PYTHON PROGRAMMING

BCA IV SEM(NEP)

NOTES

Prepared By

Mr. Prabhu Kichadi, BE, MTECH

9880437187

UNIT – IV

File Handling: File Types; Operations on Files– Create, Open, Read, Write, Close Files; File Names and Paths; Format Operator.

Object Oriented Programming: Classes and Objects; Creating Classes and Objects; Constructor Method; Classes with Multiple Objects; Objects as Arguments; Objects as Return Values; Inheritance- Single and Multiple Inheritance, Multilevel and Multipath Inheritance; Encapsulation- Definition, Private Instance Variables; Polymorphism- Definition, Operator Overloading.

File Handling: File handling is the process of creating, manipulating, and reading files in a computer program. In Python, you can perform file handling operations using built-in functions and modules.

Files: File is a collection of related data stored on a secondary storage.
File extensions: .txt, .docx, .pdf, .c, .cpp etc

Python has several functions for creating, reading, updating, and deleting files.

open() function: This function is used to open a file and returns a file object. It takes two arguments: the file name (with path, if necessary) and the mode in which the file is to be opened (e.g. read, write, append, etc.).

Reading a file: You can read the contents of a file using the read() method of the file object. You can also read one line at a time using the readline() method, or read all the lines at once using the readlines() method.

Writing to a file: You can write to a file using the write() method of the file object. If the file does not exist, it will be created. If it already exists, its contents will be overwritten unless you open it in append mode.

Closing a file: After you are done working with a file, you should close it using the close() method of the file object. This ensures that all the data that you have written to the file is saved.

File Types:

File is of two types to perform:

Type	Description
t	**Text - Default value. Text mode**
b	**Binary - Binary mode (e.g. images)**

Operations on Files– Create, Open, Read, Write, Close Files;

Modes of access of files:

Mode	**Description**
"r"	Read - Default value. Opens a file for reading, error if the file does not exist
"w"	Write - Opens a file for writing, creates the file if it does not exist
"a"	Append - Opens a file for appending, creates the file if it does not exist
"x"	Create - Creates the specified file, returns an error if the file exists

File Operations:

1. **Open a file – open()**
2. **Read or write (perform operation): read(), readline() and write()**
3. **Close the file: close()**
4. **Delete a File**

1. Open a File:

Before you can read or write data to a file, you need to open it first. You can use the built-in open() function to open a file in different modes, such as read-only, write-only, or read-write. Here's an example of how to open a file in read mode:
open():

syntax :

```
open(filename, mode)
```

Ex:

```
f = open("sample.txt")
    or
f = open("sample.txt", "r")
   OR
f = open("sample.txt", "rt")
```

2. Reading from a File:

- **read():** This method reads the entire contents of a file as a string. Here's an example:

Ex:

```
file = open('example.txt', 'r')
contents = file.read()
print(contents)
```

Ex2:

read() method with number of characters:

```
f = open("input.txt", "r")
str = f.read(6)
print(str)
```

- **readline(): This method reads one line at a time from a file. Here's an example:**

```
file = open('example.txt', 'r')
line = file.readline()
while line:
    print(line)
    line = file.readline()
```

3. Writing to a file:

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file, creates a file if not exists.
"w" - Write - will overwrite any existing content, creates a file if not exists.
“x” – Exclusive Create: - Creates a file if not exists. Failes if file already exists, FileExistsError

- **write(): This method writes a string to a file. Here's an example:**

Ex:

```
file = open('example.txt', 'w')
file.write('This is some text.')
```

Ex:

```
f = open("output.txt", "w")
f.write("This is New Data to written")
f.close()
```

- But using “w” mode overwrites the data

Ex:

```
f = open("output.txt", "a")
f.write("This is New Data to written")
f.close()
```

4. Closing a File: It is a good practice to always close the file when you are done with it. Using close() method.

```
f = open("input.txt", “r”)
str = f.readline()
print(str)
f.close()
```

4. Delete a File : To delete a file, you must import the OS module, and run its os.remove() function:

Ex1:

File Names and Paths;

Format Operator.

Object Oriented Programming: Object-oriented programming (OOP) is a programming paradigm that is based on the concept of objects.

- In OOP, everything is considered as an object that has some properties (attributes) and can perform certain actions (methods).
- It provides a way to organize and structure code in a way that is more modular, maintainable, and reusable.
- In Python, everything is an object, and you can create your own custom objects by defining classes.
- A class is a blueprint for creating objects that defines the properties and methods that the objects will have. Once you define a class, you can create multiple objects (instances) from it.

Classes and Objects & Creating Classes and Objects:

A class is a blueprint for creating objects that have specific attributes and methods.
Class is a collection of data members(instance variables+constructors+methods).

Class creation syntax:

In Python, you can create a class using the class keyword followed by the name of the class and a colon. The body of the class is indented and contains attributes and methods.

Here's the syntax for creating a class in Python:

```
class ClassName:
  def __init__(self, arg1, arg2):
    # constructor method
    self.attr1 = arg1
    self.attr2 = arg2

  def method1(self, arg3):
    # method
    self.attr1 += arg3
```

In this example, we created a class called ClassName with a constructor method (__init__) that takes two arguments (arg1 and arg2) and sets them as attributes (attr1 and attr2) of the object. The class also has a method called method1 that takes one argument (arg3) and modifies the attr1 attribute.

Objects are instances of a class, and they can have different values for their attributes while sharing the same methods.
Object is anything that has physical existence, state and behavior.

Object creation syntax:

```
obj = ClassName(val1, val2)
```

Constructor Method: the constructor method is a special method that is called when an object is created from a class. The constructor method is named __init__ and it is used to initialize the attributes of the object.

Ex:

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

  def say_hello(self):
       print(f"Hello, my name is {self.name} and I'm {self.age} years old.")
```

The constructor method in Python has the following features:

- ✓ It is a special method that is called automatically when an object is created from a class.
- ✓ It has the name __init__ and takes at least one parameter (self) which refers to the object being created.
- ✓ It is used to initialize the attributes of the object.
- ✓ It can take any number of parameters after self to initialize the attributes of the object.
- ✓ It can also contain default parameter values for its arguments.
- ✓ It can perform any necessary setup for the object before it is used.

In Python, a class can have only one constructor, which can be either a default constructor (with no parameters) or a parameterized constructor (with one or more parameters).

1. Default constructor

The default constructor is created automatically if no constructor is defined in the class. It does not take any parameters, and its body is empty. The purpose of the default constructor is to create an object of the class with default values.

```
class MyClass:
        def __init__(self):
                pass
```

2. Parameterized constructor

A parameterized constructor takes one or more parameters in addition to the self-parameter. It is used to initialize the attributes of the object with the values passed as arguments to the constructor.

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

self: self is a special parameter in Python that refers to the current instance of the class. It is the first parameter of every instance method in a class, including the constructor method (__init__). When you create an object of a class, the self parameter is automatically passed to the constructor method and to every instance method of the object.

The self parameter is used to access the attributes and methods of the current instance of the class. You can use the self parameter to set or get the values of the object's attributes or to call its methods.

Ex:

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

        def print_info(self):
           print("Name: ",self.name)
           print("Age: ",self.age)
# create two Person objects
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

# print the info of each Person object
person1.print_info()  # Name: Alice, Age: 25
person2.print_info()  # Name: Bob, Age: 30
```

Classes with Multiple Objects; In Python, you can create multiple objects of a class, each with its own set of attributes and methods. To create multiple objects of a class, you can call the constructor method of the class multiple times with different arguments.

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

   def print_info(self):
      print("Name:",self.name)
      print("Age:",self.age)
# create two Person objects
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

# print the info of each Person object
person1.print_info()  # Name: Alice, Age: 25
person2.print_info()  # Name: Bob, Age: 30
```

Instance variables & class variables:

Instance variables are created within the __init__ method of a class, and are typically prefixed with self. to indicate that they belong to the instance of the class.

For example:

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

In this example, the Person class has two instance variables: name and age. These variables are unique to each instance of the Person class, so each person object created from this class will have their own name and age values.

Class variables, on the other hand, are defined at the class level and are shared by all instances of the class. They are typically prefixed with the class name to indicate that they are class variables.

For example:

```
class Person:
        count = 0
                def __init__(self, name, age):
                 self.name = name
                self.age = age
                 Person.count += 1
```

In this example, the Person class has a class variable count, which is incremented each time a new person object is created. This variable is shared by all instances of the Person class, so if you create multiple Person objects, they will all share the same value of count.

Objects as Arguments: you can pass objects as arguments to functions just like any other data type. When an object is passed as an argument to a function, the function receives a reference to the object, not a copy of the object itself.

This means that any changes made to the object within the function will also affect the object outside the function.

Ex:

```
class Rectangle:
  def __init__(self, length, width):
     self.length = length
```

```
        self.width = width

def calculate_area(rect):
    area = rect.length * rect.width
    return area

# create a rectangle object
rect = Rectangle(5, 3)

# pass the rectangle object to the function
area = calculate_area(rect)

# print the area
print(area)  # Output: 15
```

Objects as Return Values: you can return objects from functions just like any other data type. When a function returns an object, it returns a reference to the object, not a copy of the object itself. This means that any changes made to the object within the function will also affect the object outside the function.

Ex:

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

def create_rectangle(length, width):
    rect = Rectangle(length, width)
    return rect

# create a rectangle object using the function
rect = create_rectangle(5, 3)

# print the length and width of the rectangle object
print(rect.length)  # Output: 5
print(rect.width)  # Output: 3
```

In this example, we define a Rectangle class with two instance variables: length and width. We then define a function called create_rectangle that takes a length and a width as arguments, creates a Rectangle object with the specified length and width, and returns the object. Finally, we call the create_rectangle function to create a Rectangle object, and print its length and width.

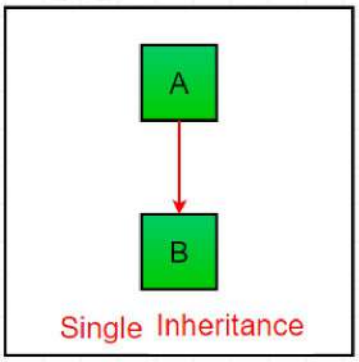
Inheritance: the process of acquiring properties from one class to another class.

- Inheritance is a fundamental concept in object-oriented programming that allows you to define a new class based on an existing class.
- The new class(child) inherits all the properties and methods of the existing(parent) class and can also have its own properties and methods.
- In Python, you can create a new class that inherits from an existing class by specifying the existing class in parentheses after the new class name, like this:

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

  def say_hello(self):
    print("Hello, my name is", self.name)

class ChildClass(ParentClass):
      pass
```

Types of inheritance:

1. Single inheritance: When a class inherits from a single parent class, it is called single inheritance.

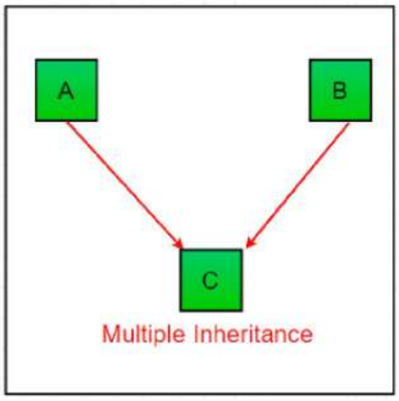


Ex:

```
class Parent:
  def parent_method(self):
        print("This is the parent method")

class Child(Parent):
   def child_method(self):
          print("This is the child method")
```

2. Multiple Inheritance:

When a class inherits from more than one parent class, it is called multiple inheritance.

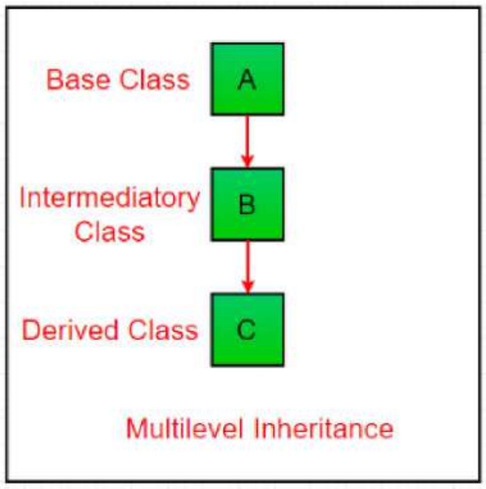


Ex:

```
class Parent1:
        def parent1_method(self):
                print("This is the parent1 method")

class Parent2:
         def parent2_method(self):
                 print("This is the parent2 method")

class Child(Parent1, Parent2):
         def child_method(self):
                 print("This is the child method")
```

3. Multilevel Inheritance:

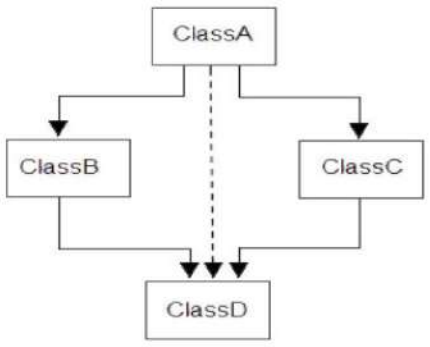
When a class inherits from a parent class, which in turn inherits from another parent class, it is called multilevel inheritance.



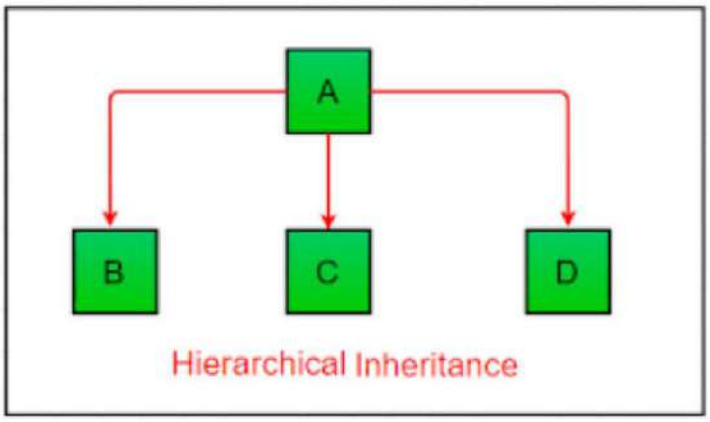
Ex:

```
class Grandparent:
        def grandparent_method(self):
                print("This is the grandparent method")

class Parent(Grandparent):
        def parent_method(self):
                print("This is the parent method")

class Child(Parent):
        def child_method(self):
                print("This is the child method")
```

4. Multipath Inheritance: Multipath inheritance in Python is a type of inheritance where a class inherits from two or more classes that have a common base class. In other words, there is more than one path from the derived class to the base class.



5. Hierarchical Inheritance: Hierarchical inheritance in Python is a type of inheritance where a single base class is inherited by multiple derived classes. Each derived class becomes a base class for its own set of derived classes, creating a hierarchical structure of inheritance.

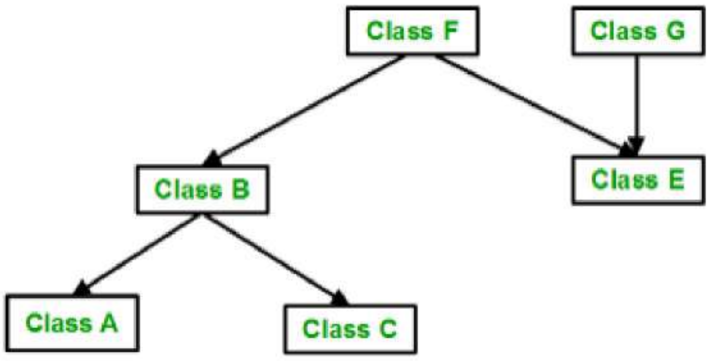


Ex:

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

    def eat(self):
        print(self.name, "is eating")

class Dog(Animal):
    def bark(self):
        print(self.name, "is barking")

class Cat(Animal):
    def meow(self):
        print(self.name, "is meowing")

class Tiger(Cat):
    def roar(self):
        print(self.name, "is roaring")
```

6. Hybrid Inheritance: Python also supports hybrid inheritance, where a combination of single, multiple, and multilevel inheritance is used.

In hybrid inheritance, there can be multiple inheritance paths to a class and the same class can be inherited at different levels in the class hierarchy.



Ex:

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

        def eat(self):
                print(self.name, "is eating")

class Mammal(Animal):
        def nurse(self):
                print(self.name, "is nursing")

class Bird(Animal):
        def fly(self):
                print(self.name, "is flying")

class Bat(Mammal, Bird):
        def __init__(self, name):
                super().__init__(name)

        def echolocate(self):
                print(self.name, "is echolocating")
```

Encapsulation- Definition: Encapsulation in Python is a mechanism that allows data and behavior to be wrapped up together in a single unit, known as a class. The purpose of encapsulation is to protect the internal state of an object from outside interference and to provide a controlled way for other objects to interact with it.In Python, encapsulation is achieved through the use of access modifiers, which limit the visibility of data and methods within a class.

There are two types of access modifiers in Python:

Public: Public members are accessible from outside the class and are denoted by using no prefix or using a single underscore prefix, such as _public_method().

Private: Private members are not accessible from outside the class and are denoted by using a double underscore prefix, such as __private_method().

Private Instance Variables: In Python, private instance variables are variables that are intended to be used within the class and not directly accessed or modified from outside the class.

Private instance variables are defined using a double underscore prefix (e.g. __variable_name) in the class definition.

Ex:

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age > 0:
            self.__age = age
```

```
person = Person("John", 30)

# Accessing private instance variables using name mangling
person._Person__age = 40
print(person.get_age())
```

Polymorphism- Definition: method overriding:

Polymorphism is a concept in object-oriented programming that allows objects of different classes to be treated as if they are objects of the same class. It means that different objects can have the same method or attribute name, but they can behave differently depending on the class they belong to.

In Python, polymorphism is achieved through **method overriding and method overloading.**

Method overriding is the process of redefining a method in a subclass that is already defined in its superclass. This allows the subclass to provide its own implementation of the method and override the behavior of the superclass.

Here's an example:

```
class Animal:
   def speak(self):
      print("The animal makes a sound")

class Dog(Animal):
   def speak(self):
      print("The dog barks")

class Cat(Animal):
   def speak(self):
      print("The cat meows")

a = Animal()
a.speak()
```

```
d = Dog()
d.speak()

c = Cat()
c.speak()
```

Operator Overloading:

Note: Refer all python Lab journal programs on this unit.