BHAGWAN MAHAVIR UNIVERSITY
BHAGWAN MAHAVIR POLYTECHNIC COMPUTER ENGINEERING DEPARTMENT
THEORY NOTES

Introduction to Python Programming-Theory (1030106502)

Chapter 6: OOPs Concepts and File handling

OOPs Concepts:

Features of OOPs:

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

Creating class and object:

Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
    # Statement-1
    . . . .
    # Statement-N
```

Objects

The object is an entity that has a state and behaviour associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects.

Instance Attribute

- An instance attribute is a Python variable belonging to one, and only one, object. This variable is only accessible in the scope of this object and it is defined inside the constructor function, __init__(self,..) of the class.

Class Attribute

- A class attribute is a Python variable that belongs to a class rather than a particular object. It is shared between all the objects of this class and it is defined outside the constructor function, __init__(self,...), of the class.

Creating a class and object with class and instance attributes

Program:

```
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

Output:

```
Rodger is a mammal
Tommy is also a mammal
My name is Rodger
My name is Tommy
```

Inheritance:

Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class.

The benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance:

Single Inheritance:

Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.

Multilevel Inheritance:

Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

Hierarchical Inheritance:

Hierarchical level inheritance enables more than one derived class to inherit properties from a parent class.

Multiple Inheritance:

Multiple level inheritance enables one derived class to inherit properties from more than one base class.

Program:

```
# parent class
class Person(object):

    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display(self):
        print(self.name)
        print(self.idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post
```

```
        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))


# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using
# its instance
a.display()
a.details()
```

Output:

```
Rahul
886012
My name is Rahul
IdNumber: 886012
Post: Intern
```

Constructor and Destructor:

Python Constructor is the special function that is automatically executed when an object of a class is created. Python __init__ function is to act as a Constructor.

Syntax:

```
def __init__(self, [args ……..]):

<statements>
```

Example:

```
class Sample:

    def __init__(self, num):
        print("Constructor of class Sample...")
        self.num = num
        print("The value is :", num)


S = Sample(100)
```

Output:

Constructor of class Sample...
The value is : 100

Python Destructor is also a special method that gets executed automatically when an object exit from the scope. In Python, __del__( ) method is used as the destructor.

Program:

```
class Student:

    # constructor
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('Object initialized')

    def show(self):
        print('Hello, my name is', self.name)

    # destructor
    def __del__(self):
        print('Inside destructor')
        print('Object destroyed')

# create object
s1 = Student('Emma')
s1.show()

# delete object
del s1
```

Output:

Inside Constructor
Object initialized
Hello, my name is Emma
Inside destructor
Object destroyed

Polymerphism:

Polymorphism defines the ability to take different forms. Polymorphism in Python allows us to define methods in the child class with the same name as defined in their parent class.

Polymorphism is taken from the Greek words Poly (many) and morphism (forms). It means that the same function name can be used for different types. This makes programming more intuitive and easier.

Polymorphism with Function and Objects

Example:

```
class Tomato():
    def type(self):
      print("Vegetable")
    def color(self):
      print("Red")
class Apple():
    def type(self):
      print("Fruit")
    def color(self):
      print("Red")

def func(obj):
      obj.type()
      obj.color()

obj_tomato = Tomato()
obj_apple = Apple()
func(obj_tomato)
func(obj_apple)
```

Output:

```
Vegetable
Red
Fruit
Red
```

Method overloading:

Overloading is the ability of a function or an operator to behave in different ways based on the parameters that are passed to the function, or the operands that the operator acts on.

In Python, you can create a method that can be called in different ways. So, you can have a method that has zero, one or more number of parameters. Depending on the method definition, we can call it with zero, one or more arguments.

Given a single method or function, the number of parameters can be specified by you. This process of calling the same method in different ways is called method overloading.

Example:

```
def product(a, b):
    p = a * b
    print(p)
def product(a, b, c):
    p = a * b*c
    print(p)
product(4, 5, 5)
```

Output:

100

Method Overriding:

Method overriding is an ability of any object-oriented programming language that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to **override** the method in the super-class.

Example:

```
class Parent():
    def __init__(self):
        self.value = "Inside Parent"
    def show(self):
        print(self.value)
class Child(Parent):
    def __init__(self):
        self.value = "Inside Child"
    def show(self):
        print(self.value)
obj1 = Parent()
obj2 = Child()
obj1.show()
obj2.show()
```

Output:

Inside Parent
Inside Child

Files:

Types of Files in Python

1. Text File
2. Binary File

1. Text File

- Text file store the data in the form of characters.
- Text file are used to store characters or strings.
- Usually we can use text files to store character data
- eg: abc.txt

2. Binary File

- Binary file store entire data in the form of bytes.
- Binary file can be used to store text, image, audio and video.
- Usually we can use binary files to store binary data like images,video files, audio files etc

Opening and Closing a File:

- Before performing any operation (like read or write) on the file, first we have to open that file. For this we should use Python's inbuilt function open()
- But at the time of open, we have to specify mode, which represents the purpose of opening file.
- We should use open() function to open a file. This function accepts 'filename' and 'open mode' in which to open the file.
- File_object = open("File_Name","Access_Mode")

The File Opening Mode

w

- To write data into file. If any data is already present in the file, it would be deleted and the present data will be stored.
- open an existing file for write operation. If the file already contains some data then it will be overridden. If the specified file is not already available then this mode will create that file.

r

- To read the data form the file. The file pointer is positioned at the beginning of the file.
- open an existing file for read operation. The file pointer is positioned at the beginning of the file.If the specified file does not exist then we will get FileNotFoundError. This is default mode.

a

- To append data to the file. Appending means adding at the end of existing data. The file pointer is placed at the end of the file. If the file does not exist, it will create new for writing data.
- open an existing file for append operation. It won't override existing data. If the specified file is not already avaialble then this mode will create a new file.

w+

- To write and read data a file. The previous data in the file will be deleted.
- To write and read data. It will override existing data.

r+

- To read and write data into the file. The previous data in the file will not be deleted.The file pointer is placed at the beginning of the file.

a+

- To append and read data from the file.It wont override existing data.
- To append and read of a file. The file pointer will be at the end of the file if the file exists. If the file does not exist, it creates a new file for reading and writing.

x

- To open a file in exclusive creation mode for write operation. If the file already exists then we will get FileExistsError.

Note: All the above modes are applicable for text files. If the above modes suffixed with 'b' then these represents for binary files.

- Eg: rb,wb,ab,r+b,w+b,a+b,xb
- **f = open("abc.txt","w")**
- **f.close()**
- close() function closes the file and frees the memory space acquired by that file.

  File_object.close()

We are opening abc.txt file for writing data.

Writing a File:

There are two ways to write in a file.

1. **write() :** Inserts the string str1 in a single line in the text file.

File_object.write(str1)

2. **writelines() :** For a list of string elements, each string is inserted in the text file.Used to insert multiple strings at a single time.

File_object.writelines(L) for L = [str1, str2, str3]

Reading a File:

There are three ways to read data from a text file.

1. **read() :** Returns the read bytes in form of a string. Reads n bytes, if no n specified, reads the entire file.

   File_object.read([n])

2. **readline() :** Reads a line of the file and returns in form of a string.For specified n, reads at most n bytes. However, does not reads more than one line, even if n exceeds the length of the line.

   File_object.readline([n])

3. **readlines() :** Reads all the lines and return them as each line a string element in a list.

   File_object.readlines()

Renaming Files:

The rename() method takes two arguments, the current filename and the new filename.

Syntax:

Import os

os.rename(current_file_name, new_file_name)

Deleting Files:

You can use the remove() method to delete files by supplying the name of the file to be deleted as the argument.

Syntax:

Import os

os.remove(file_name)

With statement:

with statement in Python is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Observe the following code example on how the use of with statement makes code cleaner.

Example:

```
#without using with statement
file = open('file_path', 'w')
file.write('hello world !')
file.close()
# using with statement
with open('file_path', 'w') as file:
    file.write('hello world !')
```

seek() method

In Python, seek() function is used to **change the position of the File Handle** to a given specific position. File handle is like a cursor, which defines from where the data has to be read or written in the file.

Syntax:

file.seek(*offset*)

tell() method

The tell() method returns the current file position in a file stream.

Syntax:

file.tell()