Python Language — HLL

- Python is simple & easy
- Free & Open source
- High Level Language
- Developed by Guido van Rossum
- Portable

- DS, ML, AI
- Web Development (Django)
- Game ....

Code

```
print ("Hello, Shajib")
print ("I'm studying at B.Sc in Engg. at", "MBSTU");
print (50+53)
```

Variables : A variable is a name give to a memory location in a program.

```
name = "Shajib"
age = 23
price = 560.99
print ("Name:", name)
print ("Age :", age)
print ("Price:", price)
```

```
str1 = "Shajib"
str2 = 'Shajib'
str3 = '''Shajib'''
```

Data types → Integer, String, Float, Boolean, None.

```
Print (type(name))
print (type(age))
print (type (price))
isAdult = True
x = None
print (type(isAdult))
print (type (x))
```

Keywords : Keywords are reserved words in Python.

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

Operators : An operator is a symbol that performs a certain operation between operands.

Arithematic operator → +, −, *, /, %, **

Relational / Comparison Operators → ==, !=, >, <, >=, <=

Assignment Operator → =, +=, −=, *=, /=, %=, **=

Logical Operator → not, or, and

Code

```
a = 50
b = 3
res = a ** b
print(res)
print(not (a==b))
```

Comments

```
# single line comment
"""
Multi Line
Comment
"""
```

Type Conversion : Automatically done by python Interpreter.

Type Casting : Manually chang the type.

Type Conversion

```
a = 2
b = 4.25
sum = a+b
print(sum) ⟹ 6.25
```

Type Casting

```
str = "20"
a = int(str)
b = 50.89
sum = a+b
print(sum) ⟹ 70.89
```

Input in Python : input() statement is used to accept values from user.

```
input()  # result for input() is always a string
int(input())  # input integer
float(input())  # input float
```

Code

```
name = input("Enter your name:")
age = int(input("Enter your age:"))
price = float(input("Enter price :"))
```

String : String is a data type that stores a sequence of characters.

```
str = "This is a String. I'm Shajib. \n I'm studying CSE at MBSTU."
print(str)
```

Concatenation

```
str1 = "Md."
str2 = "Shajib"
print(st1 + str2)
print(str2[0])
```

Length

~~# length~~

```
l = len(str2)
print(l)
```

Not Allowed

```
str2[2] = 'o'
```

We just Access index value

We can't manipulate/change value.

Slicing : Accessing parts of a string

```
str = "ApnaCollege"
print(str[1:4])   → pna
print(str[:4])    → Apna
print(str[1:])    → pnaCollege
print(str[:])     → ApnaCollege
```

Slicing Negative idx

```
str = "Apple"
      -5-4-3-2-1
print(str[-3:-1])  → pl
```

String Function

```
str = " I'm studying Python from ApnaCollege."
print (str.endswith("ege."))   → True
print (str.endswith("app"))    → False

str = "i read book"
print (str.capitalize())   → I read book

str = "Sojib Sojib"
print (str.replace("o", "ha"))   → Shajib Shajib

str = "This is a string"
idx = str.find("string")
print(idx)   → 10
idx = str.find("Shajib")
print(idx)   → -1

str = "The quick brown fox jumps over The lazy dog"
print (str.count("The"))   → 2
```

Conditional Statement

```
age = 21
if (age >=18):
    print("Adult, Can vote, Can drive")
elif (age<18 and age>0):
    print("Not Adult, can't vote, Can't drive")
else:
    print("Enter a valid age")
```

List : A build in data type that stores set of values.

Note — [String are Immutable (Can't change)
List are Mutable (Can change)

```
marks = [94.5, 50.75, 66.5, 45.2]
print(marks)
print(marks[2])
print(len(marks))
student = ["Md. Shajib", 23, 97.50, "Gazipur"]
student[0] = "Shajib"  # Mutable → change the idx value
print(student[1:])
```

List Method

```
list = [2,1,3]
list.append(4)
print(list) ⟶ [2,1,3,4]
list.sort() ⟶ [1,2,3,4]
list.sort(reverse=True) → [4,3,2,1]
list.reverse() ⟶ [1,2,3,4]

list.insert(2,500)
list.insert(2,500)
print(list) ⟶ [1,2,500,500,3,4]
list.remove(500) ⟶ [1,2,500,3,4]   Remove 1st occurrence
list.pop(2) ⟶ [1,2,3,4]
```

Remove 2nd idx

Tuples: A build-in data types that lets us create immutable sequence of values.

```
tup = (2, 1, 3, 5)
print (tup)
print (tup [2])
print (type(tup))
```

```
tup = (1,)  # always take comma when elements number is 1.
```

```
tup = (1)   ✗
```

treat as a integer

```
      0  1  2  3  4
tup = (2, 3, 1, 5, 4)
print (tup [1:4])
```

Method in tuple

```
idx = tup.index(1)   # Return the index of first occurrence
print (idx)  ⟶ 2

tot = tup.count(5)   # Return total number of occurrence
print(tot)  ⟶ 1
```

Dictionary: Dictionaries are used to store data values in key:value pairs.

↳ They are: unordered, mutable (changable) & don't allow duplicate keys.

```
info = {
    "key" : "value",
    "name" : "Shajib",
    "age" : 23,
    "is_adult" : True,
    "subjects" : ["Python", "DS", "ML"],
    "topics" : ("dict", "set"),
    53 : "This is my ID",
    3.85 : "This is my Arg cgpa"
}
print (info)
info
print (type (info))
print (info["name"])
info["name"] = "Md. Shajib",
info
info["addNewkey"] = "We can easyly add new key"
info
null_dict = {}
null_dict["name"] = "Shajib"
print (null_dict)
```

Nested Dictionary

```
student = {
    "name" : "Md Shajib",
    "subjects" : {
        "phy" : 95,
        "chem" : 93,
        "math" : 96
    },
    "batch" : 18
}

student
print ( student ["subjects"] ["math"] )
```

Dictionary Methods

```
print (student.key())      #give all keys
print ( len (student))
print ( student . values ())   # give all values
print (student. items ())      # return all pairs as tuple
print (student. get ("name"))
print (student. get ("name2"))  # if key doesn't exist, return None
print (student ["name2"])  # if key doesn't exist, give error.

student.update ({"city":"Dhaka", "age": 23 , "name": "Shajib53"})
```

↳ insert specified item to the dictionary.

Set: Set is the collection of unordered items.

↳ Set is mutable (changable), but within the set the elements are immutable.

↳ Each element in the set must be unique & immutable.

↳ Set ignores duplicate values.

Code

```
collection = {1, 2, 2, 2, "hello", "world", "world", 4}
print (collection)
print (type (collection))
print (len (collection))

collection = { }  # empty dictionary
print (type (collection))
collection = set ()  # For empty set
```

set Method

```
collection.add(1)
collection.add(2)
collection.add(2)
collection.add("shajib")
collection.add((1,2,3))
print(collection)

collection.remove(2)
collection.remove(5)   <-- Gives error, because 5 is not element of the set.
collection.add([10,20,30])  # Gives error. Set doesn't allow add a list (because list is mutable)

collection.pop()  # Remove any random value
collection.clear()  # Remove all values

set1 = {1,2,3}
set2 = {2,3,4}

u = set1.union(set2)
print(u)

i = set1.intersection(set2)
print(i)
```

Loop: Loops are used to repeat instructions.

While

Code

```
i=1
While i<=5:
    print("Hello")
    i+=1
```

Code

```
heroes = ["ironman", "thor", "superman"]
idx=0
while idx < len(heroes):
    print(heroes[idx])
    idx+=1
```

Break: used to terminate the loop when encountered.

Continue: terminates the execution in the current iteration and continues execution of the loop with the next iteration.

Code

```
nums = (1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
key = 36
i=0
while i<len(nums):
    if (nums[i] ==key):
        print("Found at idx :",i)
        break
    else:
        print("Finding")
    i+=1
```

Code

```
i=0
While i<=5 :
    if (i==3):
        i+=3
        continue
    print(i)
    i+=1
```

for

Code

```
vaggies = ["potato", "brinjal", "ladyfinger", "cucumber"]

for val in vaggies:
    print(val)
```

Code

```
tup = (1,2,3,4,2,3,5)
for num in tup:
    print(num)
```

for with end

```
str = "Hello from Shajib"
for ch in str:
    print(ch)
    if (ch=='x'):
        break
else:
    print("END")
```

Range Function : Range function returns a sequence of numbers, starting from 0 by default, and increment by 1, and stop before a specified number.

range (start?, stop, step?)

Code

```
seq = range(5)
for i in seq:
    print(i)
```

```
for i in rang(5):
    print(i)
```

ending idx

Code

```
for i in range(2,5):
    print(i)
```

starting & ending idx

```
for i in range(2,20,3):
    print(i)
```

start ↓ end ↓ step ↓

Pass Statement : pass is a null statement that does nothing. It is used as a placeholder for future code.

Code

```
for el in range(10):
    pass
```

Function: Block of statements that perform a specific task.

Code

```
def calc_sum(a,b):
    return a+b

sum = calc_sum(200,53)
print(sum)
```

```
cities = ["Dhaka", "Sylhet", "Gazipur", "Mymensingh"]

def print_list(list):
    for item in list:
        print(item, end=" ")

print_list(cities)

def calc_fact(n):
    fact = 1
    for i in range(4, n+1):
        fact *= i
    print(fact)

calc_fact(5);
```

Recursion: When a functions called itself repeatedly.

```
def fact(n):
    if (n==0 or n==1)
        return 1
    else:
        return n*fact(n-1)

print (fact(5))
```

```
dept = ["CSE", "ICT", "TE", "ME"]

def printList(list, idx):
    if (idx<0 or idx>=len(list)):
        return
    print (list[idx])
    printList (list, idx+1)

printList(dept, 0)
```

File I/O in Python :

Types of file < Text Files → .txt , .docx , .log
Binary Files → .mp4 , .mov , .png , .jpeg

Ex

```
f = open("sample.txt", "r")
data = f.read()
print(data)
print(type(data))

f.close()
```

Character & their meaning

r = open for reading (default mode)
w = open for writing, truncating the file first
x = create a new file and open it for writing.
a = open for writing, appending to the end of the file if it exists.
b = binary mode
t = text mode (default)
+ = open a disk file updating (reading & writing)

Read file

```
f = open("text.txt", "rt")
data = f.open(5)        # read only 1st 5 char
print(data)
f.close()

f = open("lineByline.txt", "rt")
while line:
line = f.readline()
while line:
    print(line, end="")
    line = f.readline()
f.close()
```

Write file

```
f = open("hello.txt", "w")
f.write("Now we can change the file txt. /n This is 2nd line!!")
f.close()
```

Append file

```
f = open("hello.txt", "a")
f.write("\n Now we append text using append mode")
f.close()
```

OOP

OOP: OOP is a computer program model that organizes software design around data or objects, rather than function and logic.

Classes: Classes are user defined data type act as a blueprint for individual objects, attributes and method.

Object: objects are instances of a class, created with specifically defined data.

Main Principle of OOP
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

} 4 pillar of OOP.

Encapsulation: Encapsulation is defined as the wrapping up of data and information in a single unit.
↳ Classes are used to achieved Encapsulation.

Abstraction: Abstraction means displaying only essential information and hiding the details.

Inheritance: Inheritance is a process in which objects acquires all the properties & behaviors of its parent object automatically.

Polymorphism: Polymorphism means 'many forms' and it occours when we have many classes that are related to each other by inheritance.

Constructor : A constructor is a special method that is called automatically when an object is created from a class.

Default Constructor : A constructor to which no arrgoments are passed is called Default Constructor.

Ex:
```
class Student:
    name = "Shajib"
    def __init__(self):
        print("Adding new student in Database...")

s1 = Student()
```

Note: self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

Parameterized Constructor : Constructor that can take arguments are known as parameterized constructor.

Ex:
```
class Student:
    def __init__(self, fullname, marks):
        self.name = fullname
        self.marks = marks
        print("Adding new student in the Database...")

s1 = Student("Md.Shajib", 87.50)
print(s1.name)
print(s1.marks)
```

Class & Instance (obj) attributes

```
class Student:
    varsity_name = "MBSTU"      #class attribute
    name = "anonymous"          #class attribute

    def __init__(self, name, marks):
        self.name = name        # object attribute
        self.marks = marks
        print("Adding new student in Database...")

s1 = Student("Shajib", 95)

print(s1.name, s1.marks, s1.varsity_name, Student.varsity_name)
```

obj attribute > class attribute

Methods: Methods are functions that belong to objects.

```
class Student:
    varsity_name = "MBSTU"

    def __init__(self, name, marks):
        self.name = name
        self.marks = marks

    def get_avg(self):
        sum = 0
        for val in self.marks
            sum += val
        print("Hi", self.name, "your avg score is:", sum/3).

s1 = Student("Tony Stark", [99, 98, 100])
s1.get_avg()
s1.name = "Ironman"
s1.get_avg()
```

Static Methods : ↳ Methods that don't use the self parameter.
↳ A static method is bound to a class rather than the objects for that class.

Ex:

```
class Student :
    @staticmethod  # Decorator
    def hello():
        print("Hello from student class")

Student.hello();
```

Decorator : Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it

Abstraction

```
class Car:
    def __init__(self):
        self.acc = False
        self.brk = False
        self.clutch = False

    def start(self):
        self.clutch = True   # Abstraction from user
        self.acc = True      #  "      "    "
        print("Car started...")

c1 = Car()
c1.start()
```

del Keyword: Used to delete object properties or object itself.

Ex

```
class Student:
    def __init__(self, name):
        self.name = name

s1 = Student("Shajib")
print(s1.name)
del s1.name  # delete object name properties.
print(s1.name)  # Gives error
```

Private (like) attributes & methods: Conceptual implementation in Python.

↳ Used only within the class & are not accessible from outside the class.

Ex

```
class Account:
    def __init__(self, accNo, accPass):
        self.accNo = accNo
        self.__accPass = accPass
            # __ indicates that is a private attribute
    def showPass(self):
        print(self.__accPass)

acc1 = Account("12345", "abcde")
print(acc1.accPass)
acc1.showPass()
```

```
class Person:
    __name = "Anonymous"
    def __hello(self):
        print("Hello, person")
    def welcome(self):
        self.__hello()

P1 = Person()
print(P1.__hello())  X Error
P1.welcome()
```

Inheritance ← Single, Multi-level, Multiple

Multilevel:

```
class Car:
    @staticmethod
    def start():
        print("Car started..")

    @staticmethod
    def stop():
        print("Car stopped..")

class ToyotaCar(Car):
    def __init__(self, brand):
        self.brand = brand

class Fortuner(ToyotaCar):  # Fortuner derived from ToyotaCar, which is derived from Car.
    def __init__(self, type):
        self.type = type

car1 = Fortuner("Diesel")
car1.start()
```

Multiple Inheritance:

```
class A:
    varA = "Welcome to class A"

class B:
    varB = "Welcome to class B"

class C:
    varC = "Welcome to class C"

c1 = C()
print(c1.varC)
print(c1.varB)
print(c1.varA)
```

Super Method : super() method is used to access methods of the parents class.

Ex

```
class Car:
    def __init__(self, type):
        self.type = type
    @staticmethod
    def start():
        print("car started...")
    @staticmethod
    def stop():
        print("car stopped")

class ToyotaCar(Car):
    def __init__(self, name, type):
        super().__init__(type)
        self.name = name
        super.start()

car1 = ToyotaCar("Prius", "Electric")
print(car1.type)
```

Class Method: A class method is bound to the class & receives the class as an implicit first argument

Ex

```
Class Person:
    name = "anonymous"
    @classmethod
    def changeName(cls, name):
        cls.name = name

p1 = Person()
p1.changeName("Shajib")
print(p1.name)      # Shajib
print(Person.name)  # Shajib
```

Property Decorator: We use @property decorator on any method in the class to use the method as a property.

Ex:

```
class student:
    def __init__(self, phy, chem, math):
        self.phy = phy
        self.chem = chem
        self.math = math

    @property
    def percentage(self):
        return str((self.phy + self.chem + self.math)/3) + "%"

stu1 = Student(98,96,92)
print(stu1.percentage) => 95.33%
stu1.phy = 89
print(stu1.percentage) => 92.33%
```

[Automatically percentage Update!!]

Polymorphism : Polymorphism means 'many forms' and it occours when we have many classes that are related to each other by inheritance.

Operator Overloading → When the same operator is allowed to have different meaning according to the context.

Operator & Dunder Function

1. a+b ⇒ a.__add__(b)
2. a-b ⇒ a.__sub__(b)
3. a*b ⇒ a.__mul__(b)
4. a/b ⇒ a.__truediv__(b)
5. a%b ⇒ a.__mod__(b)

Code

```
class Complex:
    def __init__(self, real, img):
        self.real = real
        self.img = img
    def showNumber(self):
        print(self.real, "+", self.img, "i")
    def __add__(self, num2):
        newReal = self.real + num2.real
        newImg = self.img + num2.img
        return Complex(newReal, newImg)

num1 = Complex(1, 10)
num2 = Complex(4,5)

sum
sum = num1 + num2

sum.showNumber()
```