conditions from mathematics:

Equals: a == b

Not Equals: a != b

Less than: a < b

Less than or equal to: a <= b

Greater than: a > b

Greater than or equal to: a >= b

- **These conditions can be used in several ways, most commonly in "if statements" and loops.**
- **An "if statement" is written by using the if keyword.**

Example :

```
a = 33
b = 200
if b > a:
        print("b is greater than a")
```

If , elif and else

```
a = 200
b = 33
if b > a:
        print("b is greater than a")
elif a == b:
        print("a and b are equal")
else:
        print("a is greater than b")
```

- AND

- The and keyword is a logical operator ,
- and is used to combine conditional statements:

```
a = 200

b = 33

c = 500

if a > b and c > a:

print("Both conditions are True")
```

Or

Or keyword is a logical operator, and is used to combine conditional statements:

```
a = 200
b = 33
c = 500
if a > b or a > c:
        print("At least one of the
conditions are True")
```

PYTHON LOOPS

Python has two primitive loop commands:

1) **while loops**

2) **for loops**

FOR LOOP

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

```
i = 1

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

for x in fruits:

        print(x)
```

WHILE LOOP

With the while loop we can execute a set of statements as long as a condition is true.

```
i = 1

while i < 6:

        print(i)

        i += 1
```

BREAK STATEMENTS

With the break statement we can stop the loop before it has looped through all the items:

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

  print(x)

  if x == "banana":

    break
```

CONTINUE STATEMENT

With the continue statement we can stop the current iteration of the loop, and continue with the next:

```
fruits = ["apple", "banana", "cherry"]

for x in fruits:

    if x == "banana":

        continue

    print(x)
```

PYTHON FUCTION

Fuctions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

Creating a Function

In Python a function is defined using the def keyword:

```
def  my_function():
  print("Hello from a function")
```

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return
```

Calling a Function

Defining a function gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is an example to call the **printme()** function −

```
#!/usr/bin/python3

# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return

# Now you can call printme function
printme("This is first call to the user defined function!")
printme("Again second call to the same function")
```

Try it

When the above code is executed, it produces the following result −

```
This is first call to the user defined function!
Again second call to the same function
```

Global VARIABLE

```
def compute():
    global x
    print ("Value of x in compute function is", x)
    x += 5
    return None
def dispvalue():
    global x
    print ("Value of x in dispvalue function is", x)
    x-=2
    return None
x=0
compute()
dispvalue()
compute()
```

LOCAL VARIABLES

```
def compute(x):
    x += 5
    print ("Value of x in function is", x)
    return None
x=10
compute(x)
print ("Value of x is still", x)
```

Output:

Value of x in function is 15
Value of x is still 10

LAMBDA FUNCTION

- A lambda function is an anonymous and one-use-only function that can have any number of parameters and that does some computation.
- EG:

```
g = lambda x: x*2

g(3)

6
```

FILTER FUNCTION

The filter() method returns a sequence consisting of those elements for which the included function returns true, those that satisfy the criteria given in the specified function.

```
def evenval(x):
    return x % 2 ==0
evens=filter(evenval, range(1,11))
print(list(evens))
```

Output:

[2,4,6,8,10]

MAP FUNCTION

- The map method calls the included function for each of the elements in the sequence and returns a list of the returned values.

```
def square(x):
    return x*x
sqr=list(map(square, range(1, 11)))
print(sqr)
```

Output:

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

Modules

- A module allows you to logically organize your Python code
- A module is a Python object with arbitrarily named attributes that you can bind and reference
- A module can also include runnable code.
- use any Python source file as a module by executing an import statement in some other Python source file.

IMPORTING MODULES

- use the import statement
- can use import in several forms.

DIFFERENT WAYS TO IMPORT A MODULE

- Import module
- From module import function
- From module import *
- Import module as name

Importing modules

- Consider a calendar module that displays a calendar of a specified month and year

```
import calendar

calendar.prcal(2016)
```

From calendar import prcal

Prcal(2016)

From calendar import *

Prcal(2016)

Import calendar as cal

Cal.prcal(2016)

SOME IN-BUILT MODULES

- Calendar
- Math
- Time
- Random
- sys

MATH MODULE

- Math module is which containing mathematical expression which are already saved in it
- There are a lot of expressions are saved in this module
- The dir() built-in function returns a sorted list of strings containing the names defined by a module.
- The list contains the names of all the modules, variables and functions that are defined in a module

math.pi—Returns the value of pi, 3.14159265358979 31.

math.e—Returns the value of e, 2.718281828459045 1.

ceil(x)—Displays the next larger whole number.

floor(x)—Displays the next smaller whole number.

```
import math
content = dir(math)
print (content);
```

When the above code is run output shows as

['_doc_', '_file_', ' name_', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']

PYTHON OOPs CONCEPT

Python OOPs Concepts

- Python is an object-oriented programming language. It allows us to develop applications using Object Oriented approach. In Python, we can easily create and use classes and objects.

Python OOPs Concepts

Major principles of object-oriented programming system are given below

- Object
- Class
- Method
- Inheritance
- Polymorphism
- Data Abstraction
- Encapsulation

CLASS AND OBJECTS

What is class ?

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.
- To Create a class , use the keyword **class :**

Example :

To create a class name ‘Myclass’ , with a property named x :

```
class Myclass:

    x = 5
```

What is a Object ?

- Object is a specific instance or a member of a class
- Properties is same but values are different

Syntax:

```
Objectname = classname()
```

- Example
  - Create an object named p1, and print the value of x :

```
p1 = Myclass()
print(p1.x)
```

Child Class

- Class inside another class is called child class

- Example :

```
class Human:

class teacher(human):
```

What is ‘self’ !

- The self parameter is a reference to the class itself , and is used to access variables that belongs to the class

  - It does not have to be named self , you can call it whatever you like , But it has to be the first parameter of any class

Syntax : def speak (self):

CONSTRUCTOR

Constructor / __init__() Method

- Constructor is used for access the class directly

```
class rect:
    def___init__(self):
        self.l = 8
        self.b = 5
r=rect()
```

Default constructor

```
class rect:
    def__init__(self):
        self.l = 8
        self.b = 5
    def rectarea(self):
        return self.l * self.b
r=rect()
print ("Area of rectangle is ", r.rectarea())
```

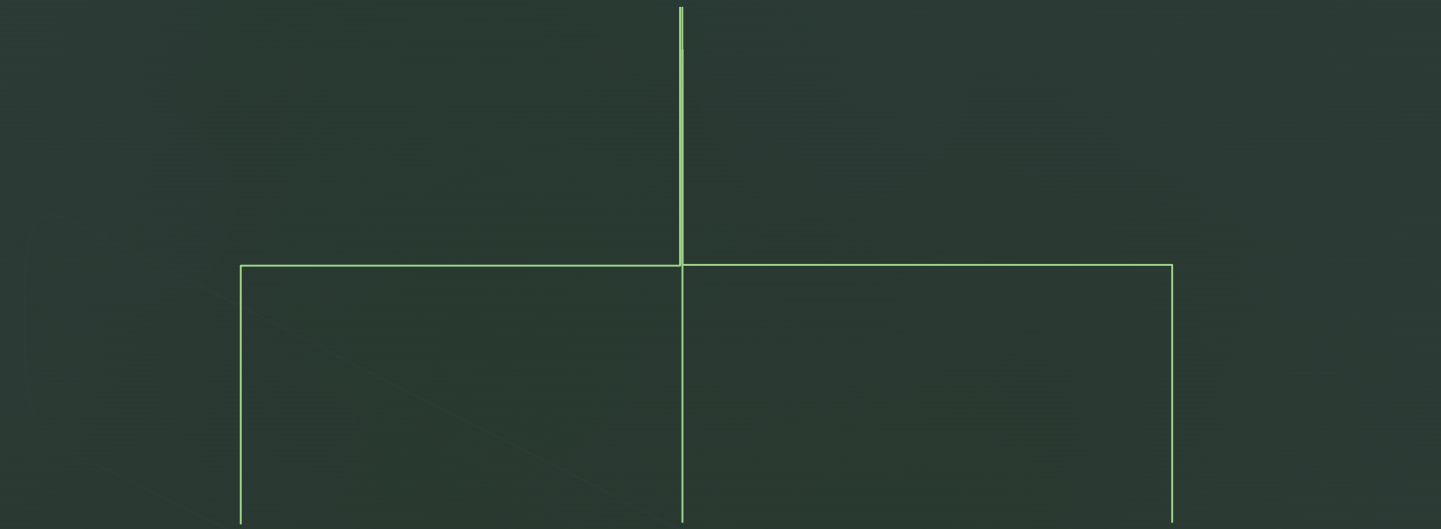
Parameterized constructor

```
class rect:
    def__init__(self, x,y):
        self.l = x
        self.b = y
    def rectarea(self):
        return self.l * self.b
r=rect(5,8)
print ("Area of rectangle is ", r.rectarea())
```

Class Inheritance

- Super class
  - super class is a class which is also called as the parent class or base class
  - super class is the class which has some sub classes
  - There are multiple number of super class may or may not be available in a single program
- Sub class
  - Sub class is a class which is also called as the child class or derived class
  - Sub class is the classes are derived from a base class
  - Many sub class are derived from a single super class

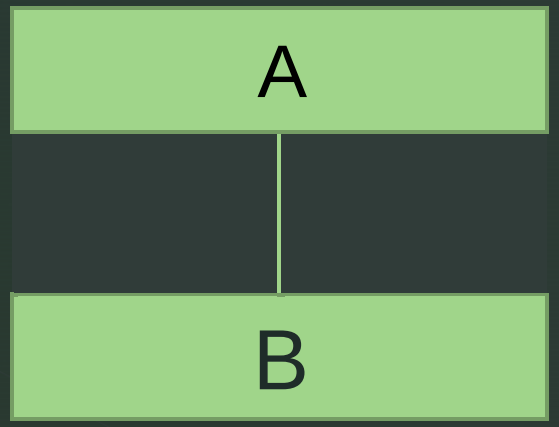
INHERITANCE



Single inheritance

- This is the simplest type of inheritance, where one class is derived from another single class

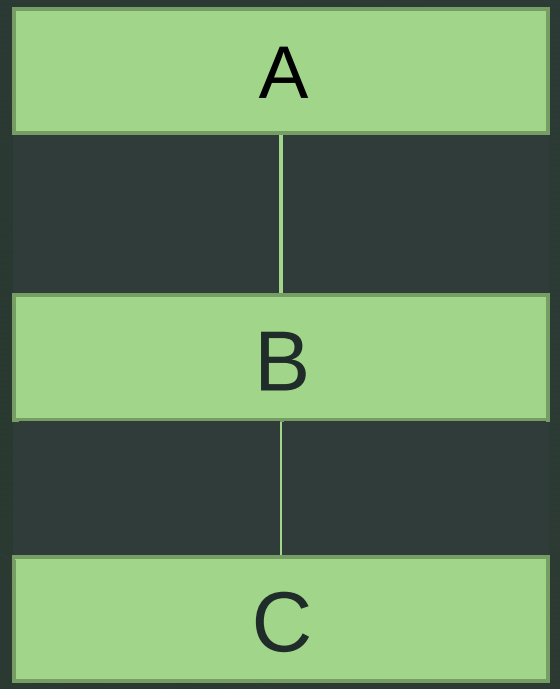
```
class A:
    def feature1(self):
        print("Feature 1 working ...")
    def feature2(self):
        print("Feature 2 working ...")

class B(A):
    def feature3(self):
        print("Feature 3 working ...")
    def feature4(self):
        print("Feature 4 working ...")

B1=B()
b1.feature1()
b1.feature2()
b1.feature3()
b1.feature4()
```



Multi-level inheritance

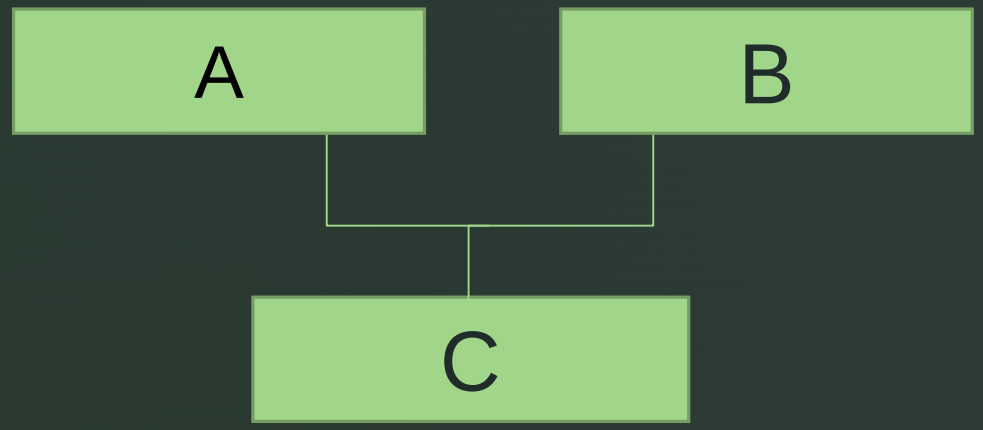
- When we have a child and grandchild relationship.

```
class A:
    def feature1(self):
        print("Feature 1 working ...")
    def feature2(self):
        print("Feature 2 working ...")
class B(A):
    def feature3(self):
        print("Feature 3 working ...")
    def feature4(self):
        print("Feature 4 working ...")
class C(B):
    def feature5(self):
        print("Feature 5 is working.... ")
c1=C()
c1.feature5()
c1.feature3()
c1.feature4()
c1.feature1()
c1.feature2()
```



Multiple inheritance

- When a child class inherits from multiple parent classes, it is called multiple inheritance.




```
class A:
    def feature1(self):
        print("Feature 1 working ...")
    def feature2(self):
        print("Feature 2 working ...")


class B:
    def feature3(self):
        print("Feature 3 working ...")
    def feature4(self):
        print("Feature 4 working ...")
class C(A,B):
    def  feature5(self):
        print("Feature5 is working.... ")
c1=C()
c1.feature5()
c1.feature3()
c1.feature4()
c1.feature1()
c1.feature2()
```

Access control specifiers

- Public member—Accessed from inside as well as outside of the class.
- Private member—Cannot be accessed from outside the body of the class. A private member is preceded by a double underscore(__).

```
class rect:
    def__init__(self, x,y):
        self.__l = x
        self.__b = y
    def rectarea(self):
        return self.__l * self.__b
r=rect(5,8)
print ("Area of rectangle is ", r.rectarea())
print ("Area of rectangle is ", r._rect__l* r._rect__b)
```

Polymorphism in Python

- The word polymorphism means having many forms. In programming, polymorphism means same function name (but different signatures) being uses for different types.

```
# A simple Python function to demonstrate
# Polymorphism

def add(x, y, z = 0):
    return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

FILES AND EXCEPTION HANDLING

- Opening and closing file
- Reading and writing the file
- Close(),write(),read() methods
- File directory
- Serialization
- Exception handling

Opening and closing file

- Creating a file anywhere in the computer
- Once a file is created u must close it
- To open a file first take an object
- And set the path where the file is present

OPEN ( )

- Before you can read or write a file, you have to open it using Python's built-in *open()*function
- This function creates a **file** object, which would be utilized to call other support methods associated with it.
- syntax

- file object = open(file_name [, access_mode][, buffering])

- **file_name: -**The file_name argument is a string value that contains the name of the file that you want to access.
- **access_mode: -**The access_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. A complete list of possible values is given below in the table. This is optional parameter and the default file access mode is read (r).
- **buffering:-** If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size. If negative, the buffer size is the system default (default behavior).

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

EXAMPLE

```
fo = open("foo.txt", "wb")

print ("Name of the file: ", fo.name)

 print ("Closed or not : ", fo.closed)

 print ("Opening mode : ", fo.mode)
```

CLOSE ( )

- The close() method of a *file* object flushes any unwritten information
- closes the file object, after which no more writing can be done.
- Python automatically closes a file when the reference object of a file is reassigned to another file
- It is a good practice to use the close() method to close a file

EXAMPLE

- fo = open("foo.txt", "wb")
- print "Name of the file: ", fo.name
- fo.close()

Reading and writing file

- The *file* object provides a set of access methods to make our lives easier

Write method

- The *write()* method writes any string to an open file
- It is important to note that Python strings can have binary data and not just text.
- The write() method does not add a newline character ('\n') to the end of the string:
- Syntax
- fileObject.write(string);

EXAMPLE

```
# Open a file

fo = open("foo.txt", "wb")

fo.write( "Python is a great language.\nYeah its great!!\n");


# Close opend file
 fo.close()
```

Read method

- The *read()* method reads a string from an open file.
- It is important to note that Python strings can have binary data, apart from text data.
- Syntax
- fileObject.read([count]);
- passed parameter is the number of bytes to be read from the opened file

File directory related methods

- file.close() Close the file. A closed file cannot be read or written any more
- file.flush() Flush the internal buffer, like stdio's flush. This may be a no-op on some file-like objects.
- file.fileno() Returns the internal file descriptor used by the OS library when working with this file.
- file.isatty() Returns true if the file is connected to the console or keyboard.
- file.next() Returns the next line from the file each time it is being called.

- file.read([size]) Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes).
- file.readline([size]) Reads one entire line from the file. A trailing newline character is kept in the string.
- file.readlines([sizehint])Reads until EOF using readline() and return a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.

- file.seek(offset[,whence]) Sets the file's current position.
- file.tell() Returns the file's current position
- file.truncate([size]) Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.
- file.write(str) Writes a string to the file. There is no return value
- file.writelines(sequence) Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings.

```
f = open("aboutbook.txt", "r")
print ("Name of the file:", f.name)
print ("Closed?", f.closed)
print ("Opening mode:", f.mode)
print ("File number descriptor is:", f.fileno())
f.close()
```

```
matter = '''Python is a great language
Easy to understand and learn
Supports Object Oriented Programming
Also used in web development '''
f = open('aboutbook.txt', 'w')
f.write(matter)
f.close()
f = open('aboutbook.txt')
while True:
    line = f.readline()
    if len(line) == 0:
        break
print (line,)
f.close()
```

COPYING A FILE

```
f = open('aboutbook.txt', 'r')

lines = f.read()

f.close()

g = open('copyaboutbook.txt', 'w' )

g.write(lines)

g.close()

print('The copy of the file is made')

g = open('copyaboutbook.txt', 'r' )

lines = g.read()

print (lines)

g.close()
```

GET ANY LINE FROM THE FILE

```
import linecache

line=linecache.getline('aboutbook.txt', 3)

print ('The content of the third line is:', line)
```

EXCEPTION HANDLING

- Exceptions occur when certain situations arise in a program
- dividing a value by 0, accessing a list element out of its index range, or reading a file that does not exist are situations that cause exceptions
- Misspelling in a statement or a missing parenthesis or quotation mark are all syntax errors.
- two kinds of try blocks:

1) Try

2) Except

What is Try….. Except…..

- The ‘try’ block lets you test a block of code for errors.
- The ‘except’ block lets you handle the error.
- The ‘finally’ block lets you execute code, regardless of the result of the try- and except blocks.

Exception Handling..

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

```
try:
  print(x)
except:
  print("An exception occurred")
```

#The try block will generate an error, because x is not defined:

Exception Handling..(Else)

You can use the ‘else’ keyword to define a block of code to be execute if no errors were

raised

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

#The try block does not raise any errors, so the else block is executed:

Exception Handling..(finally..)

The finally block, if specified, will be executed regardless if the try block raises an error or not.

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

#The finally block gets executed no matter if the try block raises any errors or not:

Exception & Description

- AssertionError : Raised when Assertion fails.
- AttributeError :Raised when an attribute is not found in an object.
- EOFError: Raised when you try to read beyond the end of a file.
- FloatingPointError: Raised when a floating-point operation fails.
- IOError: Raised when an I/O operation fails.
- IndexError: Raised when you use an index value that is out of range.
- KeyError :Raised when a mapping key is not found.
- OSError: Raised when an OS system call fails.
- OverflowError :Raised when a value is too large to be represented.
- TypeError: Raised when an argument of inappropriate type is supplied.
- ValueError :Raised when an inappropriate argument value is supplied.
- ZeroDivisionError: Raised when a number is divided by 0 or when the second argument in a modulo operation is zero.