Day1(24-08-2022)

Python is a General Purpose Programming Language.
It is Highly Growing and high demand
It was created by Guido van Rossum during 1985- 1990(1991) at mathematics and Science Research center called "CWI"
CWI --> Centrum Wiskunde & Informatica, located in Netherland's.
Also it is a interpreted, interactive, object-oriented and high-level programming language.
Python is dynamically-typed and Garbage-collected programming language.
Python is Open Source and available for Free of Cost.
Python is Easy and simple to Learn.

Python supports different varieties of languages features like
1. Object Oriented Programming Language --> Class, objects
2. Procedure ,, ,, ,,
3. Scripting Language
4. Modular ,, ,,

By using python we can implement the different varieties of applications
1. Web Applications
2. Mobile Applications
3. GUI Applications
4. Data Science
5. Data Analytics
6. Automation
7. Scientific Calculation Apps
8. Gaming Applications
9. Animation Apps
10. Scripting

The Companies which are using python
1. Google
2. Intel
3. NASA
4. PayPal
5. Facebook
6. Youtube
7. Amazon
8. Netflix
9. Pinterest
10. Uber

AC, Wi, IN,F

Careers:
Game developer
Web designer
Python developer/Software Engineer/ Associate/ Data Engineer / Sr Sf Eng / BackEnd Developer
Full-stack developer
Machine learning engineer
Data scientist

Data analyst
Data engineer
DevOps engineer
Software engineer

Day2(25-08-2022)

We can develop a python programs in Two ways
1. Interactive Mode
2. Batch Mode

1. Interactive Mode:
   1. Submitting a python statements one by one to the python interpreter is called as "Interactive mode"
   2. We can submit these statements in Command Line or Python Shell
   3. Once we exit from the command line or python shell, we wil lost the data.

```
a = 10
print(a)   ---> 10

b = "Python"
print(b) -----> Python

print(100+200)   ---> 300
```

   4. we will not use Command Line or Python Shell for developing a projects.
   5. To develop a projects we will use IDE's

2. Batch Mode:
   1. Writting group of python statements and store into a file with .py extension and run that file to python interpreter is called "Batch Mode"
  2. To write a code files, we will use IDE(Integrated Development Envirionment)
  3. Open a file and save with any name along with .py extension
Example: ex1.py

```
a = 100
b = "Python"
c = True
print(a)
print(b)
print(c)
```

Output:

```
100
Python
True
```

   4. Save a file and execute in command prompt --> Go to file location --> python ex1.py

DataTypes:
1. Datatypes are nothing but keywords and which are used to specify what type of data has to store in the variable.

2. If we not specify datatype to a variable, memory will not be allocated for that variable.
3. If memory not allocated, we will not able to store a variable data.
4. Programming Languages supports Two kinds of DataTypes
1. Static DataTypes
2. Dynamic DataTypes

1.Static DataTypes:
   1. In this, Programmer should need to deifne a datatype to the defined variable explicitly.
   2. Once we define the datatype, we will not be able to modify in the entire program.
   3. In this, one variable will store only one type of data
   Example: C, C++, Java, .net -----
        int a=10
        a = 20
        a = "Java" ---> Error
2. Dynamic DataTypes:
   1. In this, Programmer should not need to define a datatype explicitly.
   2. At the time of execution of a program, datatype of a variable will be decided automatically based on the given value.
   3. We can modify the variable of a datatype at any time.
   4. We can able to store different varieties of data.
Example: Python, JavaScript ----

a = 100 ---> int
a = 200 ---> int
a = "python" ---> str
a = 123.78 --> Float

* In python we have categorized datatypes into 2 types
1. Fundamental Types
2. Collection Types

Data --> Name, Age, Ph No, Account No
Operaions(Methods/Functions) --> Withdraw, Deposit, Transfer

DataTypes
Name --- String

Age ---- Integer
name="Anil"

10+20
30-20
20*30

Statistically Typed -->
int a=10
char b = "xyz"
bool c = False

Dynamically Typed --> Python

```
a = 100
b = "xyz"
c = False
```

Day3(26-08-2022)

---

1. Fundamental Types:
1. Fundamental Types are used to store data only.
2. We have the follwing datatypes in python
    1. Integer
    2. Float
    3. Complex
    4. Boolean
    5. Strings
    6. range

1. Integer:

```
a = 100
b = 200
print(a)
type(a)
print(id(a))
```

```
a = 100
b = 200
print(id(a))
print(id(b))
```

```
c = 100
print(id(c))
```

2. Float:

```
e = 123.67
print(e)
print(type(e))
print(id(e))
```

3. Complex:
a+ib --->

```
a = 2+3j
print(a)
print(type(a))
print(id(a))
```

4. Boolean:
True/False

```
a = True
b = False

print(a)
print(type(a))
```

```
print(id(a))


print(b)
print(type(b))
print(id(b))

print(a==b)

5. range:

# Syntax : range(start, end, step)
print(range(10)) --> range(0,10) --> 0,1,2,3,4,5,6,7,8,9
print(range(1,9,2))  --> 1,3,5,7
for i in range(1,9,2):
    print(i)  ----> 1,3,5,7 ---> Line by Line

6. Strings:
1. Sequence or Group of characters are called as "Strings"
2. The Datatype of strings is "str" class
3. We can define a string by using single quotes or Double quotes
4. To work with multiline strings we need to use Triple Single Quotes or
Triple Double Quotes
5. We can represent the string characters by using Index.
6. Both Positive and Negative indexing will support
7. positive Indexing --> Strats from 0  and ends with length -1
8. Negative Indexing --> Starts from -1 and ends with -length
9. Both Positive and Negative Slicing will support
10. Strings are Immutable

a = "Python"
print(a)
print(type(a))
print(len(a))

print("Positive Indexing")
print(a[0])
print(a[1])
print(a[5])

print("Negative Indexing")
print(a[-1])
print(a[-4])
print(a[-6])

# positive Indexing --> Strats from 0  and ends with length -1
# Negative Indexing --> Starts from -1 and ends with length

# Slicing
print("Positive Slicing")
print(a[0:3])  # --> 0,1,2
print(a[7:12])

print("Negative Slicing")
```

```
print(a[-4:])
print(a[-3:-2])
print(a[::-1])

# print(dir(a))
# ['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
# 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
# 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
# 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',
# 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
# 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
# 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
# 'title', 'translate', 'upper', 'zfill']
# Python
# print("Python" == "python".title())

print(a.istitle())
b = "Java"
print(a+ " " +b)
```

Day4(27-08-2022)

---

1. Immutable Objects
2. Mutable Objects

1. Immutable Objects:

1. Once we define an object we can not make any changes or we can not edit
2. We can not create Two diff objects with the same data
3. Will give better performance
4. All fundamental datatypes and Tuples will comes under Immutable objects
5. Applying iterations on Immutable objects takes less time.

Examples:

```
a = 100
a[0] --> Error
b = "Python"
b[0] = "X" ---> Error

x = "Python"
y = "Python"
id(x) --> 1904004132008
id(y) --> 1904004132008
```

2. Mutable:

1. Once we define an object we can make any changes or we can edit
2. We can create Two diff objects with the same data
3. Will give lower performance when compared to Immutable objects
4. List, Set, Dictionary datatypes will comes under mutable objects
5. Applying iterations on mutable objects takes longer time when compared to Immutable objects.

Examples:

```
a = [10,20,30]
b = [10,20,30]
id(a) --> 1904007304392
id(b) ---> 1904007322440

a[0] = 100 --> [100,20,30]
b [1] = 200 --> [10, 200,30]
```

Reading from Keyboard:
1. We can read the data from keyboard by using "input()" keyword/function
2. input() function will always read the data in String format
3. After reading the data in the form of string, we can convert the data in the from of required format by using conversion functions

Examples:

```
first = input("First Word: ")
second = input("second Word: ")
third = "language"
print(first)
print(second)
print(first + " " + second + " " + third)
```

Examples: Conversion

```
first = int(input("First Number: "))
second = int(input("second Number: "))
third = 100
print(first)
print(second)
print(first + second + third)
```

Examples:

```
a = 100
b = 200
x = "Python"
y = "language"
c = 300

print(a+b) --> 300
print(b-a) --> 100
print(a-b) --> -100
print(x+y) --> Python language
print(y+x) --> language Python
print(a+x) --> unsupported Operand error
print(y+c) --> unsupported Operand error
print(str(a)+x) --> 100Python
```

Day5(29-08-2022)

---

2. Collection Types:
1. These are used for storing the objects.
2. We have the following types
    . List

. Tuple
. Set
. Dictionary
3. All collection type objects are iterable objects.
4. By using the methods of a Collection type, we can perform operations on the elements of an object.

1. LIST:
1. Lists can be created by using [] or list() function
2. List objects are Mutable
3. List elements can be either mutable or immutable
4. Both negative and positive indexing supports
5. Insertion order is preserved
6. Duplicate elements are allowed
7. Heterogeneous elements are allowed

Examples:
```
a = []
print(a)
print(type(a))
print(id(a))

a = list()
print(a)
print(type(a))
print(id(a))

a= [10, 13.45,  "abc", 3+4j, True]
print(a)
print(type(a))
print(id(a))
```

Examples:
```
a= [10, 13.45,  "abc", 3+4j, True]
print(a[0])
print(a[-3])
print(a[1:4])
print(a[-3:-1])
```

* To get the methods of string, we need to use the "dir()" command
Example:
```
a = [10,20,30,10,40,20,50,60]
print(a)  -->
print(len(a)) -->
a.append(80) -->
a.count(10)   -->
a.index(40)
a.insert(6,70)  --> .insert(position, value)
a.remove(50)
a.pop()  --> Removes last element/value and return the same value/element
a.extend([100,200])
a.reverse()
a.sort()
a.clear()
```

Example: Nested Lists

```
l = [[10,20,30],[40,50,60], [70,80,90]]
print(l[0])
print(l[-1])
print(l[-2][-2])
```

2. Tuple:

1. Tuple can be created by using () or tuple() function
2. Tuple objects are Imuutable
3. Tuple elements can be either mutable or immutable
4. Both negative and positive indexing supports
5. Insertion order is preserved
6. Duplicate elements are allowed
7. Heterogeneous elements are allowed

Examples:

```
t = ()
print(type(t))
print(id(t))

t = tuple()
print(type(t))
print(id(t))

t = (10, 13.45, "Python", True, 10, 20)
print(t)
print(type(t))
print(len(t))
print(t[0])
print(t[1:5])
print(t[-3])
print(t[-2:-5])
print(t.index(20))
print(t.count(10))
```

NOTE: Tuple supports only two methods called count and index.

Example: Nested Tuples

```
t = ((10,20,30),(40,50,60), (70,80,90))
print(t[0])
print(t[-1])
print(t[-2][-2])
```

Day6 (30-08-2022)

Difference Between List and Tuple:

List	Tuple
1. List objects are Mutable	1. Tuple objects are imutable
2. Lists can be created by [] or list()	2. Tuple can be created by () or tuple()
3. Applying iterations on list takes	3. Applying iterations on tuples takes lessr

Longer time
4. Lists cannot be used as a element of a set. 4. Tuples can be used as a element of a set.
5. Lists can not be used as a key of a Dictionary 5. Tuples can be used as a key of a Dictionary, if tuple has only immutable elements.
6. For frequent operations like Insertion, Retrieval, we will go with Lists. 6. Frequent operations like Updation and deletion, we are going with Tuples.

3. Set: Set is an unordered collection of Unique elements.
1. Set objects are created by {} or set() function.
2. Set objects are Mutable
3. Set elements must be immutable
4. Indexing not supports
5. Insertion order is not preserved
6. Duplicate elements are not allowed
7. Heterogeneous elements are allowed

Examples:

```
s = {1,2,3}
print(s)
print(type(s))

s = set()
print(s)
print(type(s))

s = {1, 10.45, True, "Python", (100,200)}
print(s)
print(type(s))
```

Methods:
'add', 'clear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersection_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_difference', 'symmetric_difference_update', 'union', 'update'

```
s1 = {10,20,30,40,50}
s2 = {40,50,60,70,80}

print(s1)
print(s2)

print(s1 | s2)
print(s1.union(s2))

print(s1 & s2)
print(s1.intersection(s2))

print(s1 - s2)
```

```
print(s2 -s1)
print(s1.difference(s2))
print(s2.difference(s1))

print(s1 ^ s2)
print(s1.symmetric_difference(s2))

4. Dictionary:
1. Dictonaries can be created by {} or dict() function
2. It will represents the data in the form of "Key-Value" pairs
3. Each key-value pair called as "Item"
4. Dictionary  Objects are Mutable
5. Dictionary  keys are Immutable
6. Dictionary values can be Mutable or Immutable
7. Insertion order is preserved
8. Duplicate key are not allowed but values will be allowed
9. Heterogenous key and values are allowed
10. Indexing not supports

Examples:
d = {}
print(d)
print(type(d))

d = {'names': ["Anil", "Bhushan"], "sal":400, "disc": 300}
print(d)
print(type(d)

d['movie'] = "Liger"
d["director"] = "Poori"
d[3] = 300
d[1] = 1000
print(d['sal'])
# print(dir(d))
print(d.keys())
print(d.values())
print(d.items())
print(d['disc'])
print(d.get('sals'))
print(d.popitem())
print(d)
```

Day7 (01-09-2022)

Operators:
Operators are used to perform the operations of the data by using operands.

```
a = 20
b = 10
a + b , a - b, a % b
a, b ---> Opearands
+,-,% ---> Operators
```

1. Arithmetic Operators ---> +, -, *, /, //, %, **
Arithmetic Operators are used to perform the mathematical arithmetic operations
Examples: Addition, Substraction, Division, Floor Division, Modulo Division, Exponent

```
a = 20
b = 10
print(a+b)
print(a-b)
print(a*b)
print(a % b)   # --> Reminder
print(a / b)   # --> Floor division
print(a//b)
```

2. Comparision Operators -->
These are used to compare the data of objects which are defined by operands.
Examples: GreaterThan, LessThan, GreaterThan or Equal, LessThan or Equal, notEqual, Equal

```
a = 20
b = 10
print(a>b)  # True
print(a<b)  # False
print(a>=b) # True
print(a<=b) # False
print(a==b) # False
print(a!=b) # True
```

3. Logical Operators --> These are used to perform logical operators like Logical AND, Logical OR, Logical NOT.

```
a = True
b = False

print(a and b)
print(a & b)

print(a or b)
print(a | b)

print(not a)  # False
print(a != b) # True
```

AND Table
T T T
T F F
F T F
F F F

OR Table
T T T
T F T
F T T
F F F

```
NOT Table
T T T
T F F
F T F
F F T
"""
4. Bitwise Operators   --> Bitwise operators converts the data in the
form of binary format and perform the operations on the binary data.
a = 10
b = 1
print(a & b) # 1010 == 0101
print(a | b)
print(a ^ b)
print(~a)
print(a>>5)
print(a<<5)
"""

# 1010 = 0(2**0) + 1(2**1) + 0(2**2) + 1(2**3) = 0+2+0+8 = 10
# 0101 = 1(2**0) + 0(2**1) + 1(2**2) + 2(2**3) = 1+0+4+0 = 5

5. AssignmentOperators --> These are used to assign the data/value to the
varaiables/operands
a = 20
b = 10
a = a + 5 ---> a+=5
b = b + 6 ---> b+=6
a = a - 5 ---> a -= 5
b = b - 6  ---> b-=6
a = a / 5 --->  a/=5
a = a % 5 --> a%=6
6. Special Operators
    6.1 Membership Operators --> These are used to search for an element
in the given data(Iterable object)
          data  = "Python"
          print("y" in data)
          print("y" not in data)
          print("s" in data)
          print("s" not in data)
    6.2 Identity Operators: These are used to compare the address of
objects which are defined by operands.
           a =  10
      b = 10
      print(id(a))
      print(id(b))
      print(a is b)
      print(a is not b)

      x = [10, 20]
      y = [10, 20]
      print(id(x))
      print(id(y))
      print(x is y)
```

Control Flow Statements:
10000 ---> 20% --> 2000
20000 ---> 10% --> 2000
 Condition: Boolean --> True/False
1. Any expression which return boolean value is known as "Condition"
Block:
1. The set of statements which following some space indentation is known as "Block"

1. Conditional Statements(if, elif, else) -->
* Conditional stmts are used to decide whether block has to execute or skip the execution of a block
* We have three conditional stmts in python
    1. if
    2. elif
    3. else
1. If:
     Once condition returns True, we can execute the if block otherwise skip the execution of "if" block.
syntax:
```
if condition:
    stmt1
    stmt2
    stmt3
```
Example:
```
employee_salary = int(input("Enter Salary:"))
if employee_salary > 5000:
    print("Salary is greaterthan 5000")
```
2. Else:
  * else block will execute whenver previous blocks retuen False
  * we can use the else block in the below
     1. if
     2. elif
     3. while
     4. for
Syntax:
```
   if condition: (False)
      stmt1
      stmt2
  else:
     stmt3
     stmt4
```
Example:
```
employee_salary = int(input("Enter Salary:"))
if employee_salary > 5000:
    print("Salary is greaterthan 5000")
else:
    print("Salary is lessthan 5000")
```
3. elif:
  * whenever elif block preceding block condition returns False then only elif block will execute

```
  * Once execution came to elif block, if elif block condition returns
True then only flow will go inside elif block, other wise execution will
go to next elif block or else block or other lines of code.
Syntax:
   if condition: (False)
      stmt1
      stmt2
  elif condition:(True)
     stmt3
     stmt4
  else:
    stmt5
    stmt6
Example:
employee_salary = int(input("Enter Salary:"))
if employee_salary > 5000:
    print("Salary is greaterthan 5000")
elif employee_salary == 5000:
    print("Salary is equal to 5000")
else:
    print("Salary is lessthan 5000")

Example:
print("Programs Starts Here...........")
num = 10
if num > 0:
    if num % 2 == 0:
        print("Even Number")
    else:
        print("Odd Number")
else:
    print("Number should be greaterthan Zero")
print("Programs Ends Here...........")
2. Looping Statements(While, For):
* If we wanted to execute set of statements repeatedly, we will use Loops
* we have Two looping statements in python
1. While
* While loop executes the set of statements repeatedly until given
condition will becomes False
Syntax:
    while condition:
        stmt1
        stmt2
        stmt .....n
Example:
print("Starts Here......")
i = 1
while i<=5:
    print("Value:", i)
    i = i + 1
print("Ends Here....")
# i = 1
# Iteration1 ------> 1 <= 5 --> True  --> Value: 1 --> 1+1 = 2
# Iteration2 ------> 2 <= 5 --> True  --> Value: 2 --> 2+1 = 3
```

```
# Iteration3 ------> 3 <= 5 --> True  --> Value: 3 --> 3+1 = 4
# Iteration4 ------> 4 <= 5 --> True  --> Value: 4 --> 4+1 = 5
# Iteration5 ------> 5 <= 5 --> True  --> Value: 5 --> 5+1 = 6
# Ends Here ....
```

while-else: Whenever while block returns False then control will goes to else block.

Example:

```
print("Starts Here....")
sums = 0
i = 1
while i <= 100:
    sums = sums + i
    i = i + 1
else:
    print("Morethan", i)
print(sums)
```

Break Statement:

* break is a stmt which can be used in looping stmts
* Whenever control reached to the break stmt then loop execution will terminates.

Example:

```
print("Starts Here")
i = 0
while i < 5:
    i = i + 1
    if i == 3:
        break
    print("Value:", i)
print("Ends Here...")
```

Continue Statement:

* whenever control reached to the continue stmt then without executing the remaining stmts of that iteration control will goes to the next iteration.

```
print("Starts Here")
i = 0
while i < 5:
    i = i + 1
    if i == 2:
        continue
    print("Value:", i)
print("Ends Here...")
```

Infinite While Loop:

Example:

```
print("Starts Here")
i = 0
while True:
    i = i + 1
    print("Value:", i)
print("Ends Here...")
```

* by using the break stmt we can exit from the infinite loop

Example:

```
print("Starts Here")
i = 0
while True:
    i = i + 1
    if i == 2:
        break
    print("Value:", i)
print("Ends Here...")
```

2. For

* For loop executes set of statements with each and every element of given iterable object

Syntax:

```
for variable in iterableObject:
    stmt1
    stmt2
    stmt ..... n
```

Exampe:

```
for i in [10,20,30]:
   print(i)

for s in "Python":
   print(s)

for i in 100: ----> Error: 'int' is not an iterable object
    print(i)

for i in "python":
   print(i * 3)
```

* range() is a predefined function

syntax: range(start, end, step)

Example:

```
for p in range(1, 10):
   print(p)

for q in range(20,30,2):
   print(q)

for n in range(40,30,-1):
    print(n)
```

Example:

```
sums = 0
for i in range(1,101):
    sums =  sums + i
print(sums)

for i in range(1, 11):
    print("5 *",i, "=", 5 * i)
```

Comprehensions:

* The concept of generating elements into the required iterable object by writing some logic into that object.

Examples:

```
print([i**2 for i in range(10) if i % 2 == 0]) ----> List Comprehension -
-> [0, 4, 16, 36, 64]
print({i**3 for i in range(10) if i % 2 == 0}) ----> Set Comprehension --
-> {0, 64, 512, 8, 216}
print((i**3 for i in range(10) if i %2 == 0))  ----> Tuple Comprehension
-> generator object wil gives
print({i:i**2 for i in range(1,6)}) ----> Dict Comprehension ---> {1: 1,
2: 4, 3: 9, 4: 16, 5: 25}
```

Day8(02-09-2022)

Functions:

* Function is a syntax or structure is used to represents the business logic to perform operations

syntax:

```
def functionname(var1, var2 ....):
    """doc string"""
    stmt1
    stmt2
    stmt ....n
```

* Function will not executed automatically
* Function will be executed whenever we make a function call
* we can call/invoke a function N-no.of times

Example:

```
def hello():
    for i in range(5):
        print("Hello")

hello()
# hello()
# hello()
```

Doc String:

* doc string of a function is uded to provide description about a function.
* doc str is an optional

Example:

```
def hello():
    """
    This function has used to print hello 5 times
    """
    for i in range(5):
        print("Hello")
hello()
```

Example:

```
def test(a,b):   ---> a, b are called parameters
    print(a)
    print(b)
    print(a+b)
    print(b-a)
    print(a-5)

test(10,20) --> 10,20 are called Arguments
```

Parameters:
* The variables which are declared with in the function declaration/definition are known as "Parameters"
* Within the function declaration parameters are optional.
* Parameters of a function can be accessed with in the same function.
* we can define 2 types of parameters for the function.
1. Non-Default Parameters
2. Default Prameters
1. Non-Default Parameters or Required/Positional Parameters:
* The Parameters which are declared whithout assigning any value are known as "Non-Default Parameters"
* All the time of callingthe function, we should pass the values to the non-default parameters otherwise we will get error.
Example:

```
def non_default_parms(a, b):
    print(a)
    print(b)
non_default_parms()
# non_default_parms(50)
# non_default_parms(50, 60)
```

2. Default Prameters:
* The Parameters which are declared by assigning some values are known as "Default Parameters"
* At the time of calling the function, we need not to pass the values for the default parameters.
Example:

```
def default_parms(a=100, b=200):
    print(a)
    print(b)
# default_parms()
# default_parms(50)
# default_parms(50,69)
# default_parms(b=50, a=60)
```

* After defining the default parameters, we are not allowed to define non-default parameters otherwise we will get syntax error.
Example:

```
def default_parms(a=100, b):
    c = a + b
    print(c)
default_parms(200)
default_parms(300, 400)
```

Arguments:
* The values which we are passing to the parameters of a function at the time calling/invoking the function are known as "Arguments".
* We can pass two types of arguments for the parameters of a function.
1. Non-Keyword Arguments
2. keyword Arguments

1. Non-Keyword Arguments:

* The arguments which are passed without assigning to parameter names are known as "Non-Keyword Arguments".
Example:

```
def non_keyword(name, age):
   print(name, age)
non_keyword("Anil", 20)
```

2. keyword Arguments:
* The arguments which are passed by assigning to the parameter names are known as "Keyword Arguments".
Example:

```
def keyword(name, age):
   print(name, age)
non_keyword(name="Anil",age= 20)
non_keyword(age= 25, name="BIlla")
```

* After defining the keyword argument, we are not allowed to define non-keyword argument
Example:

```
def keyword(name, age):
   print(name, age)
non_keyword(name="Anil",20) ------> Syntax Error
```

Return Statement:
* A function can return a value or can display a value
* By using the return stmt, we can return the value
* When function returns a value, we can store the value in a variable and we can use that variable in the remaining part of the program.
* If we didn't define use return statement explicitly in the dfunction then bydefault that function returns "None" Value.
Example:

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

def sub(a,b):
    return a-b

res1 = add(10,20)
res2 = sub(40,20)
# res = add(10,20) * sub(40,20)
res = res1 + res2
print(res)
```

Arbitary/Variable length Arguments:
* The paameters which are preceeded by '*' are known as "Arbitary Parameters"
* Arbitary parameters type id taken as a "tuple"
* Atthe time of calling the function we can pass the Zero or more arguments to the arbitray parameters.
Example:

```
def arbitary(*val):
    print(type(val))
    print(val)
    print(len(val))
```

```
arbitary("Feb",20, 40, 50, 60,70)

Example:
def arbitary(month,*val):
    print(type(val))
    print(val)
    print(len(val))
    # print(sum(val))

arbitary("Feb",20, 40, 50, 60,70)

* we can not define the Non-Default parameters after the Arbitary
parameters
def arbitary(*val, month):
    print(type(val))
    print(val)
    print(len(val))

arbitary("Feb",20, 40, 50, 60,70) --------> TypeError: arbitary() missing
1 required keyword-only argument: 'month'

* To overcome/achieve the above we can go with Default parameters
def arbitary(*val, month="Jan"):
    print(type(val))
    print(val)
    print(len(val))

arbitary(20, 40, 50, 60,70)

Example:
# List --> Add value
# Tuple --> Count
# Set --> Remove
# Dict --> Value of a Key
def operations(data, item):
    if type(data) == list:
        data.append(item)
        print(data)
    elif type(data) == tuple:
        count = data.count(item)
        print(count)
    elif type(data) == set:
        data.discard(30)
        print(data)
    elif type(data) == dict:
        print(data[item])
    else:
        print("operations not supported")


lst = {10:100,30:900,50:2500}
operations(lst, 50)
# operations(data=lst, item=50)
# operations(item=lst, data=50)
```

Day9(03-09-2022)

Global Vs Local

Global Variables:

* The variables are which are defined with in the programs and outside of all the functions is called "Global Variable"
* For all the variables, memory will be ctreated once
* Global variables of a program, we can access in all the functions

Example:

```
a = 1000
def global_example():
    print("Inside:", a)
print("Outside:", a) ---> 1000
global_example()     ----> 1000
```

Example:

```
a = 1000
def global_example():
    global a
    a = 2000
    print("Inside:", a)

print("Before Calling:", a)  ---> 1000
global_example()           ---> 2000
print("After Caling:", a)    ---> 2000
```

Local Variables:

* The variables which are defined with in the function body is called "Local Variable"
* For Local variables, memory will be allocated whenever we call that function
* Local variables of one function can not be accessed from outside of that function.

Example:

```
a = 1000
def local_example():
    b = 2000
    print("Inside:", a)
print("Outside:", a)
print("Outside:", b)
local_example()
```

Example:

```
a = 1000
def local_example():
    b = 2000
    print("Inside:", a)

def add():
    print(a+10)
    print(b+10)
```

```
def sub():
    print(a-10)

print("Outside:", a)
print("Outside:", b)
local_example()

 add()
 sub()
```
* If we wanted to modify the global variable values with in the function, we should define a forward declaration of that variable.

Example:
```
a = 1000
def test1():
    global a
    a = 2000
    print(a)

def test2():
    print(a)

test1()   #---> 2000
test2()   #---> 2000
```

Recursive Function:
* A function which is called by itself is known as "Recursive function"
* Python supports recursive function caling but it will not support infinite recursion.

Example:
```
def add():
    print(100+200)
    add()
add() ---------> It will print almost 998 times,so we need to control the infinite loop
```

Example:
```
count = 0
def add():
    global count
    count = count + 1
    while count < 5:
        print(100+200)
    add()
add()
```

Lambda function / Anonymous Functions:
* A function which doesn't have anyname is known as "Lambda function or Anonymous Function"
* A lambda function we can define by using the below syntax

Syntax: lambda arguments: expression
* we can assign the lambda function to a variable and we can call the lambda function through variable.

Example:(Normal Function)
```
def square(num):
```

```
    return num ** 2
res = square(10)
print(res)

Example: (Lambda  Function)
res = lambda x: x ** 2
print(res(10))

Filter, Map, reduce:
Filter:
* The filter() Function takes a function object and an iterable and
creates a new list.
* As the name suggests, filter() forms a new list that contains only
elements that satisfy a certain condition, i.e. the function we passed
returns True.
Syntax:
filter(function, iterable(s))
Examples:
# filter_vals = list(filter(lambda x: (x%2==0),
[1,2,4,5,3,78,98,88,97,890]))
filter_vals = list(filter(lambda x: (x%2==0), range(10)))
print(filter_vals) ------->[0,2,4,6,8]

Map:
* The map() function iterates through all items in the given iterable and
executes the function we passed as an argument on each of them.
Syntax :
map(function, iterable(s))
Example:
map_vals = list(map(lambda x: (x+3), range(10)))
print(map_vals)
Reduce:
* reduce() works differently than map() and filter(). It does not return
a new list based on the function and iterable we've passed. Instead, it
returns a single value.
* Also, in Python 3 reduce() isn't a built-in function anymore, and it
can be found in the functools module.
Syntax:
reduce(function, sequence[, initial])
Example:
from functools import reduce
reduce_vals = reduce(lambda x,y : x+y, range(7))
print(reduce_vals)

Modules:
* Every python file itself is a one module.
* A python file or module can contain executable statements, global
variables, functions and classes.
* We can use the properties of one module into another module by using
"import Statement"
* Python supports 2 types of import statements
1. Normal import
2. from import
```

1. Normal import:
* In normal import, for entire python module one object created and it imports that object.
* We can access the properties of the module object by using "module name:
Examples: modules_ex.py

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

def sub(a,b):
    return a-b

def mul(a,b):
    return a*b

def div(a,b):
    return a//b
```

modules_ex2.py:

```
import modules_ex

addition = modules_ex.add(10,20)
print(addition)

substraction = modules_ex.sub(30,10)
print(substraction)

multiply = modules_ex.mul(10,20)
print(multiply)

dividion = modules_ex.div(30,5)
print(dividion)
```

* At the time of importing module by using normal import, we can create "alias" name for the module. We can access the properties of the module through alias name.
modules_ex_aliases.py:

```
from modules_ex import add as a, sub as s, mul as m, div as d

addition = a(10,20)
print(addition)

substraction = s(30,10)
print(substraction)

multiply = m(10,20)
print(multiply)

dividion = d(30,5)
print(dividion)
```

2. from import:

* Instead of importing entire module, we can import required/specific properties of the module by using "from import" and we can access those properties directly without using module name or alias name.
Example:

```
from modules_ex import add, sub, mul, div

addition = add(10,20)
print(addition)

substraction = sub(30,10)
print(substraction)

multiply = mul(10,20)
print(multiply)

dividion = div(30,5)
print(dividion)
```

Packages:
1. Package is nothing but a folder or directory, which represents the collection of python modules.
2. A package can also contain sub packages.
3. We can import the modules of a package by using packagename.modulename.
Example:
* Create a folder/directory "package" and write the below files into folder.
test1.py:

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

test2.py:

```
def sub(a, b):
    return a - b
```

test3.py:

```
def mul(a, b):
    return a * b
```

test4.py:

```
def div(a, b):
    return a // b
```

Example: package_ex.py

```
from package import test1, test2, test3, test4
addition = test1.add(10,20)
print(addition)

substraction = test2.sub(40,20)
print(substraction)

multiply = test3.mul(10,20)
print(multiply)
```

```
division = test4.div(100,20)
print(division)
```

File Handling:
* A file is named location on the disk and which stores the data permanently.
* We can open the file, read/write the data in a file and close a file.
* We can open the file by calling open() function.
* While opening a file, we need to specify the mode.

Modes:
r -----> Open a file in read mode
w ---> Open a file for write mode and creates a new file if not exists
a ---> Open a file for appending a data at end of the file without deleting the existing data.
t ---> Open a file in text mode
* After executing the open() function it returns the file object.
* File object provides various methods and by using those methods we can perform the read, write operations on the files and we can close the file.
* we can close the file by using close() function.

Example1:

```
data = open("files_ex.txt", "r")
print(data.read())
data.close()
```

files_ex.txt:

```
Hiiii
How are you
What are you doing
Where are you
When will you come
What is your name
Byeeeeee
```

Example2:(Read)

```
data = open("files_ex.txt", "r")
# print(data.read())
# print(data.readline()) ----> Read the data line by line
# print(data.readlines()) ---> Read all the data and returns in a list
lines = data.readlines()
for line in lines:
    if 'you' in line.split():
        print(line)
# print(data.readline())
# print(data.tell())   --> Tells the current cursor position
# data.seek(45) -------->  Specified cursor positin
# print(data.tell())
data.close()
```

Example: (Write)

```
data = open("files_ex1.txt", "w")
data.write("Anil\nBilla\nKumar")
data.close()
```

Example:(Append)

```
data = open("files_ex1.txt", "a")
data.write("Anil\nBilla\nKumar")
data.close()
```

Example:

```
lists = ["Hii", "Hello", "How are you", "Bye"]
data = open("files_ex3.txt", "a")
for lst in lists:
    data.write(lst)
    data.write("\n")
```

Exception Handling:

* We get the two types of errors in any programming languages. They are

1. Syntax Errors
2. Run Time Errors

1. Syntax Errors:

* The errors which occurs because of invalid syntax are known as "Syntax Errors".
* Whenever we run the python file, if any syntax error is occured then compiled python file(pyc) will not be generated for that program.
* Without getting the compiled python file(pyc), program execution will not be started.
* Python programmer or developer is responsible for providing the solutions to the syntax errors.

2. Run Time Errors:

* These errors which occurs after starting the execution of the program are known as "Runtime errors"
* There are so many reasons behind to get the runtime errors .

1. Invalid input
2. Invalid logic

Abnormal Termination:

* The concept of terminating the program in the moddle of its execution eithout executing last statement of the main module is known as "abnormal termination".

Example:

```
print("Starts .....")
num1 = int(input("Enter a number1:'))
num2 = int(input("Enter a number2:"))
res = num1 + num2
print(res)
print("Ends .....")
```

Exception Handling:

* The concept of identifying the raised exception, receiving that exception and assigning that exception to corresponding exception to the corresponding class is known as "Exception Handling".
* Exception Handling can be implemented by using try and except block.

1. Try Block:

* A block which is created by "try" keyword is known as "try block"

Syntax:

```
try:
```

```
    stmt1
    stmt2
```
* Try block will raise the exception and catch the exception in except block.

2. Except Block:
* A block which is preceded by try block and created by except keyword is known as "Except Block"
* Except block receives the exception which is given by the try block and assign that exception to the corresponding exception class.

Exception:
```
print("Starts ....")
num1 = int(input("Enter a number1:"))
num2 = int(input("Enter a number2:"))
try:
   res = num1 / num2
   print(res)
except:
   print("Number2 can not be Zero")
print("Ends .....")
```

Single Try with multiple Except Block:
* Whenever we define single try with multiple except blocks if any exception is raised in try block then control goes to the first except block.
* If that except block is not handle that exception then only control will goes to the next except block.
* If all except blocks are not handle that exception then program will be terminated abnormally.

Finally Block:
* A block which is preceded by the finally keyword is known as "finaly" block

Syntax:
```
finally:
   stmt1
   stmt2
```

Example:
```
a = input("Enter value 1:")
b = input("Enter Value 2:")
try:
    a = int(a)
    b = int(b)
    c = a/b
except ZeroDivisionError:
    print("b value must be greaterthan 0")
except ValueError:
    print("b value must be provided")
else:
    print("Yeah!, you are correct")
finally:
    print("Bye")
```

Example:

```
data = {'a': 10,'b': 20, 'c': 0, 'd': 0}
a = int(input("Enter value 1:"))
zeros = []
for key, value in data.items():
    try:
        c = a / value
        print(c)
    except ZeroDivisionError:
        zeros.append(key)
    except Exception:
        pass
print(zeros)
```

Nested try block:

* The concept of defining one try block inside another try block is known as a "nested try" block.

Example:

```
try:
    print("In try1")
    try:
        print("In try2")
    except:
        print("In except1")
    finally:
        print("In finally1")
except:
    print("In except2")
    try:
        print("In try3")
    except:
        print("In except3")
    finally:
        print("In finally2")
finally:
    print("In finally3")
    try:
        print("In try4")
    except:
        print("In except4")
    finally:
        print("In finally4")
```

Used Defined Exceptions:

* The exceptions which are defined by the programmers explicitly according to their business requirements is known as "user-defined exceptions"

1. Defining user defined exception class:

* Defining any user defined class by extending any one of the predefined exception class is known as "User Defined exception class"

2. Raising the exception:

* Creating the user defined exception class object by using "raise" keyword is known as "raising exception"

3. Handling the raised exception:
* We can handle the raised exception by using try and except blocks.
Example:

```
class ValueTooSmallError(Exception):
    pass

class ValueTooLargeError(Exception):
    pass

number = 10
while True:
    try:
        inp_num = int(input("Enter a Number:"))
        if inp_num < number:
            raise ValueTooSmallError
        elif inp_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is Too Small!, Try again")
    except ValueTooLargeError:
        print("This value is Too Large, Try again")
    except Exception:
        print("Any other error")
```

Regular Expression:

Day10(05-09-2022)

Advanced Python:
OOPS:
* OOPS principle are nothing but certain Rules and Guidelines.
Principles:
1. Encapsulation
2. Abstarction
3. Inheritance
4. Polymorphism
Benifits:
1. Security
2. Flexibility
3. Re-usablity

* The programming languages which is going to satisfy the OOPS priciples in any application we call it as OOP Languages.
Ex: C++, Java, Python
Example: Bank
cname =
caddr =
caccno =
cbal =
ename =
eaddr =

```
eid =

def deposit():
   pass

def withdraw():
   pass

def bal():
   pass

def hra():
   pass

def tax():
   pass
```

Class:
* A class is a structure or blueprint and it is used to group related data members along with its related functionalities.
Syntax:

```
class ClassName:
   """
     Doc String
   """
```

* By using the variables we can reperesents the data
* 3 types of variables
1. Static Variable OR class Variable OR Global Variable
2. Non-Static Variable OR Instance Variable
3. Local Variable
* 3 types of methods
* By using the methods We can represents the functionalities/operations.
1. Static Methods
2. Non-Static Methods
3. Class Methods

Object:
* Instance of a class is called "object".
Syntax:

```
referenceVariable = ClassName()
```

Constructor:(__init__())
* A constructor is a special method and it is used to create a non-static varaibles of a class at the time of creating an object.
* Constructor will be executed automatically at the time of creating an object
Example:

```
class Car:
    '''
        This class used to represent the Car details
    '''
    def message(self):
        print("Hello World")

bmw = Car()
```

```
bmw.message()

audi = Car()
audi.message()
```

1. Static Variable OR class Variable OR Global Variable
* The variables which are declared within the class and outside of all the methods is called " Static variables"
* The data which is common for all the objects, in this case we will go with static variables
* For all the static variables memory will be allocated only once
* We can access with in all the methods of a same class by using class name
* We can access outside of a class by using Class name

Example:

```
class Car:
    '''
        This class used to represent the Car details
    '''
    color = "Red"
    name = "BMW"
    names = ["Anil", "Bhushan", "Naga", "Seenu", "Usha"]

    def get_color(self):
        print(Car.color)

    def get_name(self):
        print(Car.name)

    def get_details(self):
        print(Car.name + ":" + Car.color)

    def looping_data(self):
        for name in Car.names:
            print(name)

c1 = Car()
c1.get_color()
c1.get_name()
c1.get_details()
c1.looping_data()

print(Car.name)
print(Car.color)
```

2. Non-Static Variable OR Instance Variable:
* The variables which are declared within the class with self keyword is called " Non Static variables"
* The data which is seperate for all the objects, in this case we will go with non static variables
* For all the non static variables memory will be allocated when we create an object
* We can access with in all the methods of a same class by using self

* We can access outside of a class by using reference variable
Example:

```
class Car:
    '''
        This class used to represent the Car details
    '''
    def get_color(self):
        self.color = "Red"
        print(self.color)

    def get_name(self):
        self.name = "BMW"
        print(self.name)

    def get_details(self):
        print(self.name + ":" + self.color)

c1 = Car()
c1.get_color()
c1.get_name()
c1.get_details()
```

Example:

```
class Car:
    '''
        This class used to represent the Car details
    '''
    def __init__(self, nm, clr):
        self.name = nm
        self.color = clr

    def get_color(self):
        print(self.color)

    def get_name(self):
        print(self.name)

    def get_details(self):
        print(self.name + ":" + self.color)

c1 = Car("BMW", "Red")
c1.get_color()
c1.get_name()
c1.get_details()

c2 = Car("Audi", "White")
c2.get_color()
c2.get_name()
c2.get_details()

c3 = Car("Hundai", "Blue")
c3.get_color()
c3.get_name()
c3.get_details()
```

3. Local variables:
* The variables which are declared within the method are called "Local variables"
* The data which is required to use with in the specific method, in this case we will go with local variables
* For all the non static variables memory will be allocated when we call that method
* We can not access from outside of a method
Example:

```
class Car:
    '''
        This class used to represent the Car details
    '''
    def get_color(self):
        color = "Red"
        print(color)

    def get_name(self):
        name = "BMW"
        print(name)

    # def get_details(self):
    #     print(self.name + ":" + self.color)

c1 = Car()
c1.get_color()
c1.get_name()
# c1.get_details()
```

1. Encapulation:
* Wrapping/Grouping of related data members into along with its related functionality is known as "Encapsulation".
Example: Customer Class

```
cname =
caddr =
caccno =
cbal =

def deposit():
   pass

def withdraw():
   pass

def bal():
   pass
```

Employee Class:

```
ename =
eaddr =
eid =

def hra():
```

```
   pass

def tax():
   pass

Example:
class Car:
    location = "Bangalore"

    def __init__(self, model, color, price):
        self.color = color
        self.price = price
        self.model = model

    def get_price(self):
        company = "Maruthi Suzuki"
        print(Car.location+ "::" + company + "  --->  " + self.model + ":
" + self.price)

name = input("Enter car name:")
color = input("Enter car color:")
price = input("Enter car price:")
c1 = Car(name, color, price)
c1.get_price()

c2 = Car("Swift", "White", "500000")
c2.get_price()
```

Day11(06-09-2022)

Methods:

1. Static Methods:
* Static methods are used to group functions which have some logical connection with a class to the class.
* Static methods will have @staticmethod property.

syntax:

```
class X:
   @staticmethod
   def m1():
      pass
```

2. Non-Static Methods
* By default all the methods in a class which is having self as a first parameter are called "Non-static methods"

Syntax:

```
class X:
   def m1(self):
      pass
```

3. Class Methods
* Nothing to do with static variables and non static variables, in that scenario we will go with class method
* Class methods will have @classmethod property and the first argument should be "cls"

Syntax:

```
class X:
   @classmethod
   def m1(cls):
      pass
```

Example:

```
class Car:
    name = "BMW"
    def __init__(self, price):
        self.price = price

    def get_data(self):
        print("In non static method")

    @staticmethod
    def get_data2():
        print("In static Method")

    @classmethod
    def get_data3(cls):
        print("In class method")

c1 = Car(1000000)
c1.get_data()
c1.get_data2()
c1.get_data3()
```

Methods Vs Constructor:

Method	Constructor
1. The method name shouls be any name	1. Constructor name should be __init__
2. Methods will be executed whenever we call that method	2. Constructor will be executed automatically when we create an object
3. With respect to one object, one method can be called number of times	3. With respect to one Constructor, one constructor will be executed only once
4. Methods are used represents business logic	4. These are used to initialize nonstatic variable

Garbage Collection:

* The concept of removing unused or unreferenced objects from the memory location is known as "Garbage Collection"
* If the reference count of any object is zero, then we call that object as unused or unreferenced object.
* The no.of reference variables which are pointing to an object is known as "reference count".
* At the time of executing a program, if any object reference count becomes zero then internally python interpreter calls garbage collector.
* Garbage collector is a predefined program or background thread, which removes unused or unreferenced object from the memory location.

Destructor:
* It is a special method and the name should be "__del__().
* Destructor of a class will be executed automatically at the time of deleting the object from memory location.
Example:

```
class Car:
    def __init__(self, name): # Constructor
        print("In Car Class Constructor")

    def display(self):
        print("In display")

    def __del__(self): # Destructor
        print("In Destructor")

c1 = Car("BMW")
c1.display()
del c1

c2 = Car("BMW")
c2.display()
del c2
```

Abstraction:
* The concept of hiding the properties of a class from outside of the class is known as "Abstraction or Data Hiding"
* Any property of a class which is preceeded by "__" is going to be hidden from the class.
Example:(Outside of a class)

```
class Car:
    __name = "BMW"

    def display(self):
        print(Car.__name)

c1 = Car()
c1.display()
print(Car.__name)
```

Example: (SubClass)

```
class Car:
    __name = "BMW"

    def display(self):
        print(Car.__name)

class Vehicle(Car):
    __color = "Red"

    def display(self):
        print(Vehicle.__color)
        print(Car.__name)

v1 = Vehicle()
```

```
v1.display()
```

* Python provides a special syntax to access the hidden properties of a class from out side of a class and in sub classes.

Example:(Outside of a class)

```
class Car:
    __name = "BMW"

    def display(self):
        print(Car.__name)

c1 = Car()
c1.display()
print(c1._Car__name)
```

Example: (In subclass)

```
class Car:
    __name = "BMW"

    def display(self):
        print(Car.__name)

class Vehicle(Car):
    __color = "Red"

    def display(self):
        print(Vehicle.__color)
        print(self._Car__name)

v1 = Vehicle()
v1.display()
```

Day12(07-09-2022)

---

3. Inheritance:

* The concept of inhereting/using the properties of one class into another class directly like a same class members is known as "Inheritance".
* A class which is extended by another class is known as "Super class or Base class or Parent Class"
* A class which is extending another class is known as "Sub Class or Derived Class or hild Class"
* Super class properties directly we can access into the subclass like as a subclass members.

Example:

```
class A:
    val1 = 1000
    def display_A(self):
        print("In A class display method")

class B(A):
    val2 = 2000
    def __init__(self):
```

```
        self.val3 = 3000

    def diplay_B(self):
        print("In B class display method")

    def display_values(self):
        print(A.val1)
        print(B.val2)
        print(self.val3)
        self.diplay_B()
        self.display_A()
        # A.display_A(self)

b = B()
b.display_values()
print(A.__bases__)
print(B.__bases__)
# B.display_values(self)
```

Types of Inheritance:
1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Cyclic Inheritance
6. Hybrid Inheritance

1. Single Inheritance:
* The concept of inheriting the properties from one class into another class is called "Single Inheritance".

Syntax:

```
class A:
   pass
class B(A):
  pass
```

Example:

```
class A:
    def display_A(self):
        print("In A class display method")

class B(A):
    def display_B(self):
        print("In B class display method")
b = B()
b.display_B()
b.display_A()
```

2. Multilevel Inheritance:
 The concept of inheriting the properties from multiple classes into one class with the concept of one after another is called "Multilevel Inheritance".

Syntax:

```
Class A:
   pass
```

```
Class B(A):
   pass
Class C(B):
   pass

Example:
class A:
    def message_A(self):
        print("In Class A message method")
class B(A):
    def message_B(self):
        print("In Class B message method")
class C(B):
    def message_C(self):
        print("In Class C message method")

c = C()
c.message_C()
c.message_B()
c.message_A()
```

3. Hierarchical Inheritance:

The concept of inheriting the properties from one class into multiple classes seperately is called "Multilevel Inheritance".

Syntax:

```
Class A:
   pass
Class B(A):
   pass
Class C(A):
   pass

Example:
class A:
    def message_A(self):
        print("In Class A message method")
class B(A):
    def message_B(self):
        print("In Class B message method")
class C(A):
    def message_C(self):
        print("In Class C message method")

c = C()
c.message_C()
# c.message_B()   # Error
c.message_A()

b = B()
b.message_B()
b.message_A()
# b.message_C()  # Error
```

4. Multiple Inheritance:

* The concept of inheriting the properties from multiple classes into single class at a time is called "Multiple Inheritance".

Syntax:

```
Class A:
   pass
Class B:
   pass
Class C(A, B):
   pass
```

Example:

```
class A:
    def message_A(self):
        print("In Class A message method")
class B:
    def message_B(self):
        print("In Class B message method")
class C(B,A):
    def message_C(self):
        print("In Class C message method")

c = C()
c.message_C()
c.message_B()
c.message_A()
```

Example: MRO --> Method Resolution Order --> Left to Right

```
class A:
    def message_A(self):
        print("In Class A message method")

    def test(self):
        print("In Class A test method")

class B:
    def message_B(self):
        print("In Class B message method")

    def test(self):
        super().test()
        print("In Class B test method")

class C(B,A):  # MRO --> Method Resolution Order --> Left to Right
    def message_C(self):
        print("In Class C message method")

    def test1(self):
        print("In Class c test method")

c = C()
c.test()
```

5. Cyclic Inheritance:

* The concept of inheriting the properties from sub class to super class is known as "Cyclic Inheritance"
* Python doesn't supports the Cyclic Inheritance

Syntax:

```
Class A(C):
   pass
Class B(A):
   pass
Class C(B):
   pass
```

Example:

```
class A(C):
    def message_A(self):
        print("In Class A message method")
class B(A):
    def message_B(self):
        print("In Class B message method")
class C(B):
    def message_C(self):
        print("In Class C message method")

c = C()
c.message_C()
c.message_B()
```

6. Hybrid Inheritance:
* Combination of single, multi-level, hierarchical and multiple inheritances is called "Hybrid Inheritance"

4. Polymorphism:
* Poly means Many, morphism means forms and forms means functionalities or logic.
* The concept of defining multiple logics to perform the same operation is called "Polymorphism"
* Polymorphism can be achieve/implemented by using method overloading and method overriding.

1. Method Overloading:
* The concept of defining multiple methods with same name and same no.of parameters or different no.of parameters with in a class is called "Method Overloading".

Syntax:

```
class A:
   def calc(self):
      pass
   def calc(self, a):
      pass
   def calc(seld, a,b):
      pass
```

Example:

```
class A:
    def calc(self):
        print("In A class calc method - 0 Params")
```

```
    def calc(self, a):
        print("In A class calc method - 1 Param")

    def calc(self, a, b):
        print("In A class calc method - 2 Params")

    def calc(self, a, b, c):
        print("In A class calc method - 3 Params")

a = A()
# a.calc()
# a.calc(10)
# a.calc(10,30)
a.calc(10, 20, 30)
```

NOTE: By default in python, last defined method only will be executed, To overcome that we need to do below

Example:

```
class A:
    def calc(self, a=None, b=None, c=None):
        print("In A class calc method - 3 Params")

a = A()
a.calc()
a.calc(10)
a.calc(10,30)
a.calc(10,30,50)
```

2. Method Overriding:

* The concept of defining multiple methods with same no.of params or diff no.of params that is one is in Super class and other is in sub class is called "Method Overriding".

Example:

```
class A:
    def calc(self):
        print("In A class calc method - 0 Params")

class B(A):
    def calc(self):
        print("In B class calc method - 0 Param")
b = B()
b.calc()

a = A()
a.calc()
```

Example:

```
class A:
    a = 1000
    def calc(self):
        print("In A class calc method - 0 Params")

class B(A):
    def calc(self):
```

```
        print("In B class calc method - 0 Params")

    def calc_new(self):
        self.calc()
        super().calc()
        # print(super().a)
        print("In B class calc_new method")

b = B()
# b.calc()
b.calc_new()

Example:
class A:
    def __init__(self):
        self.val1 = 1000
        self.val2 = 2000


class B(A):
    def __init__(self):
        super().__init__()
        self.val3 = 3000
        self.val4 = 4000

b = B()
print(b.val1)
print(b.val3)
print(b.val4)

# a = A()
# print(a.val1)
```

Day13(08-09-2022)

MultiThreading:
* Thread is a light-weight process
* Thread is a logic or functionality which executes symultaneously along with the other part of the program
* A program which is under execution is known as "process"
* we can define the functionality or logic as a thread by overriding run() method of thread class.

```
def run(self):
   pass
start()
```

* Thread class is a predefined class, which is defining from the threading module.
* Threading module is a predefined module
* If we wanteds to execute the thread by calling start() method of a class.

Examples:

```
"""
```

```
# Sequencial Process
class ABC:
    def m1(self):
        for i in range(1, 101):
            print("In ABC", i)

class XYZ:
    def m2(self):
        for i in range(101, 201):
            print("In XYZ",i)

abc = ABC()
abc.m1()

xyz = XYZ()
xyz.m2()
"""
"""
# import threading
from threading import Thread

class ABC(Thread):
    def run(self):
        for i in range(1, 101):
            print("In ABC", i)

class XYZ(Thread):
    def run(self):
        for i in range(101, 201):
            print("In XYZ",i)

abc = ABC()
# abc.run()
abc.start()

xyz = XYZ()
# xyz.run()
xyz.start()
"""
"""
import threading
class ABC(threading.Thread):
    def run(self):
        for i in range(1, 101):
            print(i)

a = ABC()
a.start()
b = ABC()
b.start()
"""
"""
import threading
```

```
class ABC(threading.Thread):
    def run(self):
        func()

class XYZ(threading.Thread):
    def run(self):
        func()

def func():
    for i in range(1, 101):
        print(i)

a = ABC()
a.start()

x = XYZ()
x.start()
"""
"""
import threading

class ABC(threading.Thread):
    def run(self):
        func()

class XYZ(threading.Thread):
    def run(self):
        func()

def func():
    for i in range(1, 101):
        print(i)

a = ABC()
a.start()

x = XYZ()
x.start()

for i in range(201, 301):
    print(i)
"""
```

Scheduling:

* Among multiple threads which thread has to start the execution first and how much time that thread has to execute after allocation of time is over, which thread has to continue the execution next is comes under scheduling.

* We can stop the execution of the threads in the middle of its execution temporarly by calling sleep() function or by calling join() method.
* sleep() predefined function in the time module
* sleep() function stop the execution of a current thread until given time is over.

Example:

```
from threading import Thread
import time

class ABC(Thread):
    def run(self):
        time.sleep(5)
        for i in range(1, 101):
            print("In ABC", i)

class XYZ(Thread):
    def run(self):
        time.sleep(5)
        for i in range(101, 201):
            print("In XYZ",i)

abc = ABC()
# abc.run()
abc.start()

xyz = XYZ()
# xyz.run()
xyz.start()
for i in range(201, 301):
    print(i)
```

* join() method is predefined method which is defined in thread class.
* join() method stops the execution of a current thread until specified thread is over.

Example:

```
from threading import Thread
import time

class ABC(Thread):
    def run(self):
        time.sleep(10)
        for i in range(1, 101):
            print("In ABC", i)

class XYZ(Thread):
    def run(self):
        time.sleep(5)
        for i in range(101, 201):
            print("In XYZ",i)

abc = ABC()
abc.start()

xyz = XYZ()
xyz.start()
for i in range(201, 301):
    print(i)

abc.join()
xyz.join()
```

Synchronization:
* The concept of avoiding multile threads to access the same function logic at a time is caled "Synchronization"
* Synchronization can be implemented by calling acquire() and release() methods, which is defined Lock class.
* Lock class is a predefined class, which is defined in threading module.
* Synchronization will degrates the performance of the application.
Example:

```
import threading
import time

class ABC(threading.Thread):
    def run(self):
        lc.acquire()
        func("Django")
        lc.release()

class XYZ(threading.Thread):
    def run(self):
        lc.acquire()
        func("Flask")
        lc.release()

def func(msg):
    print("[Hello [", msg)
    time.sleep(10)
    print("] World]")

lc = threading.Lock()
abc = ABC()
abc.start()

xyz = XYZ()
xyz.start()
```

Explanation:
* Here lc.acquire() methos wil see whether any thread is running or not. If not running, then it will starts or lock.
  First 'ABC' class acquire() method will process 'print("[Hello [", msg)' and then waiting for 10 seconds and execute the 'print("] World]")' along with any other statements if exists.
* Once this process has completed then release() method will release the lock and then "XYZ" class acquire() method will starts the executions and follows remaining.

POC:
1. What is Process
2. What SubProcess
3. What is Thread
4. What is MultiThread
5. What is MultiProcess
6. MultiPrcocess vs MultiThreading ----------> Important
7. Synchronization vs Asynchronization
8. Semaphore

Thread Life Cycle:

Day14(09-09-2022)

Iterators:

* Iterator is protocol ---> Some Rules and Guidelines or regulation --->(__iter__() and __next__())
* If any class is implementing iterator protocol that class object we will convert into iterable object.
* If any class containing the two methods called __iter__() and __next__() then we can call that class as a itearator class
* we will convert the class object into iterable object by using iter().
* When we call the iter function on any object then internally __iter__() method of that class will be executed.
* Iterable object can store Zero or More elements.
* To get the elements one by one from the iterable object we call next() function on iterable object.
* When we call the next function on any object then internally __next__() method of that class will be executed.
* According to the iterator protocol rules __iter__() method should return the object and __next__() method should contain the StopIteration exception raising statement.

Example:

```
class Calculations:
    def __init__(self, limit):
        self.limit = limit

    def __iter__(self):
        self.num = 0
        return self

    def __next__(self):
        if self.num <= self.limit:
            result = 2 ** self.num
            self.num += 1
            return result
        else:
            raise StopIteration

calc = Calculations(5)
itr = iter(calc)
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))


class Calculations:
```

```
    def __init__(self, limit):
        self.limit = limit

    def __iter__(self):
        self.num = 0
        return self

    def __next__(self):
        if self.num <= self.limit:
            result = 2 ** self.num
            self.num += 1
            return result
        else:
            raise StopIteration

calc = Calculations(5)
itr = iter(calc)
while True:
    try:
        print(next(itr))
    except StopIteration:
        break
```

* In python, for lopp is implemented like below

```
for element in iterable:
    itr_obj = next(iterable)
    while True:
        try:
            element = next(itr_obj)
        except StopIteration:
            break
```

* For loop takes the given object and converts that object in the form of iterable object and get the elements one by one from the itrable object.
* While getting the elements in one by one from the iterable object, it will give StopIteration exception is raised. For loop interanally handles that exception.

Example:

```
class Calculations:
    def __init__(self, limit):
        self.limit = limit

    def __iter__(self):
        self.num = 0
        return self

    def __next__(self):
        if self.num <= self.limit:
            result = 2 ** self.num
            self.num += 1
            return result
        else:
            raise StopIteration

for res in Calculations(5):
    print(res)
```

Example:

```
lst = [10,30,40,60,90]
itr = iter(lst)

print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
"""
lst = [10,30,40,60,90]
itr = iter(lst)
while True:
    try:
        print(next(itr))
    except StopIteration:
        break
```

Generator: yield

* Generator is a function which contains one or more yield statements.
* Whenever we call the next function on the iterable object then controls goes to first yield statements and return that yield value.
* Again if we call next function then control will starts from the previous yield statement and executes the curent yield staments.
* If no more yield statments then we will get StopIteartion exception.
* Whenever control reached to the yield stmt of a function without terminating the function execution, it will pause the execution and control comes out from the function execution.

Example:(Normal Function)

```
def func():
    return "Anil"
print(func())
```

Example:

```
def generator():
    i = 1
    yield "Anil"
    i = i + 1
    yield "Naga"
    i = i + 1
    yield "Bhushan"
    i = i + 1
    yield "Usha"
gen = generator()
# print(gen)
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

Example:

```
def generator():
    i = 1
    yield "Anil"
    i = i + 1
    yield "Naga"
    i = i + 1
    yield "Bhushan"
    i = i + 1
    yield "Usha"
gen = generator()
while True:
    try:
        print(next(gen))
    except StopIteration:
        break
```

Example:

```
def generator():
    i = 1
    yield "Anil"
    i = i + 1
    yield "Naga"
    i = i + 1
    yield "Bhushan"
    i = i + 1
    yield "Usha"
for gen in generator():
    print(gen)
```

Example:

```
tp_compr = (s for s in "Python")
for p in tp_compr:
    print(p)
```

Example:

```
def calculations(limit):
    num = 0
    while num <= limit:
        yield 2 ** num
        num = num + 1
for calc in calculations(5):
    print(calc)
```

NOTE:

* Generator implementatios is simple when compared to the Iterator Implementation
* Genarators are momory effiecient
* range() is an example of generator function

```
x = range(10)
itr = iter(x)
print(next(itr))
```

Decorators:(Function inside Function):

* Decorators are used to add some extra logic/functionality to the existing code without modifying the existing function.

* The functions which contains the logic is known as "Decorator" Function.
* The logic which are going to apply to the the function, that function is called "Decorated" function.
* Decorated function object will be passed as a parameter to the decorator function whenever we call the decorated function.
* To return object of the Decorator function is going to be executed automatically.

Synatx:

```
def decorator_function(decorated_function):
    def inner():
       pass
    retrun inner

def decorated_function:
   pass
```

Example:

```
def vehicle(func):
    def inner():
        print("In vehicle - inner")
        func()
    return inner


@vehicle
def car():
    print("In car")

c = car()
```

Example:

```
def vehicle(func):
    def inner(nm, cl):
        print("In vehicle - inner")
        func(nm, cl)
    return inner

@vehicle
def car(name, color):
    print(name + ": " + color)

c = car("BMW", "Red")
```

Example:

```
def smart_divide(func):
    def inner(a,b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops!!!: can't divide")
            return None
        func(a, b)
    return inner
```

```
@smart_divide
def divide(num1, num2):
    print(num1/num2)

div = divide(20,10)
```

Day15(10-09-2022)

Closures :

* A function which is defined in another function is called as nested function or inner function.
* A function which contains another function is called as enclosing function or outer function.
* Whenever we call the outer function bydefault outer function only will be executed.
* Whenever we call inner function then we can able to called directly outside of the outer function
* Inner function can be called directly within outer function
* If we call the iner function with in the outer function then that inner function wil be executed whenever we call the outer function.

Example:

```
def vehicle(): #-----> Outer or enclosing
    print("Inside Outer function")
    def inner(): #---> Inner or nested
        print("In vehicle - inner")
    print("Exit")
    inner()  # Calling inner() function in outer function
vehicle()
```

* After returning the inner function object by the outer function, we can able to delete the outer function.
* Eventhoough we delete the outer function still we can access the inner function but we can not access the outer function.

Example:

Example:

```
def vehicle():
    print("Inside Outer function")
    def inner():
        print("In vehicle - inner")
    print("Exit")
    return inner
vhl = vehicle()
del vehicle
vhl()   # inner() function caling
vhl()
vhl()
# vehicle()
```

Closures Rules and Guidelines:

* A function which is satisfying the below rules and guidelines then we call that function as a closure.

1. Function should contain the nested function
2. The nested function must refer to a value defined in the enclosing/outer function
3. The enclosing function must return the nested function.

Example:

```
def outer(val):
    print("In Outer")
    def inner():
        print("In inner")
        print(val)
    return inner
    # inner()

out = outer(1000)
out()
# outer(2000)
```

Example:

```
def outer(val):
    print("In Outer")
    def inner():
        print("In inner")
        print(val)
    return inner
    # inner()

out = outer(1000)
del outer
out()
```

Advantages:

1. Closures can avoid the use of global varaibles and provides some sort of data hiding.
2. When there are few methods to be implemented recommended to go with closures
3. Instead of defining single class with single method better to go with Closures.

Example:

```
class Outer:
    print("In Outer")
    def __init__(self, val):
        self.val = val

    def inner(self):
        print("In inner")
        print(self.val)
out = Outer(1000)
out.inner()
```