01 Variable

#Python/Basics

- Variable is nothing but container/memory location which holds value. i.e. name given to container/memory location is Variable.
- Declare and assign value to variable -
  - In Python, we declare variable without use of data type. I.e. at any point of time in code we can create variable by assigning value to it.
  - e.g. name = “Rohit”
  - This will create string object “Rohit” and variable is ‘name’
- Valid name of python variable :
  - should start with alphabet or _
  - industry standard to create variable is → combination of alphabet, numbers and ‘_’
    - Camel Case e.g. _ageOfEmployee
    - Pascal Case e.g. _AgeOfEmployee
    - Snake Case e.g. age_of_employee
- **Assignment of values -**
  - Single values to multiple variables e.g. a=b=c=5
  - Multiple values to multiple variables e.g. a,b,c=5,10,15
- **Types of variables -**
  - **Local :**
    - variable whose scope is limited to function only and can’t be accessed outside of function
  - **Global :**
    - Global variables can be utilized all through the program, and its extension is in the whole program. Global variables can be used inside or outside the function.
    - By default, a variable declared outside of the function serves as the global variable.
    - The function treats it as a local variable if we don't use the **global** keyword.

- e.g.

```
# Declare a variable and initialize it
x = 101

# Global variable in function
def mainFunction():
    # printing a global variable
    global x
    print(x)
    # modifying a global variable
    x = 'Welcome To Javatpoint'
    print(x)

mainFunction()
print(x)
```

Output:

```
101
Welcome To Javatpoint
Welcome To Javatpoint
```

- **Nonlocal :**
  - Variable which defined in function and want to use in nested functions then we can use ‘nonlocal’ keyword for that variable
- **Delete variable -**
  - We can delete variable using ‘del’ keyword i.e. we are un-referencing that variable e.g. del x
  - post above statement ‘x’ variable is no longer use in that program and if we try to access that variable we will get error ‘name 'x' is not defined’

02 Object Identity, Reference & Identifiers

#Python/Basics

Object Identity

Object Identity

Every object created in Python has a unique identifier. Python gives the dependable that no two items will have a similar identifier. The object identifier is identified using the built-in id() function. consider about the accompanying model.

```
a = 50
b = a
print(id(a))
print(id(b))
# Reassigned variable a
a = 500
print(id(a))
```

Output:

```
140734982691168
140734982691168
2822056960944
```

We assigned the b = a, an and b both highlight a similar item. The id() function that we used to check returned the same number. We reassign a to 500; The new object identifier was then mentioned.

Object References -

- In Python, factors are a symbolic name that is a reference or pointer to an item. The factors are utilized to indicate objects by that name.
- e.g. a = 50
  - b=a
  - here, both variables are pointing/referencing same object

Identifiers -

- Identifiers are things like variables. An Identifier is utilized to recognize the literals utilized in the program.
- i.e. just like constants in java
- Valid naming convention for Identifiers -
  - should start with _
  - combination of uppercse and/or lowercase alphabets and/or numbers
  - e.g. _a123, _Days, _RateOfInterset

03 Python Data Types

#Python/Basics

Below are defined data types in python :

Numeric

- **Int**
  - e.g. a = 5
  - type() - This function returns data type of variable
    - e.g. print(type(a)) → <class 'int'>
- **Float**
  - e.g. a = 5.1
- **Complex**
  - e.g. a = 3 + 2.2J
    - a.real = 3, a.img=2.2
  - The isinstance() function returns True if the specified object is of the specified type, otherwise False.
    - e.g. isinstance(a, Complex) returning True

Sequence

- **String**
  - String means Collection of characters and immutable
  - Assignment of String variable :
    - using single quotation e.g. name = ‘Rohit’
    - using double quotes e.g. name = “Rohit”
    - using triple quotes e.g. s = '''A multiline string'''
  - String Operations:
    - Concat
      - e.g. a = ‘A’, b = ‘B’ then a+b = ‘AB’
    - Repetition
      - e.g. a = ‘A’ then a*2 = ‘AA’ (Note - here * act as repetition factor)
    - Slice
      - e.g. a = ‘Hello World’ then a[0:4] will give ‘Hell’
    - To find character at position in string
      - e.g. a = “Rohit” then a[0] will give ‘R’
    - To reverse
      - e.g. a = ‘MAR’ then a[::-1] will give ‘RAM’

- How to insert/delete character at specific position in string
  - e.g. ch = ‘R’, a = ‘AM’ then a[0]=ch will this work? - this will not work as strings are immutable
- Simple formatting of string -
  - print("The string str : **%s**" **%(str)**)
  - print(**r**'C://python37') - It is used to specify the raw string.
- Format method -
  - The **format()** method is the most flexible and useful method in formatting strings. i.e .curly braces {} are used as the placeholder in the string and replaced by the **format()** method argument.
  - e.g.

```
print("{} and {} both are the best friend".format("Devansh","Abhishek"))

print("{1} and {0} best players ".format("Virat","Rohit"))
print("{a},{b}, {c}".format(a = "James", b = "Peter", c = "Ricky"))
print(f"This is {shape_name} shape")
```

- Escape sequence char
  - ’\’ for single and double quotes in string
  - More info on different escape sequence char - e.g. ’\n’, ‘\r’, ‘\t’, ‘\v’
    More on escape sequence

- **List**
  - It is like array however it can contains data of **different types**
  - Elements are present in order
  - We can access elements using index I.e. position
  - List is mutable hence we can add/remove elements

  - Assignment of List

        - e.g. list1 = [1, "hi", "Python", 2, 3, 4, 5, 6, 7]

    - Accessing list using slice operator
        - e.g. list1[0:1] → [1, "hi"]
        - **list[start:stop:skip]** e.g. list[3:-1:2] → [2,4,6]
    - We can use concatenation using '+' and also, repetition factor '*' with list

    - Adding element to list
        - e.g. append(ele), insert(pos, ele)

    - Removing elements from list
        - e.g. pop(pos), remove(value)

    - To sort list use built in sort(reverse=True|False, function_to_specifiy_sorting_criteria)
        - Python List sort() Method
        - e.g.

```
my_list = [1,5,2,7,8,3]
my_list.sort(reverse=True)
```

```
[8, 7, 5, 3, 2, 1]
```

    - Rundown - instance of List e.g. [1,2,3]
    - **List Comprehension**
        - List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.
        - Syntax -
            -
```
newlist =
[expression for item in iterable
if condition == True]
```
        - e.g.

```
def _list_comprehension_demo():
```

```
int_list = [10, 20, 30, 40, 11, 13, 23,
   24]
odd_number_squares_list = [n * n for n in
   int_list if n % 2 != 0]
print("Input list:", int_list)
print("Output list:",
   odd_number_squares_list)
```

- output

```
Input list: [10, 20, 30, 40, 11, 13, 23, 24]
Output list: [121, 169, 529]
```

- **Advantages of List Comprehension -**
  - Easy to use as syntax is more or less similar in different operations
  - In some cases, list comprehension is efficient compared to process a list by loops and maps

- **Tuple**
  - It is like list i.e. collection of elements from different data type. However, it is immutable i.e. it is read-only data type.
  - we can't alter size or value of items in tuple
  - e.g. a = (1,"Rohit", 31)
  - we can use same operation like concatenation, slice, repetition with tuple
  - **Tuples have the following advantages over lists:**
    - take less time than lists.
    - Due to tuples, the code is protected from accidental modifications. It is desirable to store non-changing information in "tuples" instead of "records" if a program expects it.
    - A tuple can be used as a dictionary key if it contains immutable values like strings, numbers, or another tuple. **"Lists" cannot be utilized as dictionary**

keys because they are mutable.

Boolean

- True(Anything which is not zero or T) or False(0,F)

Dictionary

- A dictionary is a key-value pair set arranged in any order. It stores a specific value for each key, like an associative array or a hash table.
- Value is any Python object, while the key can hold any primitive data type.
- The comma (,) and the curly braces are used to separate the items in the dictionary.
- e.g. d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}

- To access specific value
  - using position/index
    - e.g. d[2] will give ‘Alex’
  - using get()
    - d.get(2) will give ‘Alex’

- To update value of specific key
  - using position/index
    - d[1] = ‘Sam’
  - using update - mainly useful for multiple item updates
    - d.update({1:’Sam’, 2:’Amy’})

- To access all keys or values in dictionary
  - e.g. d.keys(), d.values(),
  - d.items() - returns all the key-value pairs as a tuple

- We can delete elements of dictionary using -
  - ‘del’ keyword
    - e.g. delete d[2] → this will delete entry 2:'Alex'
  - pop() built in functions -
    - e.g. d.pop(2) → this will delete entry 2:'Alex'
  - clear() - this will remove all entries from dictionary
  - popitem() - removes the most recent key-value pair entered

- **Properties of dictionary keys**
  - We can assign different types of values to same key.However, it will only store the last one as value
    - e.g. if key is 1 and we have assigned multiple values like 1 first time, next time ‘Rohit’. It will store only ‘Rohit’ as value i.e. ({1:’Rohit’})
  - The key cannot belong to any mutable object in Python. Numbers, strings, or tuples can be used as the key, however **mutable objects like lists** cannot be used as the key in a dictionary.

Set

- The **data type's unordered collection** is Python Set. It is iterable, **mutable**(can change after creation), and has remarkable components.
- The elements of a set have no set order; It might return the element's altered sequence.
- Either a sequence of elements is passed through the curly braces and separated by a comma to create the set or the built-in function set() is used to create the set.
- **It is collection of unique values**
- It can contain different kinds of values. i.e. It can contain any type of element such as integer, float, tuple etc. **But mutable elements (list, dictionary, set) can't be a member of set.**
- e.g. set1 = set()
- To intialise set
  - set2 = {1,”Rohit”,2,"Pooja”}

- Adding element to set
  - set2.add(5)
  - set2.update([7,8])

- Removing elements from set
  - set2.remove(“Rohit”) or set2.discard(“Rohit”)
  - **main difference between remove and discard - if key is not present then python will not raise error if we are using discard.** However, it will raise error if we use remove()

        - best to use discard()
        - **pop**() - this will always **remove last element from set**.However, it is **uncertain** which element will be last at that time because **set is un-ordered collection of unique elements**

- **Python Set Operations**
    - **union** -
        - set1 = {1,2}, set2={3,4} then
            - **union() function** - set1.union(set2) → {1,2,3,4}
            - **union operator** - print(set1|set2) → {1,2,3,4}
    - **intersection** -
        - set1 = {1,2,3}, set2={3,4}, set3={3,5} then
            - **intersection() function** - set1.intersection(set2, set3) → {3}
            - **intersection operator** - print(set1&set2&set3) → {3}
    - **intersection_update** -
        - Similar as intersection.Only difference is it will modify original set. On the other hand intersection will provide new set
    - **difference** -
        - set1 = {1,2,3,4}, set2={3,4} then
            - **difference() function** - set1.difference(set2) → {1,2}
            - **difference operator** - print(set1-set2) → {1,2}
                - Need to check with more than one set
    - **symmetric_difference** -
        - set1 = {1,2,3,4}, set2={3,4,5} then
            - **symmetric_difference() function** - set1.symmetric_difference(set2) → {1,2,5}
            - **symmetric_difference operator** - print(set1^set2) → {1,2,5}
- **FrozenSet**
    - immutable version of set data type
    - One of the main advantages of using frozen set objects is that they are hashable, meaning they can be used as

keys in dictionaries or as elements of other sets.
    - Their contents cannot change, so their hash values remain constant. Standard sets are not hashable because they can be modified, so their hash values can change.
    - e.g.
        - Frozenset = frozenset([1,2,3,4,5])
    - **Frozen set to dictionary**
        - If we pass the dictionary as the sequence inside the frozenset() method, it will take only the keys from the dictionary and returns a frozenset that contains the key of the dictionary as its elements.
            - e.g.
              Dictionary = {"Name":"John", "Country":"USA", "ID":101}

Frozenset = frozenset(Dictionary)

- □ Use of relational operators with list, set, tuple and dictionary data type
- □ Explore different built in functions for list, set and dictionary data type
- □ set and frozen set with operations
- □ list and dictionary with operations

04 Keywords and Literals

#Python/Basics

Keywords

- unique words reserved with defined meanings
- https://www.javatpoint.com/python-keywords
- e.g. None, Pass, try, except, def, del, is , and, or

Literals

- Python Literals can be defined as data that is given in a variable or constant.
- e.g. a='Rohit', here 'Rohit' is literal

05 Operators

#Python/Basics

Operators

Operators refer to special symbols that perform operations on values and variables.

- ☑ **Types of operators**
- **Arithmetic Operators**
  - ‘+’, ‘-’, ‘*’, ‘/’, ‘%’
  - **Exponent** operator i.e. ‘’ e.g. a=4, b=2 then a**b = 16 (Note - having highest precedence among others)
  - **Floor Division** operator i.e. ‘//’ e.g. a=5, b=4 then a//b = 1
- **Assignment Operators**
  - ‘=’, ‘+=’, ‘-=’, ‘*=’, ‘/=’, ‘**=’, ‘//=’
- **Relational Operators / Comparison Operators**
  - ‘>’, ‘<’, ‘>=’, ‘<=’, ‘!=’, ‘==’
- **Logical Operators**
  - ‘and’, ‘or’, ‘not’ - negation of expression
  - Having least precedence among other operators while evaluation
  - e.g.
    - a=3, then expression (a>1 and a<4) is true
    - a=1, then expression (a>1 or a<4) is false
    - a=1, then expression not(a>1 or a<4) is true
- **Binary Operators**
  - ‘&’, ‘|’, ‘^’, ‘~’, ‘«’, ‘»’
- **Membership Operators**
  - ‘in’, ‘not in’
  - e.g. a=3 and b=[1,2,3] then expression a in b gives true
- **Identity Operators**
  - ‘is’, ‘is not’
  - e.g.
    - a=3 and b=[1,2,3] then expression a is b gives false
    - a=3 and b=3 then expression a is b gives true
- ☑ **Precedence of operators -**

Operator	Description
**	Overall other operators employed in the expression, the exponent operator is given precedence.
~ + -	the minus, unary plus, and negation.
* / % //	the division of the floor, the modules, the division, and the multiplication.
+ -	Binary plus, and minus
>> <<	Left shift. and right shift
&	Binary and.
^ |	Binary xor, and or
<= < > >=	Comparison operators (less than, less than equal to, greater than, greater then equal to).
<> == !=	Equality operators.
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

06 Control statements and Loops

#Python/Basics

Control Flow Statements -

If else statement

```
if expression 1:
    # block of statements

elif expression 2:
    # block of statements

elif expression 3:
    # block of statements

else:
    # block of statements
```

Loop

Loop Control Statements

- **Break statement** - break current loop's execution and return control to next statement post loop
- **Continue statement** - skip current iteration of loop i.e. all statements after continue does not gets executed in current iteration
- **Pass statement -** just to make code syntactically correct and execute nothing

Types

- **For Loop**
  - Syntax : **for** value **in** sequence: { code block }
- **While Loop**
  - Syntax : **while** Condition: Statement

07 Functions

#Python/Basics

Function

- A collection of statements that carry out a mathematical, analytical, or evaluative operation is known as a function.

- ## Advantages of functions:
    - Once defined, Python functions can be called multiple times and from any location in a program.
    - Python program gets rid of repetitive code block
    - We can get as many as return outputs from functions along with variety of arguments
    - **Disadvantage**
        - Is calling a function expensive in Python?

Function calls in Python are relatively expensive, and due to Pythons dynamic nature, it is not possible for the compiler to inline function calls.

- Is code written inline faster than using function calls?

- **Syntax of user defined functions**

```
#  An example Python Function
def function_name( parameters ):
   # code block
```

- ## Call by value vs Call by reference
    - In python all function arguments are typically **call by object reference**
        - i.e. based on object type it will execute
        - suppose we pass integer value to function then it will be call by value as locally it will copy actual value.
        - However, when we pass list as input to function then it will be call by reference. Hence, any

modifications made to that list within function can be available outside of function too

- Pass by reference vs value in Python
- Python Memory Management

- **Function arguments**
  - **Default Arguments**

```
def function( n1, n2 = 20 ):
  print("number 1 is: ", n1)
  print("number 2 is: ", n2)
```

- In above example argument n2 is having default value 20. Hence, it is Default Argument
- Function will use default value, when call is made without providing second input
- **Remember - default arguments must present after all non-default arguments**
  - Otherwise python will give error - **SyntaxError: non-default argument follows default argument**
  - ☑ Need to check in python for this

  - **Keyword Arguments**
    - Keyword arguments are linked to the arguments of a called function.
    - i.e. it is useful when we are calling function and providing value to arguments using there keywords

e.g.

```
function(n1=10, n2=50)
```

- **please note we can alter the sequence while calling function using keyword arguments e.g. function(n2=10, n1=50)**

  - **Variable-Length Arguments**
    - In Python function can receive as many as arguments hence it is called as variable-length arguments

- Note - To receive variable-length arguments we use
  - *args_list - arguments not based on keywords
  - **kargs_list - arguments based on keywords
- e.g.

```
        # Python code to demonstrate the use
        of variable-length arguments
# Defining a function
def function( *args_list ):
    ans = []
    for l in args_list:
        ans.append( l.upper() )
         return ans

# Passing args arguments

        object = function('Python',
        'Functions', 'tutorial')
print( object )

# defining a function
def function( **kargs_list ):
    ans = []
    for key, value in kargs_list.items():
        ans.append([key, value])
        return ans

# Paasing kwargs arguments
        object = function(First = "Python",
        Second = "Functions", Third =
        "Tutorial")

print(object)
```

Output

```
['PYTHON', 'FUNCTIONS', 'TUTORIAL']
```

```
[['First', 'Python'], ['Second',
        'Functions'], ['Third', 'Tutorial']]
```

- **Function return statement**
  - syntax - `return`
    `< expression to be returned as output >`
    - Single output - `return ageOfEmployee`
    - multiple output - `return ageOfEmployee, employeeId`

- **Lambda Function/ Anonymous function**
  - Function which are created without def keyword. Hence, they don't have name as well.
  - Instead **lambda** keyword use to declare Anonymous function
  - Syntax -
    - `lambda arguments: expression`
      - e.g. `lambda a,b:a+b`

  - Note -
    - lambda functions accepts any count of inputs
    - **lambda functions are limited to a single statement. Hence, expression should be one liner**

  - Usage -
    - We can use lambda function with built in API's like filter, map etc.
      - e.g.
        Lets assume we have list, my_list = [1,2,3,4,5] and we want even number list using lambda function

```
even_num_list = list( filter( lambda num: (num%2 ==
                    0), my_list )     )
```

    - We can use lambda function with list comprehension

- Details - Python Lambda

- **Built In Functions**
  - Python Built-in Functions

08 Python Arrays

#Python/Basics

- An array is **a collection of items of same data type stored at contiguous memory** locations.
- Python does not give regular array data structures like Java or C++. The one big point of difference of how arrays are implemented in Python is that they're not normal arrays, **they're dynamic arrays**. A dynamic array has the property of auto-resizing.
- Link for time complexity - Understanding Time Complexity with Simple Examples

- **How to create arrays -**
  -

```
from array import *
arrayName = array(typecode, [initializers])
```

e.g. arr = array('i',[0,1,2]), arr1 = array('f')

  - Type-code example - 'i', 'f', 'd', 'b' etc.
  - '**array**' - **module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers.**

Hence, 'array' module can't create array of strings

  - Document - array — Efficient arrays of numeric values

- **Operation we can perform with arrays -**
  - initialisation of array

```
from array import *

# creating array of integers and empty array of
    float
i_arr = array('i', [10, 20, 30])
f_arr = array('f')
```

  - traverse array -

```
# Traverse array
for i in range(len(i_arr)):
    print("arr[", i, "]", " - ", i_arr[i])
```

  - add elements to array -

```
# insert elements using append and insert
    method
# no need of index in append method as it will
    append at end position
f_arr.append(11)
print("11 is appeneded in f_arr")

# insert will need index to insert value at
    that position and adjust
# old values post insertion
f_arr.insert(0, 22)
print("22 is inserted in f_arr at 0 index")

# if index is not present then it will add at
# current last_index + 1 position
f_arr.insert(5, 33)
print("33 is inserted in f_arr at 5 index")
```

- update elements of array -

```
# update element at last position in arr1
print("f_arr - ", f_arr)
f_arr[-1] = 55
print("Post update of element at last position
    f_arr is - ", f_arr)
```

- delete element from array -

```
print("Deleting element from i_arr - ",
    i_arr[0])
del i_arr[0]
print("i_arr post deletion - ", i_arr)
```

- search element in array -

```
# search by traversing array
ele = input("enter number to search")
cnt = 0

for i in range(len(i_arr)):
    if i_arr[i] == int(ele):
        print("Number {} found at {} position
     in i_arr"
         .format(ele, i))
        cnt += 1

if cnt == 0:
    print("number {} is not present in i_arr
     ".format(ele))
```

```
# using in operator
print("Number {} is present in i_arr : "
  .format(ele), int(ele) in i_arr)
```

- Sample O/P -

```
arr[ 0 ]  -  10
arr[ 1 ]  -  20
arr[ 2 ]  -  30
11 is appeneded in f_arr
22 is inserted in f_arr at 0 index
33 is inserted in f_arr at 5 index
f_arr -  array('f', [22.0, 11.0, 33.0])
Post update of element at last position f_arr
    is -  array('f', [22.0, 11.0, 55.0])
Deleting element from i_arr -  10
i_arr post deletion -  array('i', [20, 30])
enter number to search 30
Number  30 found at 1 position in i_arr
Number  30 is present in i_arr :  True
```

09 Python OOPs

#Python/Basics

- To map real-world entity into programming world we use OOP's

Class

- Binding data(state) and method(behaviour) of an entity under one capsule i.e. encapsulation
  - **This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data**
  - We can use private variables - always start with '_ _'

        - e.g. _ _id
- Encapsulation is denoted by Class
- Class is nothing but collection of objects. In a sense, classes serve as a template to create objects.
- It is a logical entity that has some specific attributes and methods. For example: if you have an employee class, then it should contain an attributes i.e. an email id, name, age, salary, etc. and method i.e. getEmailId(), get Name(), setSalary(new_salary) etc.
- Syntax :

```
class ClassName:
    <statement-1>
    .
    .
    <statement-N>
```

Object

- An entity that has properties : **state**, identity, **behaviour** and responsibility
    - state - value of attributes
    - identity - identifying factor between two objects of same class
    - behaviour - different objects of same class should give same response to external world
        - i.e. Method in class
    - responsibility - importance of that object in that system

- It is instance of class i.e. when we create object memory is getting allocated to same
- Syntax to create object -

```
# Declare an object of a class
object_name = Class_Name(arguments)
```

_ _ init() _ _ method

- when we create object in python, specific function _ _ init() _ _ gets called automatically to initialise objects state. Thus it is often referred as ‘Constructor’
- A constructor is a special type of method (function) which is used to initialize the instance members of the class.
- It accepts the **self**-keyword as a first argument which allows accessing the attributes or method of the class.
    - '**self**' - The self-parameter refers to the current instance of the class and accesses the class variables.
    - it is like ‘this’ key word from Java
- Types of constructor -
    - Default -
        - when we don’t declare _ _ init() _ _ method in class. It will use default constructor
    - Unparameterised-
        - when we declare _ _ init() _ _ method in class with no parameters. It will be Unparameterised constructor
    - Parametrised -
        - when we declare _ _ init() _ _ method in class with parameters. It will be Parametrised constructor
- Note - Constructor overloading is not allowed in python. Hence, object of class always call last constructor defined in class

Python built in Class functions

The built-in class attributes are given in the below table.

SN	Attribute	Description
1	__dict__	It provides the dictionary containing the information about the class namespace.
2	__doc__	It contains a string which has the class documentation
3	__name__	It is used to access the class name.
4	__module__	It is used to access the module in which, this class is defined.
5	__bases__	It contains a tuple including all base classes.

Python built in Class attributes

The built-in class attributes are given in the below table.

SN	Attribute	Description
1	__dict__	It provides the dictionary containing the information about the class namespace.
2	__doc__	It contains a string which has the class documentation
3	__name__	It is used to access the class name.
4	__module__	It is used to access the module in which, this class is defined.
5	__bases__	It contains a tuple including all base classes.

E.g.

```
# This file gives demo about class and object

class Person:
    def __init__(self, pid, name=None, age=None,
gender=None):
        """constructor for person class"""
        self.pid = pid
        self.name = name
        self.age = age
        self._gender = gender

    def display_person_details(self):
        print("Person details are -")
        print("ID: ", self.pid, " Name: ",
self.name, " Age : ",
              self.age, " Gender : ", self._gender)
        print("*"*100)

# Creating Object
p1 = Person(1, "Rohit Phadtare", 31, 'M')
p2 = Person(2)

p1.display_person_details()
p2.display_person_details()
```

- # Important Key points -
  - Private members and methods -
    - we can define private members or functions by appending single quote '_' to member or function name
    - e.g.

```
class Shape:

    def __init__(self, name_of_shape):
        self._name = name_of_shape

    def __str__(self):
        return "Shape is " + self._name


my_shape = Shape("Circle")
print(my_shape)
```

  - Meaning trailing '_' after variable name is -
    - **To define temporary or unused variables**

```
for _ in range(100):
    print(_)
```

    - **The result of the last evaluation in the python interactive interpreter is stored in "_"**
    - As per PEP 515, underscores can now be added to Numeric Literals to improve the readability of long numbers.

  - dunder methods -

- Dunder- double underscore methods
- **Dunder methods are reserved methods that you can still overwrite.**
- e.g. _ _ init() _ _ , _ _ str() _ _, _ _ call() _ _

- **Double leading underscores:** -
  - In python, we can also perform data hiding by adding the double underscore (_ _) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.
  - typically used for name mangling.
  - Name mangling is a process by which the interpreter changes the attribute name to avoid naming collisions in subclasses
  - e.g. can be refer from multilevel inheritance

- Details - What's the Meaning of Single and Double Underscores In Python?

- **Method overloading -**
  - Python doesn't support method overloading by default
  - We can achieve by workaround -
    - Python | Method Overloading
    - Most efficient is By Using Multiple Dispatch Decorator
    - e.g.

```
from multipledispatch import dispatch


@dispatch(int, int)
def add_numbers(a, b):
    print("First add_numbers..")
    return a + b
```

```
@dispatch(int, int, int)
def add_numbers(a, b, c=0):
    print("Second add_numbers..")
    return a + b + c


print("Addition of two numbers 10 and 20 :
          ", add_numbers(10, 20))
print("Addition of two numbers 10, 20 and
          30 : ", add_numbers(10, 20, 30))
```

Output:

```
First add_numbers..
Addition of two numbers 10 and 20 :  30
Second add_numbers..
Addition of two numbers 10, 20 and 30 :  60
```

- **Static in python -**
  - When we declare a variable inside a class, but outside the method, it is called a **static** or class variable.
    - Properties -

      Static variable or function can be access via Classname.

      Static function can't access class attributes

      In python, object can also access static variable or static function.However, if it changes value of static variable. It will only reflect for that object. All other object still have value which is already present there at time of class creation.

      How to define static function -

        - use built in staticmethod() -this will take function as input and return static version of it
        - use @staticmethod decorator
    - e.g.

```
class Employee:
    org_name = "Wipro Ltd"

    def __init__(self, emp_id, name, sal):
        self.emp_id = emp_id
        self.name = name
        self.sal = sal

    def __str__(self):
        return f"Employee[id={self.emp_id},
       name={self.name}, " \
              f"sal={self.sal}, Company_name
       = {Employee.org_name}]"

    @staticmethod
    def get_org_name():
        return Employee.org_name


def main():
    e1 = Employee(1, "Pooja", 9000)
    e2 = Employee(2, "Rohit", 6000)
    e3 = Employee(3, "Rushi", 5000)

    for e in (e1, e2, e3):
        print(e)
        print("*" * 50)

    print(f"Updating org_name via obj for
       {e1.name}")
    e1.org_name = "ABC Steels"

    print(f"Updating org_name via class")
```

```
        Employee.org_name = "UBS"

        print("Organization name post completion
           of update")
        print("-" * 50)
        for e in (e1, e2, e3):
            print(f"[Accessing using object]
           organization "
                  f"of {e.name}: {e.org_name}")

            print(f"[Accessing using static
           method] organization "
                  f"of {e.name}:
           {e.get_org_name()}")

            print("*" * 50)

        print("[Accessing static method using
           class name] "
              "org value:",
           Employee.get_org_name())

if __name__ == "__main__":
    main()
else:
    print(__name__)
    pass
```

Output:

```
Employee[id=1, name=Pooja, sal=9000,
       Company_name = Wipro Ltd]

       **************************************
       ************
Employee[id=2, name=Rohit, sal=6000,
       Company_name = Wipro Ltd]
```

```
        ************************************
        ************
Employee[id=3, name=Rushi, sal=5000,
        Company_name = Wipro Ltd]

        ************************************
        ************
Updating org_name via obj for Pooja
Updating org_name via class
Organization name post completion of update

        ------------------------------------
        ------------
[Accessing using object] organization of
        Pooja: ABC Steels
[Accessing using static method] organization
        of Pooja: UBS

        ************************************
        ************
[Accessing using object] organization of
        Rohit: UBS
[Accessing using static method] organization
        of Rohit: UBS

        ************************************
        ************
[Accessing using object] organization of
        Rushi: UBS
[Accessing using static method] organization
        of Rushi: UBS

        ************************************
        ************
[Accessing static method using class name]
        org value: UBS
```

Inheritance

- Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch
- In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class.
- A child class can also provide its specific implementation to the functions of the parent class.
- It provides 'is-a' relationship between two classes
- Syntax

```
class derived-class(base class):
  <class-suite>
```

- **issubclass(sub,sup) method -**
  - to check the relationships between the specified classes.
  - It returns true if the first class is the subclass of the second class, and false otherwise.
- **isinstance (obj, class) method -**
  - to check the relationship between the objects and classes.
  - It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.
- **Types of inheritance -**
  - **Multilevel** -
    - e.g.

```
class Animal:

    animal_id = 0
    def __init__(self, name, id):
        print("This is animal...")
        self.__name = name
        self.animal_id = id

    def speak(self):
        print("Animal {} with id {}
```

```
            speaks ..."
                   .format(self.__name,
            self.animal_id))


class Dog(Animal):

    def __init__(self, name, id_):
        super().__init__("Dog", id_)
        print("This is Dog..")
        self.__name = name

    def speak(self):
        print("Dog with name '{}' and with id
       {} barks ..."
                   .format(self.__name,
       self.animal_id))


d = Dog("Tobo", 1)
d.speak()
print(d.__dict__)
```

Output :

```
This is animal...
This is Dog..
Dog with name 'Tobo' and with id 1 barks ...
{'_Animal__name': 'Dog', 'animal_id': 1,
       '_Dog__name': 'Tobo'}
```

- **Multiple**
  - **Python supports multiple inheritance**
  - syntax -

```
    class d(<base class 1>, <base class 2>,
    ..... <base class n>):
  <class - suite>
```

- e.g.

```
class BaseClass:
    def call_me(self):
        print("Base Class Method")

class LeftSubclass(BaseClass):
    def call_me(self):
        super().call_me()
        print("Left Subclass Method")

class RightSubclass(BaseClass):
    def call_me(self):
        super().call_me()
        print("Right Subclass Method")

class Subclass(LeftSubclass, RightSubclass):
    def call_me(self):
        super().call_me()
        print("Subclass Method")

subClass = Subclass()
print(subClass.__class__.__mro__)
subClass.call_me()
```

Output:

```
 (<class '__main__.Subclass'>,
  <class '__main__.LeftSubclass'>,
  <class '__main__.RightSubclass'>,
```

```
 <class '__main__.BaseClass'>,
 <class 'object'>)

Base Class Method
Right Subclass Method
Left Subclass Method
Subclass Method
```

- **Diamond ring problem gets resolved in python using super() function**
  - Method Resolution Order (MRO): Python follows the C3 linearization algorithm to determine the order **in which classes are searched for a method or attribute.**
  - **super() respects this order and ensures that methods are called in the correct sequence.**
  - Details - Understanding the Diamond Problem in Python

• Polymorphism -

- Different implementation of same method by different objects of same hierarchy
- Polymorphism is denoted by method overriding
- **Method overriding -**
  - When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding.
  - e.g.

```
class Person:

    def __init__(self, name_="", id_=0):
        self.name = name_
        self.id = id_

    def __str__(self):
        return "Person [ name = {} , id =
```

```
{} ]".format(self.name, self.id)

        def show(self):
            print(self)
            print("*"*50)


    class Student(Person):
        def __init__(self, name_="", id_=0,
std_=""):
            super().__init__(name_, id_)
            self.std = std_

        def __str__(self):
            return "Student [ name = {} , id = {},
std = {} ]"
                    .format(self.name, self.id,
self.std)

        def show(self):
            print(self)
            print("*"*50)


    class Employee(Person):
        def __init__(self, name_="", id_=0,
salary=0):
            super().__init__(name_, id_)
            self.salary = salary

        def __str__(self):
            return "Employee [ name = {} , id = {},
salary = {} ]"
                    .format(self.name, self.id,
self.salary)

        def show(self):
            print(self)
            print("*"*50)
```

```
p = Person("Rohit", 1)
s = Student("Rohit", 1, "12th")
e = Employee("Rohit", 1, 2000)

for x in (p, s, e):
    x.show()
```

Output:

```
    Person [ name = Rohit , id = 1 ]

**************************************************
    Student [ name = Rohit , id = 1, std = 12th ]

**************************************************
    Employee [ name = Rohit , id = 1, salary =
2000 ]

**************************************************
```

- **Abstraction**
  - Fundamental concept of OOP, which helps to hide complex details and focus on essential details
  - It involves hiding unnecessary details and exposing only the relevant information to the users. i.e. User is familiar with that **"what function does"** but they don't know **"how it does."**
  - **What is use of abstraction -**
    - an abstraction is used to hide the irrelevant data/ class in order to reduce the complexity.
  - **Ways to achieve abstraction -**
    - **Data hiding -**
      - we can use '__' double underscore as a prefix

to instance variable so that variable is not exposed to child classes. Also it is not accessible through object as well.

- **Abstract Class -**
  - **A class that consists of one or more abstract method is called the abstract class.**
  - **Abstract methods do not contain their implementation.**
  - Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass. If subclass doesn't implement abstract method then that subclass also becomes abstract one.
  - **Points to remember -**
    We can't create Object of abstract class
    Abstract class can contain normal methods as well
  - **Implementation**
    Python provides the **abc** module to use the abstraction in the Python program.
    - Syntax -

```
from abc import ABC
class ClassName(ABC):
```

    - ABC - Abstract Base Classes
      - is the common application program of the interface for a set of subclasses.
    - Working -
      - Python doesn't provide the abstract class itself. We need to import the abc module, which provides the base for defining Abstract Base classes (ABC).
      - The ABC works by

decorating methods of the base class as abstract.

- We use the *@abstractmethod* decorator to define an abstract method
- or if we don't provide the definition to the method, it automatically becomes the abstract method.

e.g.

```
# This file gives demo about abstraction

from abc import ABC


class Shape(ABC):

        def __init__(self,
           shape_name=''):
           self.shapeName = shape_name

        def draw(self):
           pass

        def show(self):
           print(f"This is
           {self.shapeName} shape")


class TriangleShape(Shape):

        def __init__(self, a=0, b=0,
           c=0):
           super().__init__("Triangle")
           self.side1, self.side2,
           self.side3 = a, b, c
```

```
        def draw(self):
            print("{}[side1={}, side2={},
            side3={}]"
                .format(self.shapeName,
            self.side1, self.side2,
            self.side3))
            print("*"*50)


class RectangleShape(Shape):

        def __init__(self, length=0,
            breadth=0):
            super().__init__("Rectangle")
            self.length_, self.breadth_ =
            length, breadth

        def draw(self, ):
            print("{}[length={},
            breadth={}"
                .format(self.shapeName,
            self.length_, self.breadth_))
            print("*"*50)


s1 = TriangleShape(10, 20, 30)
s2 = RectangleShape(15, 25)

for s in [s1, s2]:
        s.show()
        s.draw()
```

Output :

```
This is Triangle shape
Triagle[side1=10, side2=20, side3=30]
```

```
**************************************************
        This is Rectangle shape
        Rectangle[length=15, breadth=25

**************************************************
```

- **Interfaces in python -**
  - Interfaces aren't natively supported in Python, but abstract classes and methods offer a workaround.
  - Details - How do I implement interfaces in python?

- **Main function**
  - Python main() function

10 Python exception handling

#Python/Basics

Error

- Abruptly termination of program while execution of instructions
- It is irrecoverable and beyonds developer's control
- Errors are problems in a program due to which the program will stop the execution.
- Errors in python are of two types
  - **SyntaxError** - error is caused by the wrong syntax in the code.
  - **Exception** - Exceptions are raised when the program is syntactically correct, but the code results in an error.

Exception

- Unexpected behaviour occurs during execution of program
- An event which occurs during execution of program that disrupts normal flow of execution
- exceptions are raised when some internal events occur which change the normal flow of the program.
- e.g. **ZeroDivisionError**, ValueError, TypeError, NameError,

IndexError, **KeyError**, **AttributeError**, **IOError**, ImportError etc.

Exception handling

- It is process of responding to exception so that normal flow execution couldn't get disrupted
- without this process, exceptions would disrupt the normal operation of a program

- **Try and Except Statement – Catching Exceptions**
  - Try and except statements are used to catch and handle exceptions in Python.
  - In try block we will keep those line of code which might raise exception. While in except block will handle different exceptions which code from try block can raise.
  - e.g.

```
def demo():
    a = 10
    b = 0
    c = a / b
    return c


try:
    print(demo())
except ZeroDivisionError:
    print("Exception handled")
```

Output:

```
Exception handled
```

- **Else block**
  - Python provides **else** block which gets executed when try block does not raise any exceptions.

- This block must be present after all **except** clauses
- e.g.

```
def demo(a, b=0):
c = a / b
return c

try:
print(demo(10, 12))
except ZeroDivisionError:
  print("Exception handled")
else:
  print("Function Demo executed
     successfully!!")



0.8333333333333334
Function Demo executed successfully!!
```

- Note - Else block does not exist without except clauses

- **Finally block**
  - Code in this block always gets executed irrespective of exception occurred or not
  - Usually used for clean up tasks such as closing database connection, closing files etc.
  - e.g.

```
def demo(a, b=0):
    c = a / b
    return c


try:
    print(demo(10))
except ZeroDivisionError:
```

```
    print("Exception handled")
else:
    print("Function Demo executed
     successfully!!")
finally:
    print("This is finally block!!")
```

```
Exception handled
This is finally block!!
```

- **Raise built-in exception:**
  - we can use '**raise**' statement to raise any exception
  - e.g.

```
def demo(a, b=0):
    if b == 0:
        raise ValueError("Value of second
     argument should not be 0")
    else:
        c = a / b

    return c


try:
    print(demo(10))
except ZeroDivisionError:
    print("Exception handled")
except ValueError as e:
    print(e)
else:
    print("Function Demo executed
     successfully!!")
finally:
```

```
    print("This is finally block!!")
```

```
Value of second argument should not be 0
This is finally block!!
```

- **Custom exception**
  - In Python, We can create user defined exceptions deriving Exception class directly or indirectly i.e. deriving subclasses e.g. ValueError, RunTimeError etc
  - Although not mandatory, most of the exceptions are named as names that end in **"Error"** similar to the naming of the standard exceptions in python.
  - e.g.

```
class InvalidAgeError(Exception):
    def __init__(self, age_):
        self.age = age_

    def __str__(self):
        return f"Given age is {self.age}. Age
     should be >= 18"


def validateAge(age_):
    if age < 18:
        raise InvalidAgeError(age_)
    else:
        return True


try:
    age = 17
    validateAge(age)
except InvalidAgeError as e:
```

```
    print(e)
else:
    print("Age validation successful!!")
finally:
    print("Age validation process completed.")
```

```
Given age is 17. Age should be >= 18
Age validation process completed.
```

11 Python Date

#Python/Basics

Python Date

12 Python Regular Expressions

#Python/Basics

- A regular expression is a set of characters with highly specialized syntax that we can use to find or match other

characters or groups of characters.

- The re-module in Python gives full support for regular expressions of Pearl style.
    - re module raises the re.error exception whenever an error occurs while implementing or using a regular expression

- Special Characters/ Meta characters

Characters	Meaning
.	**Dot** - It matches any characters except the newline character.
^	**Caret** - It is used to match the pattern from the start of the string. (Starts With)
$	**Dollar** - It matches the end of the string before the new line character. (Ends with)
*	**Asterisk** - It matches zero or more occurrences of a pattern.
+	**Plus** - It is used when we want a pattern to match at least one.
?	**Question mark** - It matches zero or one occurrence of a pattern.
{}	**Curly Braces** - It matches the exactly specified number of occurrences of a pattern
[]	**Bracket** - It defines the set of characters
|	**Pipe** - It matches any of two defined patterns.

- Special sequences

Character	Meaning
\d	It matches any digit and is equivalent to [0-9].
\D	It matches any non-digit character and is equivalent to [^0-9].
\s	It matches any white space character and is equivalent to [\t\n\r\f\v]
\S	It matches any character except the white space character and is equivalent to [^\t\n\r\f\v]
\w	It matches any alphanumeric character and is equivalent to [a-zA-Z0-9]
\W	It matches any characters except the alphanumeric character and is equivalent to [^a-zA-Z0-9]
\A	It matches the defined pattern at the start of the string.
\b	r"\bxt" - It matches the pattern at the beginning of a word in a string. r"xt\b" - It matches the pattern at the end of a word in a string.
\B	This is the opposite of \b.
\Z	It returns a match object when the pattern is at the end of the string.

- # Implementation with RegEx Functions
  - *compile* -
    - This functions returns compiled regex object which helps us for pattern matching in many ways
  - **Note - following all functionalities we can use with compiled reg ex object which we can create using compile functions.**
  - *search*
    - this gives **first occurrence of matched pattern from input string**
    - it will return None if does not find any pattern match in input string
    - e.g.

```
import re

# create re object using compile method
pattern = "Hello"
input_str = "Hello, This is regex demo!!"

print(f"Input: {input_str}")
print(f"Search Pattern: {pattern}")

pattern_obj = re.compile(pattern)
match_obj = pattern_obj.search(input_str)
print("Is pattern matched in input string: ",
      bool(match_obj))
if bool(match_obj):
    print("Overall result", match_obj)
    print("Starting position of matched
       pattern:",
                match_obj.start())
    print("End position of matched pattern:
       ", match_obj.end())
    print("Group : ", match_obj.group())
```

```
Input: Hello, This is regex demo!!
Search Pattern: Hello
Is pattern matched in input string:  True
Overall result <re.Match object; span=(0, 5),
       match='Hello'>
Starting position of matched pattern:  0
End position of matched pattern:  5
Group :  Hello
```

- how to check first occurrence from end of string

- *re.match* -
  - same as compile and search only thing is it will **lookup pattern match at start of string**
  - re.match = compile + search
  - syntax - `re.match(pattern, input_string)`
  - e.g.

```
def matchDemo():
    pattern = "Hello"
    input_str = "Hello, This is regex
       demo !!"
    print("Result: ", re.match(pattern,
       input_str))
```

When pattern = “Hello”

```
Result:  <re.Match object; span=(0, 5),
       match='Hello'>
```

When pattern = “This”

```
Result:  None
```

- *fullmatch* - It is used to match the whole string with a regex pattern.
  - same as match only difference is it will try to match pattern with whole input
  - e.g.

```
def fullMatchDemo():
    pattern = "[a-zA-Z0-9]*@{1}[a-z]{1,5}\\.
       [a-z]{1,3}"
    input_str = "rohitphadtare39@gmail.com"
    print("Result: ", re.fullmatch(pattern,
       input_str))
```

```
Result:  <re.Match object; span=(0, 25),
      match='rohitphadtare39@gmail.com'>
```

- *findall* - It is used to find all non-overlapping patterns in a string. **It returns a list of matched patterns**.
  - e.g.

```
def findAllDemo():
    pattern = "[pP]{1}[a-z]*"
    input_str = "Pooja Rohit Phadtare"
    print("Result: ", re.findall(pattern,
       input_str))
```

```
Result:  ['Pooja', 'Phadtare']
```

- *finditer* - It returns an iterator that yields match objects.

- similar like '**findall**'. However, this gives iterator object
- e.g.

```
def findItrDemo():
    pattern = "[pP]{1}[a-z]*"
    input_str = "Pooja Rohit Phadtare"
    print("Result: ")
    cnt = 1
    for i in re.finditer(pattern, input_str):
        print(f"{cnt} match occur in input:
       {input_str} "
              f"at position "
              f"{i.start()} and group is
       {i.group()}")
        cnt += 1
```

```
Result:
1 match occur in input:Pooja Rohit Phadtare
       at position 0 and group is Pooja
2 match occur in input:Pooja Rohit Phadtare
       at position 12 and group is Phadtare
```

- *split* - It is used to split the pattern based on the regex pattern.
  - e.g.

```
def splitDemo():
    pattern = " "
    input_str = "This is split demo"
    obj = re.compile(pattern)
    print("Split result with no max splits :
       ")
    print(obj.split(input_str))
```

```
    print("Split result with no 1 splits : ")
    print(obj.split(input_str, 1))
```

```
Split result with no max splits :
['This', 'is', 'split', 'demo']
Split result with no 1 splits :
['This', 'is split demo']
```

- *sub* - It returns a string after substituting the **first occurrence** of the pattern by the replacement.
  - e.g.

```
def subDemo():
    pattern = "split"
    input_str = "This is split demo !!"
    obj = re.compile(pattern)
    print(f"Input string: {input_str}")
    print(f"Output string:
       {obj.sub('substitute', input_str)}")
```

```
Input string: This is split demo !!
Output string: This is substitute demo !!
```

- *subn* - It works the same as 'sub'. It returns a tuple (new_string, num_of_substitution).
  - e.g.

```
def subN_Demo():
    pattern = "Java"
    input_str = "This is Java REG EX demo!!
```

```
    Java is OOP language"
obj = re.compile(pattern)
print(f"Input string: {input_str}")
output = obj.subn('python', input_str)
print(f"Output: {output}")
print(f"Output string: {output[0]}")
print(f"Number of replaced instances:
    {output[1]}")
```

```
Input string: This is Java REG EX demo!! Java
    is OOP language
Output: ('This is python REG EX demo!! python
    is OOP language', 2)
Output string: This is python REG EX demo!!
    python is OOP language
Number of replaced instances: 2
```

- *escape* - It is used to escape special characters in a pattern.
  - It escapes the special character in the pattern.
  - The **escape** function become more important when the string contains regular expression **meta-characters** in it.
  - e.g.

```
def escapeDemo():
    pattern = "[RA01.*@TST"
    input_str = "[RA01.*@TST"
    pattern_with_escape = re.escape(pattern)
    print(f"Input: {input_str}")
    print(f"patten: {pattern}")
    print(f"escape patten:
        {pattern_with_escape}")
    print("Result: ",
```

```
        re.search(pattern_with_escape,
        input_str))
```

```
Input: [RA01.*@TST
patten: [RA01.*@TST
escape patten: \[RA01\.\*@TST
Result:  <re.Match object; span=(0, 11),
       match='[RA01.*@TST'>
```

- here in above example if we just use pattern without escaping its meta-characters, this code will throw an exception stating **re.error: unterminated character set at position 0**

- *purge* - purge function does not take any argument that simply clears the regular expression cache.