Python Notes - Week 3

Date:- 04/03/2022

**Revision Topics**

**Python Data Types**

1) Boolean - A boolean data type has two types of values – True or False. These values are constants and can be used to assign or compare boolean values. For example:-

| flag = True if flag == True:  print("Condition fulfilled!") else:  print("Condition not fulfilled!") |
| --- |

2) Numbers- The number data type comes in three different formats in Python- int, float, and complex.

-> Python has a built-in function type() to determine the data type of a variable or the value.

-> Another built-in function isinstance() is there for testing the type of an object.

Example:-

| var\_1 = 19 # Integer Literal  var\_2 = 15.23 # Float Literal  var\_3 = 7 + 8.54j # Complex Literal- A complex value has two parts real part and imaginary part(denoted by j)  print("The number ", var\_1, " is of type", type(var\_1)) |
| --- |

3) Strings- A sequence of one or more characters enclosed within either single quotes ‘ or double quotes ” is considered a String in Python. Any letter, a number or a symbol could be a part of the string. We can also have multi-line strings which require triple-quotes to mark the start and end of the string. Also strings in Python are immutable, it means the memory will be allocated once and re-used thereafter.

-> **String Slicing**- is about obtaining a sub-string from the given string by slicing it respectively from start to end. For performing slicing operation we make use of square brackets inside which we specify three parameters:-

a) Start Index- From where slicing should start

b) Stop Index- Where slicing should stop

c) Step- Number of elements to be skipped or ignored after slicing a particular element of the string.

Example:

| String = 'Quick Brown Fox'   print(String[:3]) # Output: Qui print(String[1:5:2]) # Output: uc print(String[-1:-12:-2])   print(String[::-1]) # Reversing a String |
| --- |

4) Lists- These are somewhat similar to arrays but are not homogeneous in nature. It can contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

a) Lists in Python can be created by just placing the sequence inside the square brackets[].

b) A list may contain duplicate values with their distinct positions.

c) In order to add elements to the list we can make use of append() method. Only one element at a time can be added to the list by using the append() method, for the addition of multiple elements with the append() method, loops are used.

d) Another way of adding elements to the list is insert() method. The difference b/w append and insert is "append()" method only works for the addition of elements at the end of the List, for the addition of elements at the desired position, "insert()" method is used. insert() method requires two arguments(position, value).

e) In order to access the list items we refer the index number. We make use of the index operator [ ] to access an item in a list.

f) There also exists a concept of "Negative Indexing" which represent positions from the end of the array. So instead of iterating through the whole list we can directly access the last or second last element by specifying list\_name[-1] or list\_name[-2]

g) Also we can make use of List Comprehension or creating new lists from other iterables like tuples, strings, arrays, lists, etc. A list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element. Example:-

| odd\_square = [x \*\* 2 for x in range(1, 11) if x % 2 == 1] print(odd\_square) |
| --- |

Lists Example:-

| list\_items = ['mango', 'grapes', 'papaya', 'watermelon', 'orange']  # Slicing operation sliced\_list = list\_items[1:4] print(sliced\_list)   sliced\_list = list\_items[2:] print(sliced\_list)   # Printing elements from start to finish sliced\_list = list\_items[:] print(sliced\_list) |
| --- |

5) Tuples- A tuple is a heterogeneous collection of Python objects which means objects of different data types can co-exist in a tuple. We can define a tuple using enclosing parentheses () having its elements separated by commas inside. But a tuple cannot be modified after it is created. We can not add or remove any element later. We can perform slicing operations in tuples just like in lists. Example:-

| first\_tuple = (13, 25, 7, 91) second\_tuple = ('Hello', 'World') res\_tuple = (first\_tuple, second\_tuple) print(res\_tuple) |
| --- |

We can also perform concatenation of two or more tuples by the use of ‘+’ operator. For Example:-

| tuple\_1 = (0, 1, 2, 3) tuple\_2 = ('New', 'Text', 'Added')   tuple\_3 = tuple\_1 + tuple\_2 print(tuple\_3) |
| --- |

6) Set- Set is an unordered collection of data type that is iterable, mutable and has no duplicate elements but unordered in nature. It can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces {}. Example:-

| str\_val = 'Hello World' set\_1 = set(String) print(set\_1) |
| --- |

-> To add elements to a set, we can make use of the built-in add() function.

-> For addition of two or more elements the Update() method is used. The update() method accepts lists, strings, tuples as well as other sets as its arguments.

7) Dictionary- Dictionary in Python is an unordered collection of data values, used to store data values in the form of key:value pairs.In Python, a Dictionary can be created by placing a sequence of elements within curly {} braces, separated by ‘comma’. Values in a dictionary can be of any data type and can be duplicated, whereas keys can’t be repeated and must be immutable. Example:

| dict\_items = dict({1: 'New Text', 2: 'For', 3:'Dictionary'}) # Creating Dictionary with dict() method print(dict\_items) |
| --- |

-> Addition of elements in a dictionary can be done by defining value along with the key e.g. dict[Key] = ‘Value’.

-> Updating an existing value in a Dictionary can be done by using the built-in update() method.

Example:-

| dict\_items = {}  dict\_items[0] = 'Geeks' dict\_items[2] = 'For' dict\_items[3] = 1 print(dict\_items) |
| --- |

-> In order to access the items of a dictionary we refer to the key name which can be used inside square brackets.

-> Some common dictionary functions to access elements include:-

a) **keys()** – It isolates the keys from a dictionary.

b) **values()** – It isolates the values from a dictionary.

c) **items()** – It returns the items in a list style of (key, value) pairs.

**OOPS Concepts**

1) Classes and Objects- Class is a user-defined data structure, which has its own data members and member functions, which can be accessed and used by creating an instance of that class. Some points on Python class include:

-> Classes are created by keyword class.

-> Attributes are the variables that belong to a class.

-> Attributes are public by default and can be accessed by the class instance/object using the dot (.) operator.

a) **self keyword**- This keyword represents the instance or refers to the current object of a class. It helps in accessing the class members such as attributes from the class methods. Whenever we create a method inside a class, this is the first parameter in the method definition. We do not give a value for this parameter when we call the method.

b) **\_\_init\_\_ method**- This is the constructor method which runs as soon as an object of a class is instantiated. Like methods, a constructor also contains a collection of statements that are executed at the time of Object creation. The task of constructors is to initialize(assign values) to the data members of the class when an object of the class is created.

Types of constructors :

-> Default constructor: The default constructor is a simple constructor which doesn’t have any arguments except one argument which is a reference to the instance being constructed(self keyword).

-> Parameterized constructor: A constructor with parameters is known as parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as self and the rest of the arguments are provided by the programmer.

Example:-

| class personName:   def \_\_init\_\_(self, name): # constructor method (Parameterized Constructor)  self.name = name    def display\_name(self):  print('Hello, my name is', self.name)   pn = personName('Amit') pn.display\_name() |
| --- |

2) Inheritance- Inheritance is the capability of one class to derive or inherit the properties from another class. The biggest advantage of using this concept is that It provides reusability of a code. We don’t have to write the same code again and again.

Types of inheritance:-

a) **Single inheritance**: When a child class inherits from only one parent class, it is called single inheritance. Example:-

| class personName:  def \_\_init\_\_(self, n = 'Rahul'):  self.name = n   class personNo(personName):  def \_\_init\_\_(self, roll):  personName.\_\_init\_\_(self, "Ashwini") #   self.roll = roll   obj\_1 = personRoll(19) print (obj\_1.name) |
| --- |

b) **Multiple inheritance**: When a child class inherits from multiple parent classes, it is called multiple inheritance. Example:-

| class Father(object):  def \_\_init\_\_(self):  self.str\_1 = "father"  print("Father Class")   class Mother(object):  def \_\_init\_\_(self):  self.str\_2 = "mother"   print("Mother Class")   class Child(Father, Mother):  def \_\_init\_\_(self):    # Calling constructors of Father and Mother classes  Father.\_\_init\_\_(self)  Mother.\_\_init\_\_(self)  print("Derived Child Class")    def printStrs(self):  print(self.str\_1, self.str\_2)     ob = Child() ob.printStrs() |
| --- |

c) **Hierarchical inheritance**: More than one derived class is created from a single base class.

d) **Multilevel inheritance**: When we have a child and grandfather relationship. Example:-

| class GrandFather(object):  def \_\_init\_\_(self, name):  self.name = name    def getName(self):  return self.name   class Father(GrandFather):  def \_\_init\_\_(self, name, age):  GrandFather.\_\_init\_\_(self, name)  self.age = age    def getAge(self):  return self.age   class GrandChild(Father):  def \_\_init\_\_(self, name, age, address):  Father.\_\_init\_\_(self, name, age)  self.address = address   def getAddress(self):  return self.address    gc = GrandChild("Sameer", 23, "Gurgaon")  print(gc.getName(), gc.getAge(), gc.getAddress()) |
| --- |

3) Encapsulation- It is the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To provide security to our data, we can make our variables:

a) **Private**- The class members declared private cannot be accessed outside the class nor by any base class. To define a private member prefix the member name with double underscore “\_\_”.

b) **Protected**- Protected members are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses. To accomplish this in Python, we prefix the name of the member by a single underscore “\_”.