4.Object Oriented programming

Subject : Programming with Python (IT3008)

Faculty: Ms. Twinkle Kosambia

Assistant Professor, Computer Science Department,

Asha M. Tarsadia Institute of Computer Science

Website: utu.ac.in

Syllabus of Programming with Python:

https://app.utu.ac.in/utuformaccess/utusyllabus.aspx?CF=7&CM=177&SY=1

Programming with Python

Subject code : IT3008

Credits : 5

Theory marks : 60+ 40= 100

Practical marks : 100 (CIE)

Reference books

Text book:

1. Allex Martelli, Anna Ravenscroft and Steve Holden, “Python in Nutshell”, 3rd Edition, O’Reilly Publication.

Reference books:

1. Magnus Lie Hetland, "Beginning Python From Novice to Professional", Third Edition, Apress, 2017.

2. David Beazley, Brian K. Jones, "Python Cookbook", 3rd edition, O’Reilly Publication, 2016.

3. Brett Slatkin, "Effective Python: 59 Specific Ways to Write Better Python", Novatec, 2016.

4. Mark Lutz "Learning Python", 4th Edition, O’Reilly Publication, 2016.

Course outcomes

Unit No.	Unit Name	Course Outcomes					
		CO1	CO2	CO3	CO4	CO5	CO6
1	Introduction to Python	✓					
2	Data Structures		✓				
3	Control Structure and Functions			✓			
4	Object Oriented Programming				✓		
5	Exception Handling and Regular Expression					✓	
6	File Handling						✓

Unit 4 : Object Oriented programming

Object Oriented programming

The object oriented paradigm, Scopes and Namespaces, Class definition, Class objects, Instance objects, Method objects, Class and Instance variables, Inheritance, Private variables, Polymorphism - Method overloading and method overriding, Data hiding, Decorators, Metaclass, Multithreading, Using properties to control attribute access, Creating complete fully integrated data types.

Contents

- Differences Procedure vs Object Oriented Programming
- Features of OOP
- Fundamental Concepts of OOP in Python
- What is Class
- What is Object
- Methods in Classes
- Instantiating objects with ___init_____self
- Encapsulation
- Data Abstraction
- Public, Protected and Private Data
- Inheritance
- Polymorphism
- Operator Overloading

Difference between Procedure Oriented and Object Oriented Programming

- Procedural programming creates a step-by-step program that guides the application through a sequence of instructions. Each instruction is executed in order.
- Procedural programming also focuses on the idea that all algorithms are executed with functions and data that the programmer has access to and can change.
- Object-oriented programming is much more similar to the way the real world works; it is analogous to the human brain. Each program is made up of many entities called objects.
- Instead, a message must be sent requesting the data, just like people must ask one another for information; we cannot see inside each other's heads.

Summarized the differences

Procedure Oriented Programming	Object Oriented Progrmming
In POP, program is divided into small parts called **functions**	In OOP, program is divided into parts called **objects**.
POP follows **Top Down approach**	OOP follows **Bottom Up approach**
POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**
In POP, Overloading is not possible	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading
In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data.
POP does not have any access specifier	OOP has access specifiers named Public, Private, Protected, etc.

Features of OOP

- Ability to simulate real-world events much more effectively
- Code is reusable thus less code may have to be written
- Data becomes active
- Better able to create GUI (graphical user interface) applications
- Programmers can produce faster, more accurate, and better-written applications

Fundamental concepts of OOP in Python

The four major principles of object orientation are:

- Encapsulation
- Data Abstraction
- Inheritance
- Polymorphism

What is an Object..?

- Objects are the basic run-time entities in an object-oriented System.
- They may represent a person, a place, a bank account, a table of data, or any item that the program must handle.
- When a program is executed the objects interact by sending messages to one another.
- Objects have two components:

- Data (i.e., attributes)
- Behaviors (i.e., methods)

An object has the following two characteristics:

- Attribute
- Behavior

For example, **A Car is an object**, as it has the following properties:

name, price, color as **attributes**

breaking, acceleration as **behavior**

Object Attributes and Methods Example

Object Attributes

- Store the data for that object
- Example (taxi):
  - Driver
  - OnDuty
  - NumPassengers
  - Location

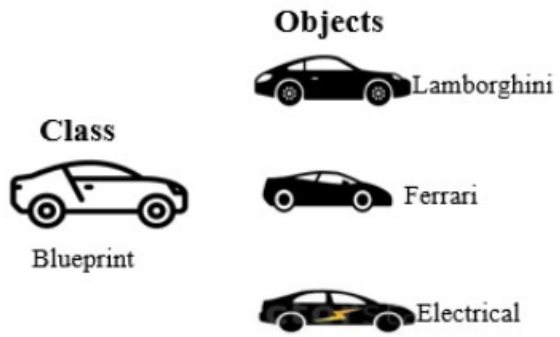
Object Methods

- Define the behaviors for the object
- Example (taxi):
  - PickUp
  - DropOff
  - GoOnDuty
  - GoOffDuty
  - GetDriver
  - SetDriver
  - GetNumPassengers

What is a Class..?

- A class is a blueprint for the object. To create an object we require a model or plan or blueprint which is nothing but class.
- For example, you are creating a vehicle according to the Vehicle blueprint (template). The plan contains all dimensions and structure. Based on these descriptions, we can construct a car, truck, bus, or any vehicle. Here, a car, truck, bus are objects of Vehicle class

- A class contains the properties (attribute) and action (behavior) of the object. Properties represent variables, and the methods represent actions. Hence class includes both variables and methods.



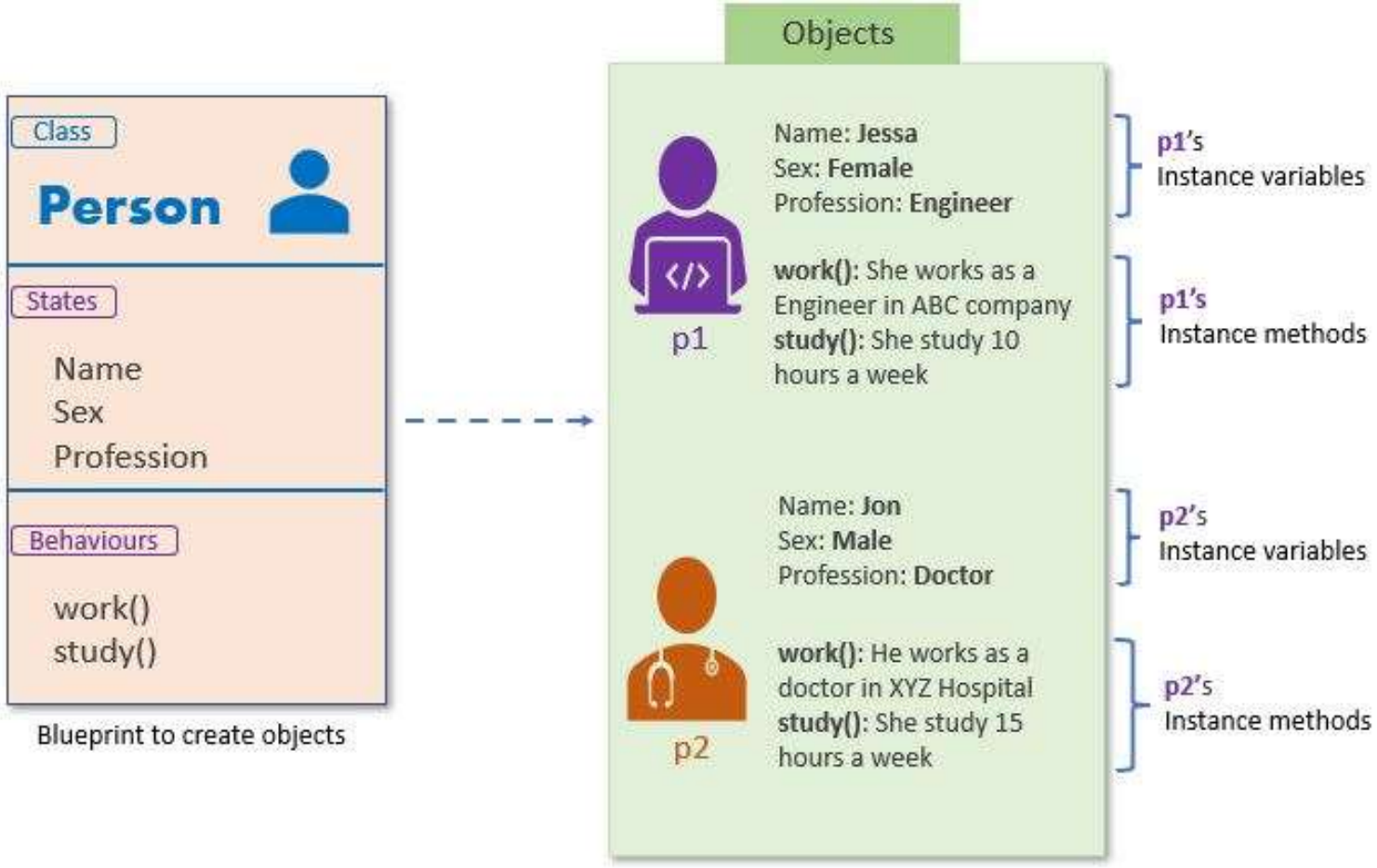
What is a Class and Objects in Python?

UKA TARSADIA university
Imparting Knowledge. Awakening Wisdom. Transforming Lives.

- **Class**: The class is a user-defined data structure that binds the data members and methods into a single unit. Class is a **blueprint or code template for object creation**. Using a class, you can create as many objects as you want.
- **Object**: An **object is an instance of a class**. It is a collection of attributes (variables) and methods. We use the object of a class to perform actions.

Every object has the following properties.

- **Identity**: Every object must be uniquely identified.
- **State**: An object has an attribute that represents a state of an object, and it also reflects the property of an object.
- **Behavior**: An object has methods that represent its behavior.


Class
Person
States
Name
Sex
Profession
Behaviours
work()
study()
Blueprint to create objects
Objects
p1
Name: Jessa
Sex: Female
Profession: Engineer
work(): She works as a Engineer in ABC company
study(): She study 10 hours a week
p1's Instance variables
p1's Instance methods
p2
Name: Jon
Sex: Male
Profession: Doctor
work(): He works as a doctor in XYZ Hospital
study(): She study 15 hours a week
p2's Instance variables
p2's Instance methods

In Python, class is defined by using the class keyword. The syntax to create a class is given below.

```
class class_name:
    '''This is a docstring. I have created a new class'''
    <statement 1>
    <statement 2>
    .
    .
    <statement N>
```

- class_name: It is the name of the class
- Docstring: It is the first string inside the class and has a brief description of the class. Although not mandatory, this is highly recommended.
- statements: Attributes and methods

Example: Define a class in Python


```python
class Person:
    def __init__(self, name, sex, profession):
        # data members (instance variables)
        self.name = name
        self.sex = sex
        self.profession = profession

    # Behavior (instance methods)
    def show(self):
        print('Name:', self.name, 'Sex:', self.sex, 'Profession:', self.profession)

    # Behavior (instance methods)
    def work(self):
        print(self.name, 'working as a', self.profession)
```

Create Object of a Class

- An object is essential to work with the class attributes. The object is created using the class name.
- When we create an object of the class, it is called instantiation.
- The object is also called the instance of a class.
- A constructor is a special method used to create and initialize an object of a class. This method is defined in the class.
- In Python, Object creation is divided into two parts in **Object Creation** and **Object initialization**
- Internally, the __new__ is the method that creates the object
- And, using the __init__() method we can implement constructor to initialize the object.

Syntax

```
<object-name> = <class-name>(<arguments>)
```

Below is the code to create the object of a Person class

```
jessa = Person('Jessa', 'Female', 'Software Engineer')
```

le:

```python
class Person:
    def __init__(self, name, sex, profession):
        # data members (instance variables)
        self.name = name
        self.sex = sex
        self.profession = profession

    # Behavior (instance methods)
    def show(self):
        print('Name:', self.name, 'Sex:', self.sex, 'Profession:', self.profession)

    # Behavior (instance methods)
    def work(self):
        print(self.name, 'working as a', self.profession)

# create object of a class
jessa = Person('Jessa', 'Female', 'Software Engineer')

# call methods
jessa.show()
jessa.work()
```

```
Name: Jessa Sex: Female Profession:
Software Engineer Jessa working as a
Software Engineer
```

Methods in Classes

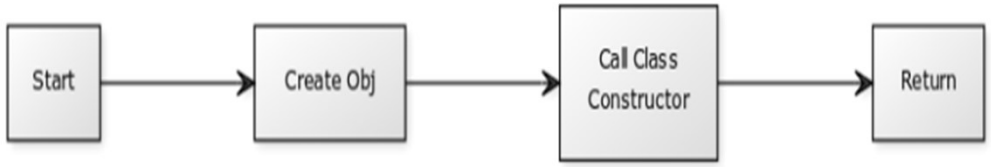
- Define a method in a class by including function definitions within the scope of the class block
- There must be a special first argument **self** in all of method definitions which gets bound to the calling instance
- There is usually a special method called **__init__** in most classes

What is a constructor in Python?

- The constructor is a method that is called when an object is created. This method is defined in the class and can be used to initialize basic variables.
- If you create four objects, the class constructor is called four times. Every class has a constructor, but its not required to explicitly define it.

Example

- Constructor
- Each time an object is created a method is called. That methods is named the constructor.
- The constructor is created with the function init. As parameter we write the self keyword, which refers to itself (the object).



- Inside the constructor we initialize two variables: legs and arms. Sometimes variables are named properties in the context of object-oriented programming.
- We create one object (bob) and just by creating it, its variables are initialized.

```
class Human:
    def __init__(self):
        self.legs = 2
        self.arms = 2

bob = Human()
print(bob.legs)
```

Types of Constructors in Python

1. Parameterized Constructor
2. Non-Parameterized Constructor
3. Default Constructor

Parameterized Constructor in Python

In Python, a parameterized constructor is a type of constructor that takes additional arguments besides the standard self reference. These arguments are used to initialize the object's attributes or perform other operations when the object is created.

```
class Family:
    # Constructor - parameterized
    members = 5

    def __init__(self, count):
        print("This is a parameterized constructor")
        self.members = count

    def show(self):
        print("Number of members is", self.members)

family_object = Family(10)
family_object.show()
```

Output:

```
This is a parameterized constructor
Number of members is 10
```

Non-Parameterized Constructor in Python

A non-parameterized constructor in Python is a constructor that does not accept any arguments except the mandatory self. This type of constructor is used for initializing class members with default values or performing standard initialization tasks that do not require external inputs.

```python
class Fruits:
    favourite = "Apple"

    # Non-parameterized constructor
    def __init__(self):
        self.favourite = "Orange"

    # A method to display the favorite fruit
    def show(self):
        print(self.favourite)

# Creating an object of the class
obj = Fruits()

# Calling the instance method using the object
obj.show()
```

Output:

```
Orange
```

Default Constructor in Python

When a class is defined without an explicit constructor in Python, Python automatically provides a default constructor. This default constructor is a basic, non-parameterized constructor that performs no specific task or initialization beyond the basic object creation.

```python
class Assignments:
    check = "not done"

    # A method to check assignment status
    def is_done(self):
        print(self.check)

# Creating an object of the class
obj = Assignments()

# Calling the instance method using the object
obj.is_done()
```

Output:

```
not done
```

A Simple Class def: Student

```
class Student:
"""A class representing a student """
        def ____init____(self , n, a):
                self.full_name = n
                self.age = a
        def get_age(self):                    #Method
                return self.age
```

- Define class:
  - Class name, begin with capital letter, by convention
  - object: class based on (Python built-in type)
  - Define a method
  - Like defining a function
  - Must **have a special first parameter**, self, which provides way for a method to refer to object itself

21

Instantiating Objects with '__init__ '

- __init__is the default constructor
- __init__is serves as a constructor for the class. Usually does some initialization work
- An _____init___method can take any number of arguments
- However, the first argument self in the definition of _init__is special

Self

- The first argument of every method is a reference to the current instance of the class
- By convention, we name this argument **self**
- In __init__, self refers to the object currently being created; so, in other class methods, it refers to the instance whose method was called
- Similar to the keyword 'this' in Java or C++
- But Python uses self more often than Java uses this
- You **do not** give a value for this parameter(self) when you call the method, Python will provide it.

Continue…

- Although you must specify self explicitly when defining the method, you don't include it when calling the method.
- Python passes it for you automatically

Defining a method:

(this code inside a class definition.)

```
def get_age(self, num):
self.age = num
```

Calling a method:

```
>>> x.get_age(23)
```

Deleting instances: No Need to "free"

- When you are done with an object, you don't have to delete or free it explicitly.
- Python has automatic garbage collection.
- Python will automatically detect when all of the references to a piece of memory have gone out of scope. Automatically frees that memory.
- Generally works well, few memory leaks
- There's also no "destructor" method for classes

Syntax for accessing attributes and methods

```
>>> f = student("Python", 14)

>>> f.full_name # Access attribute
"Python"

>>> f.get_age() # Access a
method 14
```

Creating Classes for Objects

```
class Puppy(object):
            def __init__(self, name, color):
         self.name = name
         self.color = color
      def bark(self):
         print "I am", color, name
puppy1 = Puppy("Max", "brown")
puppy1.bark()
puppy2 = Puppy("Ruby", "black")
puppy2.bark()
```

Class: Code that defines the **attributes** and **methods** of a kind of object

Instantiate: To create an object; A single object is called an **Instance**

The Simple Critter Program

```
class Critter(object):
    """A virtual pet"""
    def talk(self):
        print "Hi. I'm an instance of class Critter."
# main
crit = Critter()
crit.talk()
```

Define class:

- Class name, begin with capital letter, by convention
- object: class based on (Python built-in type)

Define a method

- Like defining a function
- Must **have a special first parameter**, `self`, which provides way for a method to refer to object itself

Simple Program for Class and Object

```
class Student:
# Initializing the variables
def _____init__(self,name,age):
self.fullname=name self.sage=age
# Display the entered data such as Students name and age
def display(self):
print 'Student FullName: %s' %(self.fullname)
print 'Stuent Age: %d'%(self.sage)
# S1,S2 and S3 objects
# S1,S2 and S3 are three different student details and age
s1=Student('Python',14)
s1.display()
s2=Student('Jython',23)
s2.display()
s3=Student('Objects',45)
s3.display()
```

Output for the Sample Code

Two More Special Methods

```
class CustomList(object):
        def __init__(self, elements=0):
                self.my_custom_list = [0] * elements
        def __setitem__(self, index, value):
                self.my_custom_list[index] = value
        def __getitem__(self, index):
                return "Hey you are accessing {} element whose value is: {}".format(index,
self.my_custom_list[index])
        def __str__(self):
                return 'hey these are my contents' + str(self.my_custom_list)

>>> obj = CustomList(12)
>>> print(CustomList())
hey these are my contents[]
>>> obj[0] = 1
>>> print(obj[0])
Hey you are accessing 0 element whose value is: 1
>>> print(obj)
hey these are my contents[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Two More Special Methods

```
class Puppy(object):
    def __init__(self):
        self.name = []
        self.color = []

    def __setitem__(self, name, color):
        self.name.append(name)
        self.color.append(color)

    def __getitem__(self, name):
        if name in self.name:
            return self.color[self.name.index(name)]
        else:
            return None

dog = Puppy()
dog['Max'] = 'brown'
dog['Ruby'] = 'yellow'
print "Max is", dog['Max']
```

Using Class Attributes and Static Methods

Class attribute: A single attribute that's associated with a **class** itself (not an instance!)

Static method: A method that's associated with a class itself

Class attribute could be used for counting the total number of objects instantiated, for example

Static methods often work with class attributes

Creating a Class Attribute


```
class Critter(object):
    total = 0
```

`total = 0` creates class attribute `total` set to `0`

- Assignment statement in class but outside method creates class attribute
- **Assignment statement executed only once, when Python first sees class definition**
- Class attribute exists even before single object created
- Can use class attribute without any objects of class in existence

Accessing a Class Attribute

```
class Critter(object):
    total = 0

    def status():
        print "Total critters", Critter.total

    status = staticmethod(status)

    def __init__(self, name):
        Critter.total += 1

print Critter.total     #the class
print crit1.total       #the instance
#crit1.total += 1  # won't work;
```

can't assign new value to a class attribute through instance

Creating a Static Method

```
class Critter(object):
...
    def status():
        print "Total critters", Critter.total

    status = staticmethod(status)

status()
```

Is static method

Doesn't have `self` in parameter list because method will be invoked through class not object

`staticmethod()`

Built-in Python function

Takes method and returns static method

Invoking a Static Method

```
...
crit1 = Critter("critter 1")
crit2 = Critter("critter 2")
crit3 = Critter("critter 3")

Critter.status()

Critter.status()
```

- Invokes static method `status()` defined in `Critter`
- Prints a message stating that 3 critters exist
- Works because constructor increments class attribute `total`, which `status()` displays

classy_critter.py

Setting default values

```
class Person(object):
        def __init__(self, name="Tom", age=20):
                self.name = name
                self.age = age
        def talk(self):
                print ("Hello, I am", self.name)
        def __str__(self):
                return "Hi, I am " + self.name

one = Person(name="Yuzhen", age = "forever 20")
print (one)
one.talk()
```

```
Hi, I am Yuzhen
Hello, I am Yuzhen
```

```
two = Person()
print (two)
two.talk()
```

```
Hi, I am Tom
Hello, I am Tom
```

Encapsulation

- Important advantage of OOP consists in the encapsulation of data. We can say that object-oriented programming relies heavily on encapsulation.
- The terms encapsulation and abstraction (also data hiding) are often used as synonyms. They are nearly synonymous,

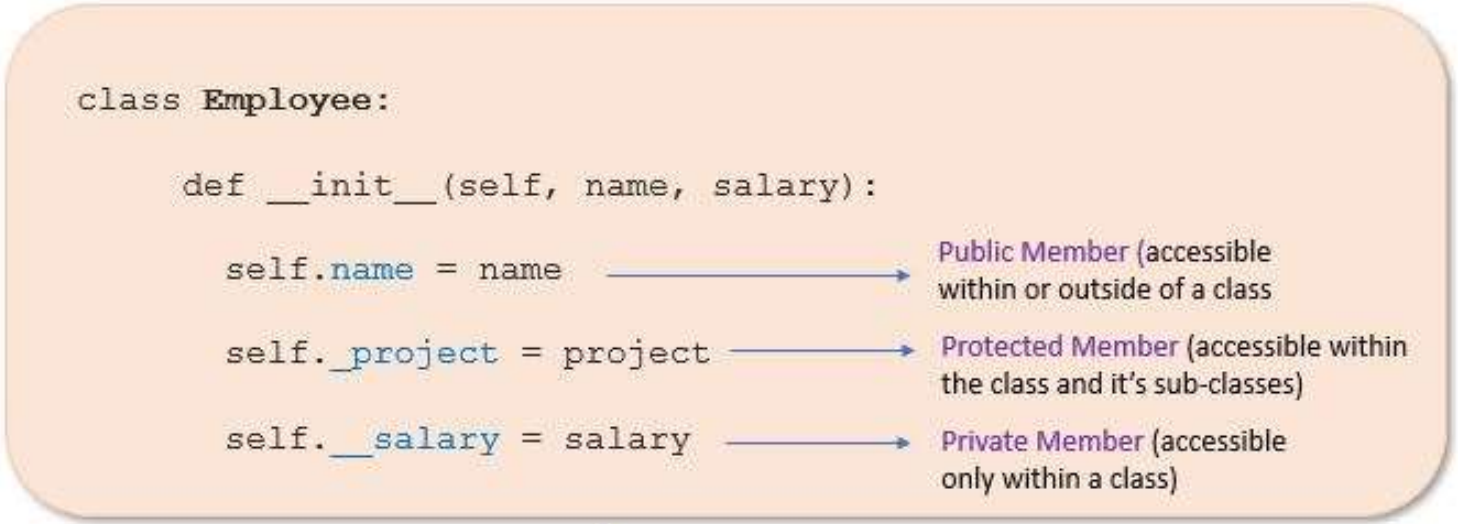
i.e. abstraction is achieved though encapsulation.

- Data hiding and encapsulation are the same concept, so it's correct to use them as synonyms
- Generally speaking encapsulation is the mechanism for restricting the access to some of an objects's components, this means, that the internal representation of an object can't be seen from outside of the objects definition.

- Access to this data is typically only achieved through special methods: ***Getters*** and ***Setters***
- By using solely get() and set() methods, we can make sure that the internal data cannot be accidentally set into an inconsistent or invalid state.
- C++, Java, and C# rely on the public, private, and protected keywords in order to implement variable scoping and encapsulation
- It's nearly always possible to circumvent this protection mechanism

Public, Protected and Private Data

- If an identifier doesn't start with an underscore character "_" it can be accessed from outside, i.e. the value can be read and changed
- Data can be protected by making members private or protected. Instance variable names starting with two underscore characters cannot be accessed from outside of the class.
- At least not directly, but they can be accessed through private name mangling.
- That means, private data __A can be accessed by the following name construct: ***instance_name._classname A***



Data Hiding using Encapsulation

Public Member

Public data members are accessible within and outside of a class. All member variables of the class are by default public.

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data members
        self.name = name
        self.salary = salary

    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing public data members
print("Name: ", emp.name, 'Salary:', emp.salary)

# calling public method of the class
emp.show()
```

Output

```
Name:  Jessa Salary: 10000
Name:  Jessa Salary: 10000
```

Private Member

We can protect variables in the class by marking them private. To define a private variable add two underscores as a prefix at the start of a variable name.

Private members are accessible only within the class, and we can't access them directly from the class objects.

```
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing private data members
print('Salary:', emp.__salary)
```

- If an identifier is only preceded by one underscore character, it is a protected member.
- Protected members can be accessed like public members from outside of class

Example:

```
class Encapsulation(object):
def ___init____(self, a, b, c):
self.public = a
self._protected = b
self.___private = c
```

The following interactive sessions shows the behavior of public, protected and private members:

```
>>> x = Encapsulation(11,13,17)
>>> x.public
 11
>>> x._protected
13
>>> x._protected = 23
>>> x._protected
23
>>> x._____private
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Encapsulation' object has no attribute '              private‘
>>>x._encapsulation__private
17
```

The following table shows the different behavior Public, Protected and Private Data

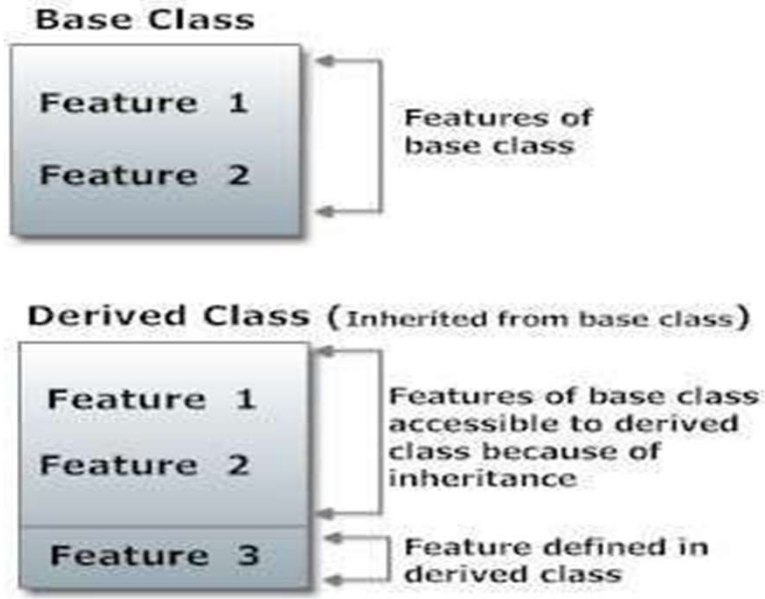
Name	Notation	Behavior
name	Public	Can be accessed from inside and outside
_name	Protected	Like a public member, but they shouldn't be directly accessed from outside
_____name	Private	Can't be seen and accessed from outside

Inheritance

- Inheritance is a powerful feature in object oriented programming
- It refers to defining a new class with little or no modification to an existing class.
- The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.
- Derived class inherits features from the base class, adding new features to it.
- This results into re-usability of code.

Syntax:

```
class Baseclass(Object):
    body_of_base_class
class DerivedClass(BaseClass):
    body_of_derived_clas
```



While designing a inheritance concept, following key pointes keep it in mind

- A sub type never implements less functionality than the super type
- Inheritance should never be more than two levels deep
- We use inheritance when we want to avoid redundant code.

Two built-in functions *isinstance()* and *issubclass()* are used to check inheritances.

- Function isinstance() returns True if the object is an instance of the class or other classes derived from it.
- Each and every class in Python inherits from the base class object.

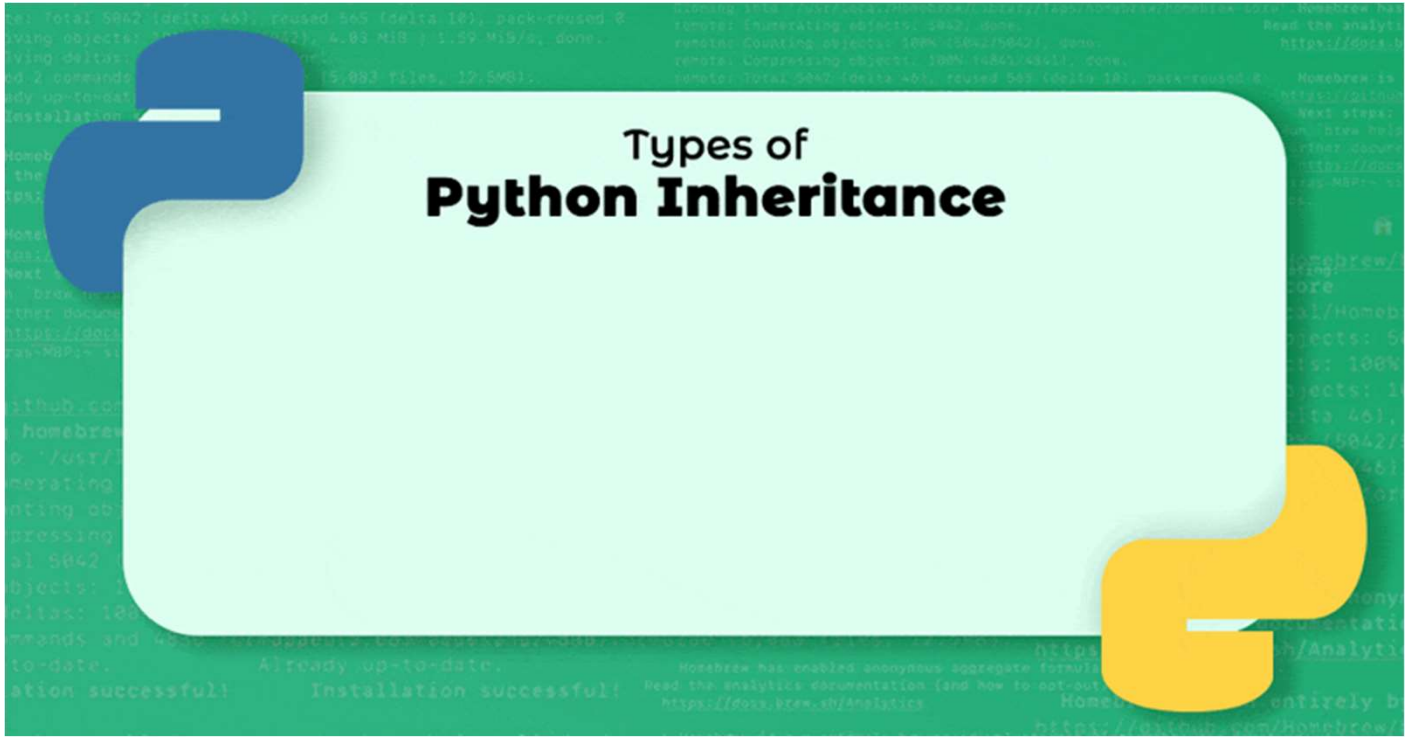
Sample code for Inheritance

```
class Person(object):
        def __init__(self, name):
                self.name = name
        def getName(self):
                return self.name
        def isEmployee(self):
            return False
class Employee(Person):
        def isEmployee(self):
             return True
emp = Person("Geek1")  # An Object of Person
print(emp.getName(), emp.isEmployee())

emp = Employee("Geek2") # An Object of Employee
print(emp.getName(), emp.isEmployee())
```

Output

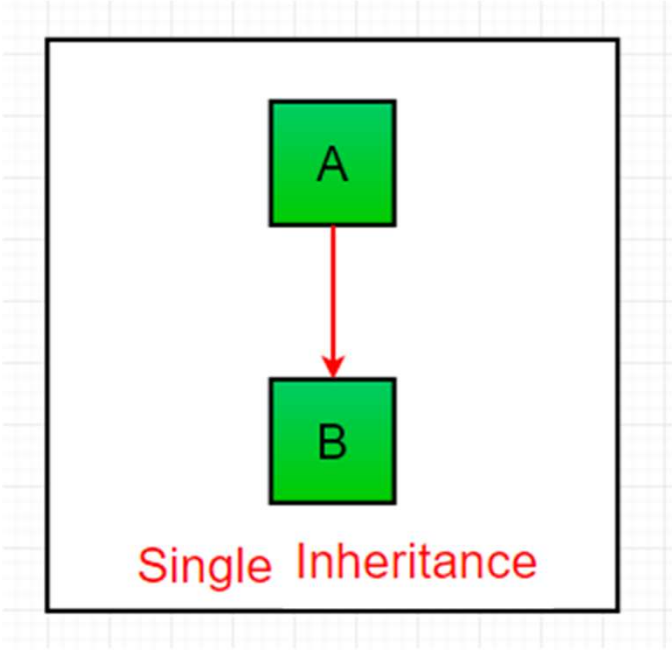
```
Geek1 False
Geek2 True
```

Types of inheritance Python



Single Inheritance:

Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.



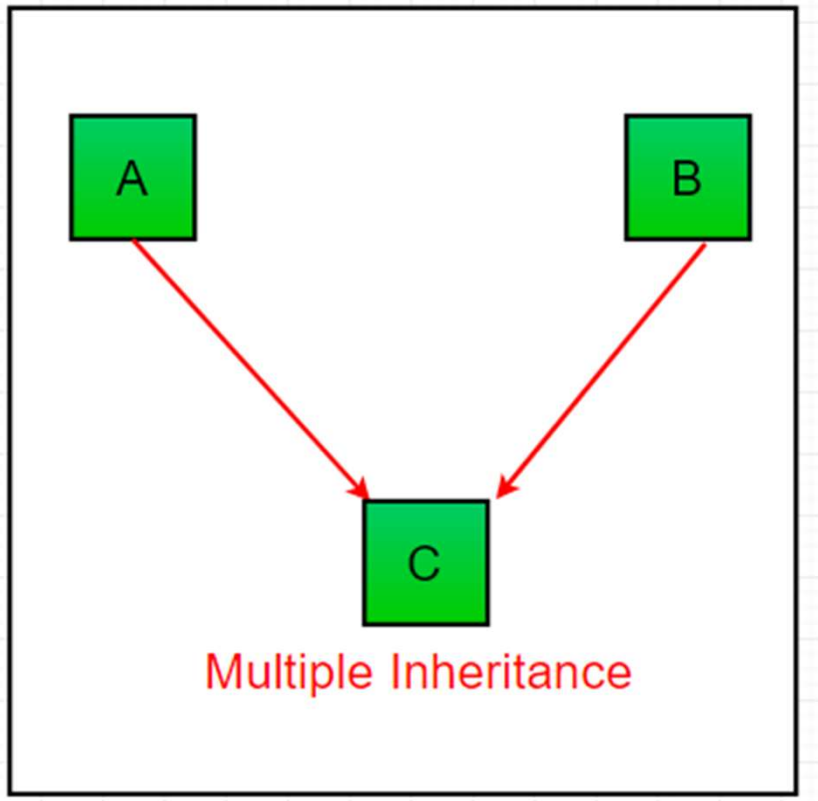
Example

```
# Python program to demonstrate
# single inheritance

# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class


class Child(Parent):
    def func2(self):
        print("This function is in child class.")


# Driver's code
object = Child()
object.func1()
object.func2()
```

```
This function is in parent class.
This function is in child class.
```

Multiple Inheritance:

- When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.



Example

```
# Python program to demonstrate
# multiple inheritance

# Base class1
class Mother:
    mothername = ""

    def mother(self):
        print(self.mothername)

# Base class2


class Father:
    fathername = ""

    def father(self):
        print(self.fathername)

# Derived class


class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)
```

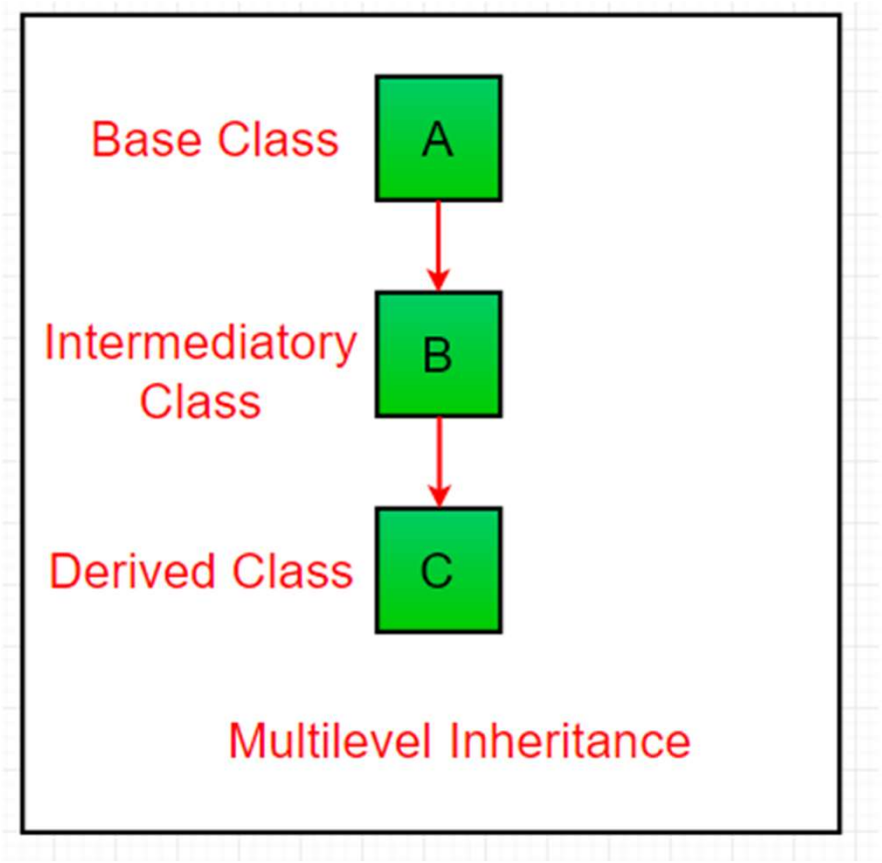
```
# Driver's code
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

Output:

```
Father : RAM
Mother : SITA
```

Multilevel Inheritance :

- In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.



```python
class Grandfather:

    def __init__(self, grandfathername):
        self.grandfathername = grandfathername

# Intermediate class


class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername

        # invoking constructor of Grandfather class
        Grandfather.__init__(self, grandfathername)

# Derived class


class Son(Father):
    def __init__(self, sonname, fathername, grandfathername):
        self.sonname = sonname

        # invoking constructor of Father class
        Father.__init__(self, fathername, grandfathername)

    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)
```

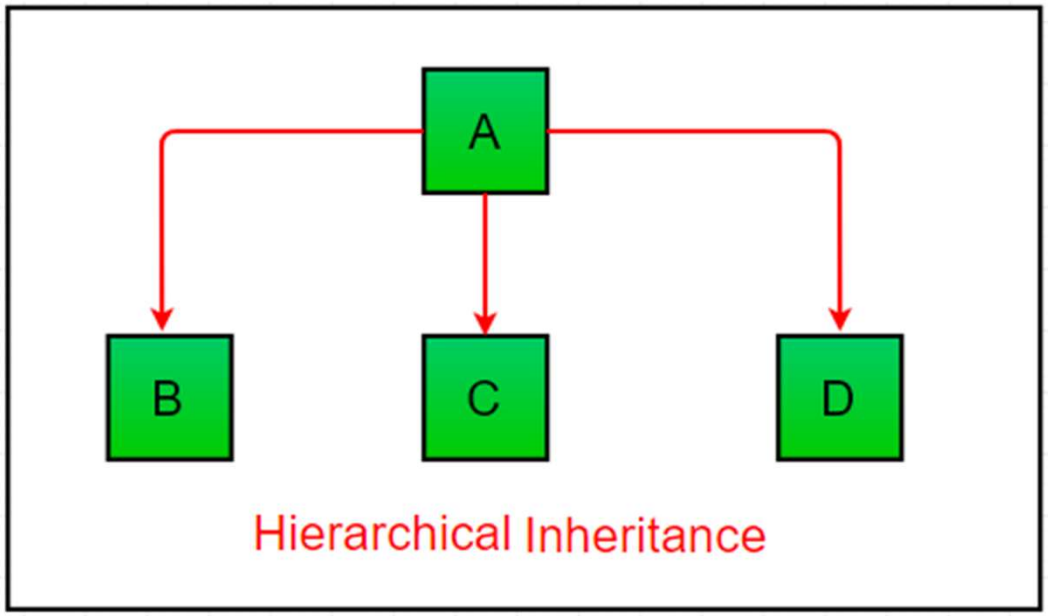
```python
#  Driver code
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

Output:

```
Lal mani
Grandfather name : Lal mani
Father name : Rampal
Son name : Prince
```

Hierarchical Inheritance:

- When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.



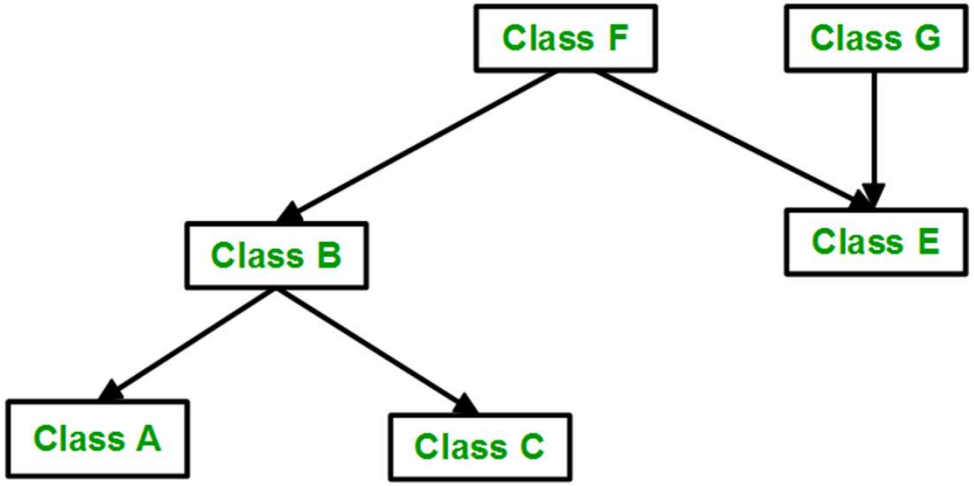
```python
# Python program to demonstrate
# Hierarchical inheritance

# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class1


class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")

# Derivied class2


class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")


# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output:

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

Hybrid Inheritance:

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.



```python
# Python program to demonstrate
# hybrid inheritance


class School:
    def func1(self):
        print("This function is in school.")


class Student1(School):
    def func2(self):
        print("This function is in student 1. ")


class Student2(School):
    def func3(self):
        print("This function is in student 2.")


class Student3(Student1, School):
    def func4(self):
        print("This function is in student 3.")


# Driver's code
object = Student3()
object.func1()
object.func2()
```

Output:

```
This function is in school.
This function is in student 1.
```

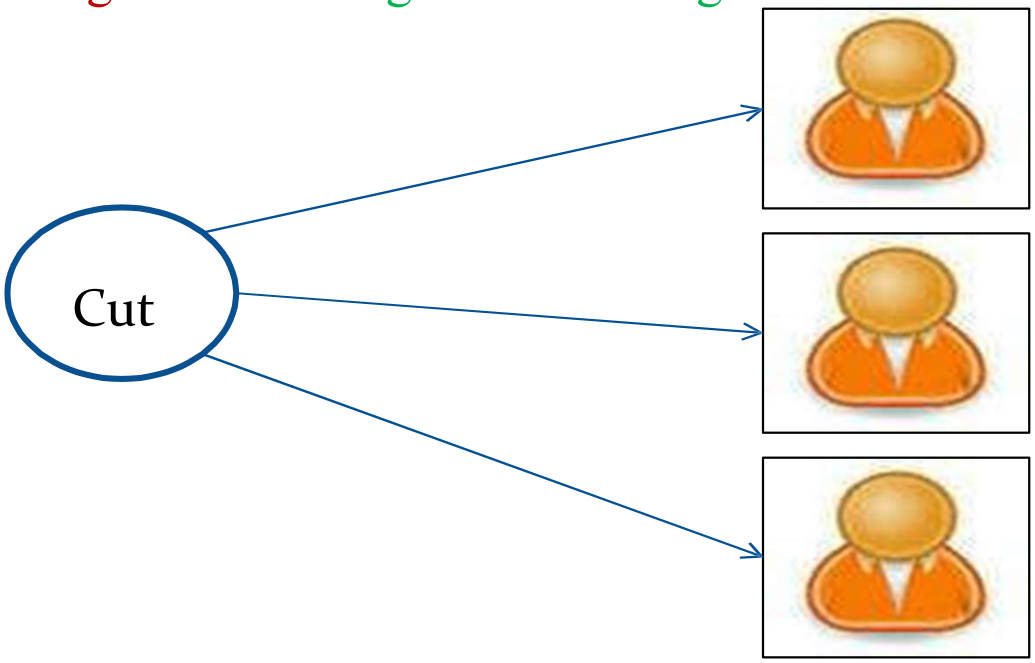
Polymorphism

- Polymorphism in Latin word which made up of ***'ploy'*** means many and ***'morphs'*** means forms
- From the Greek , Polymorphism means ***many(poly)*** ***shapes (morph)***
- This is something similar to a word having several different meanings depending on the context
- Generally speaking, polymorphism means that a method or function is able to cope with different types of input.

A simple word 'Cut' can have different meaning depending where it is used

If any body says "Cut" to these people

Surgeon: The Surgeon would begin to make an incision



Hair Stylist: The Hair Stylist would begin to cut someone's hair

Actor: The actor would abruptly stop acting out the current scene, awaiting directional guidance

In OOP , Polymorphism is the characteristic of being able to assign a different meaning to a particular symbol or operator in different contexts specifically to allow an entity such as a variable, a function or an object to have more than one form.

There are two kinds of Polymorphism

Overloading :

Two or more methods with different signatures Overriding:

Replacing an inherited method with another having the same signature

What is Method Overloading?

- Method Overloading is a fundamental concept in OOP that enables a class to define multiple methods with the same name but different parameters

Example

```
def addition(a, b):
    c = a + b
    print(c)


def addition(a, b, c):
    d = a + b + c
    print(d)


# the below line shows an error
# addition(4, 5)

# This line will call the second product method
addition(3, 7, 5)
```

output

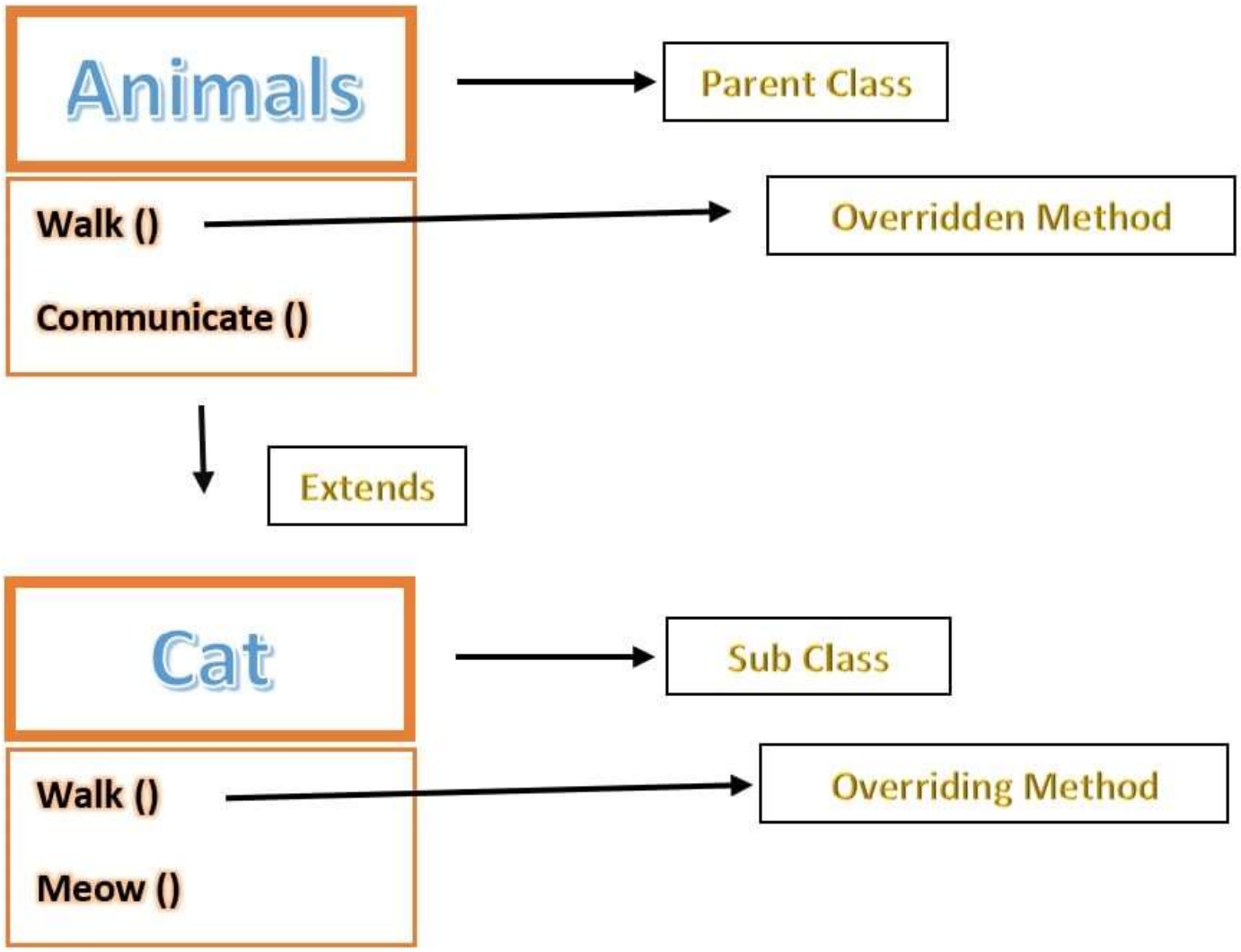
```
15
```

Method Overriding

- Method Overriding is another crucial concept in OOP) that empowers subclasses to provide a specific implementation for a method already defined in their superclass. In Python, this powerful feature allows developers to tailor the behavior of a method in a subclass to suit the unique requirements of that subclass.
- **Method overriding** in Python is when you have two methods with the same name that each perform different tasks. This is an important feature of inheritance in Python.

- There are two prerequisite conditions for Method overriding:

1. Inheritance should be present in the code, method overriding cannot be performed in the same class, and overriding can only be executed when a child class is derived through inheritance.
2. The child class should have the same name and the same number of parameters as the parent class.


Animals
Parent Class
Walk ()
Communicate ()
Overridden Method
Extends
Cat
Sub Class
Walk ()
Meow ()
Overriding Method

Example

```
class Animal:
    def Walk(self):
        print('Hello, I am the parent class')

class Dog(Animal):
    def Walk(self):
        print('Hello, I am the child class')

print('The method Walk here is overridden in the
    code')
```

```
#Invoking Child class through object r

r = Dog()
r.Walk()

#Invoking Parent class through object r

r = Animal()
r.Walk()
```

```
The method Walk here is overridden in the code
Hello, I am the child class
Hello, I am the parent class
```

A Sample Program for Polymorphism

```
class Person:
    def __init__(self, name):
        self.name = name
    def designation(self):
        raise NotImplementedError("Subclass must implement abstract method")
class Employe(Person):
    def designation(self):
        return "Software Engineer"
class Doctor(Person):
    def designation(self):
        return "Cardiologist"
class Student(Person):
    def designation(self):
        return "Graduate Engineer“
```

Operator Overloading

- Python operators work for built-in classes.
- But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings. This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading
- One final thing to mention about operator overloading is that you can make your custom methods do whatever you want. However, common practice is to follow the structure of the built-in methods.

Operator Overloading Sample Program

Programming with Python (IT3008)

```
# add 2 numbers
print(100 + 200)

# concatenate two strings
print('Jess' + 'Roy')

# merger two list
print([10, 20, 30] + ['jessa', 'emma', 'kelly'])
```

Output:

```
300
JessRoy
[10, 20, 30, 'jessa', 'emma', 'kelly']
```

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        # Overloading the '+' operator
        return Point(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        # Overloading the '-' operator
        return Point(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        # Overloading the '*' operator
        return Point(self.x * scalar, self.y * scalar)

    def __eq__(self, other):
        # Overloading the '==' operator
        return self.x == other.x and self.y == other.y

    def __str__(self):
        # Overloading the str() function
        return f"({self.x}, {self.y})"
```

```python
# Create some Point instances
point1 = Point(2, 3)
point2 = Point(1, 1)

# Testing operator overloading
print("point1 + point2:", point1 + point2)  # Calls __add__()
print("point1 - point2:", point1 - point2)  # Calls __sub__()
print("point1 * 3:", point1 * 3)            # Calls __mul__()

# Testing equality
print("point1 == point2:", point1 == point2)  # Calls __eq__()
print("point1 == Point(2, 3):", point1 == Point(2, 3))  # Calls __eq__()

# Testing string representation
print("String representation of point1:", str(point1))  # Calls __str__()
```

output

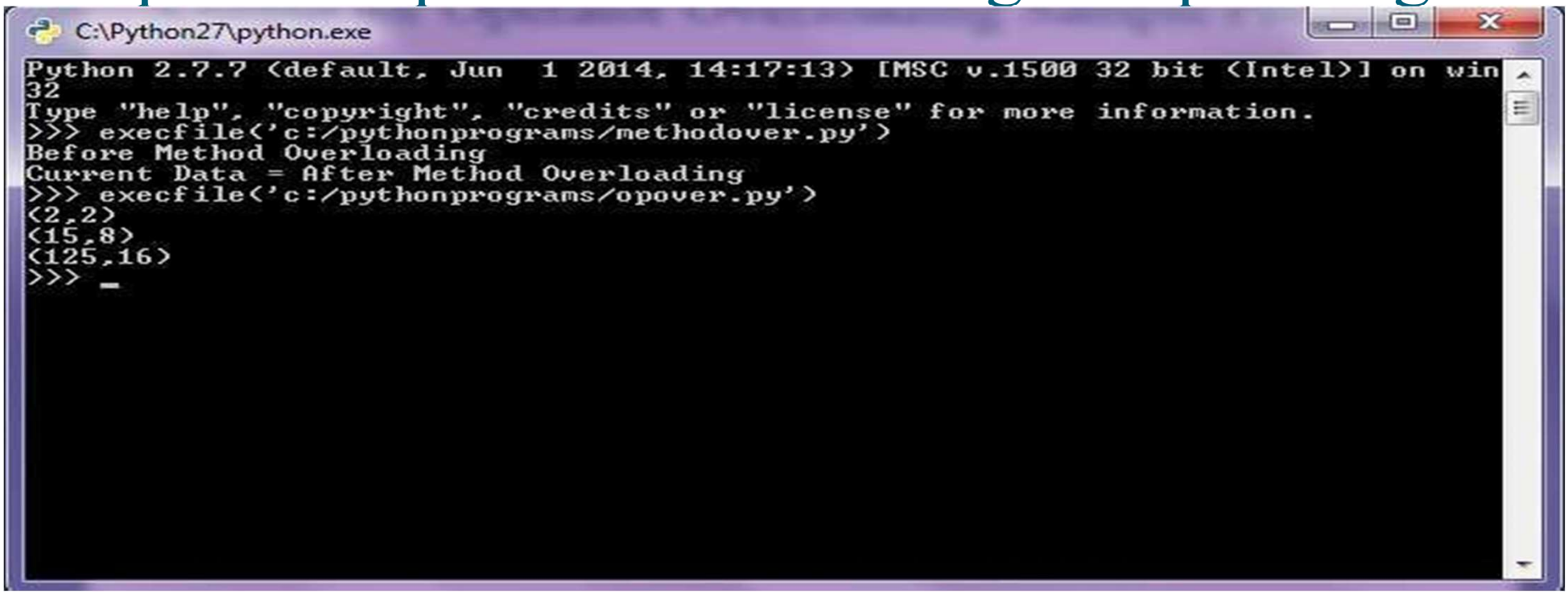
```
point1 + point2: (3, 4)
point1 - point2: (1, 2)
point1 * 3: (6, 9)
point1 == point2: False
point1 == Point(2, 3): True
String representation of point1: (2, 3)
```

Explanation for Operator Overloading Sample Program

What actually happens is that, when you do p1 - p2, Python will call p1.___sub___(p2) which in turn is Point.__sub (p1,p2). Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	Expression	Internally
Addition	p1 + p2	p1._____add_____(p2)
Subtraction	p1 – p2	p1._____sub_____(p2)
Multiplication	p1 * p2	p1._____mul_____(p2)
Power	p1 ** p2	p1._____pow_____(p2)
Division	p1 / p2	p1._____truediv____(p2)

Output for Operator Overloading Sample Program



1. For this problem, we'll round an int value up to the next multiple of 10 if its rightmost digit is 5 or more, so 15 rounds up to 20. Alternately, round down to the previous multiple of 10 if its rightmost digit is less than 5, so 12 rounds down to 10. Given 3 ints, a b c, return the sum of their rounded values.

2. Given two strings, return True if either of the strings appears at the very end of the other string, ignoring upper/lower case differences (in other words, the computation should not be "case sensitive").