CIT Students Only

Object Oriented Programming: Concept of class, object, and instances, constructor, class attributes, and destructor; Real-time use of class in live projects, Inheritance, overlapping and overloading operators, Adding and retrieving dynamic attributes of classes, Programming using OOPs support
Design with Classes: Objects and Classes, Data Modeling Examples, Structuring Classes with Inheritance and Polymorphism.

Brief History of OOP
The term "**Object Oriented Programming**" was first coined by **Alan Kay** around **1966**. **Simula** was the first programming language that has the features of OOP.

OOP began to grow in the **1990s with the popularity of C++**. After that, the OOPs methodology has been adopted by several programming languages such as **Python and Java**.

Now OOP applications are used in almost every field such as Web apps, Mobile apps, Machine Learning, Artificial Intelligence, Data Science, Expert systems, Client-server systems, Object-oriented databases, and so on.

What is the overview of OOPs and explain the advantages and disadvantages of OOPs?

What is OOPs in Python?
OOPs stands for **O**bject-**O**riented **P**rogramming **S**ystem. This programming paradigm/methodology focuses on organizing code into **objects.** The objects are self-contained instances of classes. OOPS allows us to design our code similar to real-world entities making our code easier to understand, maintain, and reuse.

Object Oriented Programming in Python solves a problem by treating every entity as an Object.

What is a Class and an Object?

- ➢ **Class:** A class is a blueprint for creating objects. It defines the **data (attributes)** and **behaviors (methods)** of the objects of the class. In other words, a class is a template or a set of instructions for creating objects.
- ➢ **Object:** An object is an instance of a class. Each object is a **combination of data and methods that operate on that data.**

We can create many objects for a given class.

CIT Students Only

What are the 4 main principles (or pillars) of OOPs?
The main principles of OOPs in Python are Abstraction, Encapsulation, Inheritance, and Polymorphism.

1. **Abstraction:** Abstraction is the ability to hide the unnecessary or sensitive part of our code implementation from the user. The abstraction would still provide a simple and easy-to-use interface. Data abstraction in Python can be achieved by creating abstract classes.

2. **Encapsulation:** Encapsulation refers to the practice of wrapping data (attributes) and methods within a class to protect the data from external interference. This adds restrictions to directly access the variables and methods. To prevent accidental modifications, an object's data variable can only be changed by the object's method. Such variables are called private variables. **Note:** The private variables or methods must be preceded by __ (double underscore).

3. **Inheritance:** Inheritance enables the creation of a new class (derived or child class) from an existing class (base or parent class). A derived class inherits the data and methods from the base class and can have new data and methods, or may modify the parent's data and methods.

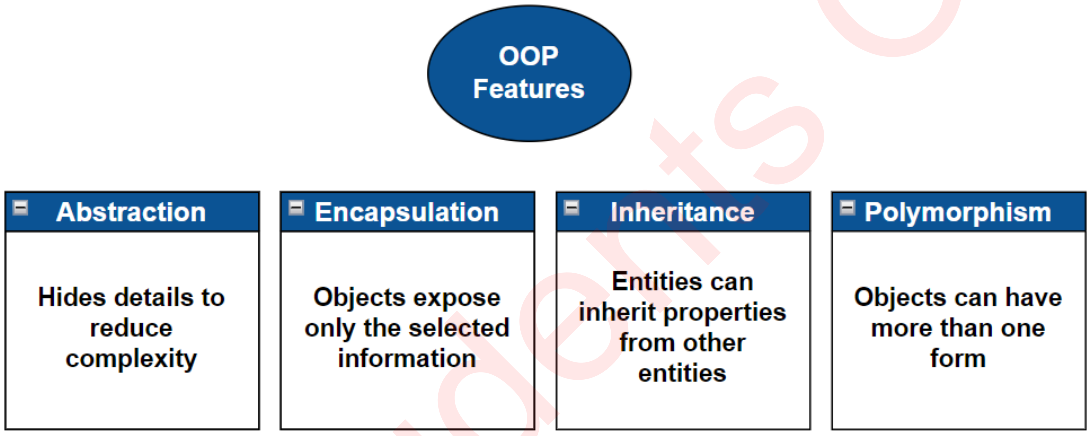
   Benefits of Inheritance:
   - Better representation of real-world relationships
   - Code reusability without rewriting the same code
   - Add more features to a class without modifying it
   - Create specialized child classes based on more general parent classes.

   4 Types of Inheritance
   1. **Single** Inheritance - derived class inherits properties from a single parent class
   2. **Multilevel** Inheritance - derived class inherits properties from its immediate parent class which in turn inherits properties from its parent class
   3. **Hierarchical** Inheritance - more than one derived classes inherit properties from a parent class
   4. **Multiple** Inheritance - one derived class inherits properties from more than one parent class.

4. **Polymorphism:** Polymorphism means having many forms. In Python, it means the ability to use an object of a derived class wherever an object of the base class is expected. Polymorphism is achieved through **method overriding** and **method overloading**. **Ex:** using the same function name with different definitions are used for different data types and different numbers of arguments.

These principles of OOPs provide a powerful and flexible way to design and organize code. That helps to manage and develop complex software systems easily. Python supports OOPs methodology to create classes, define objects, and utilize inheritance and polymorphism in our application programs.

CIT Students Only


OOP Features
Abstraction
Hides details to reduce complexity
Encapsulation
Objects expose only the selected information
Inheritance
Entities can inherit properties from other entities
Polymorphism
Objects can have more than one form

Advantages of OOP

1. **Reusability:** OOP allows developers to create code that can be reused in different parts of an application. This makes development faster and more efficient because developers do not have to write new code from scratch each time they need to create a new feature.
2. **Modularity**: OOP allows developers to break down complex systems into smaller, more manageable modules. This makes it easier to develop, test, and maintain code because changes made to one module do not affect other parts of the system.
3. **Encapsulation:** OOP allows developers to hide the implementation details of objects, making it easier to change the behavior of an object without affecting other parts of the system. It limits access to sensitive data and improves security.
4. **Inheritance**: OOP allows developers to create new classes by inheriting characteristics from existing classes. This reduces the amount of code that needs to be written and makes it easier to maintain the codebase.

Disadvantages of OOP

1. **Steep Learning Curve:** OOP is a complex paradigm, and it can take time for developers to become proficient in it. The concepts of inheritance, polymorphism, and encapsulation can be difficult to understand for beginners.
2. **More resources:** OOP often requires more memory and processing power than other paradigms.
3. **Complexity:** OOP can lead to complex code, especially when dealing with large systems that have many interdependent objects. This complexity can make it more difficult to debug and maintain code.

CIT Students Only

Explain creating classes and instance objects with examples.

Class: A class is a blueprint for creating objects. It defines the **data attributes (variables)** and **dynamic behaviors (methods)** of the objects of the class. In other words, a class is a template or a set of instructions for creating objects with characteristics and functionalities.
Note: The data variables and methods inside a class are called members of the class.

CLASS
Name
Data Attributes (Variables)
Dynamic Behaviors (Methods)

Syntax: class

```
class ClassName:
   # class definition - data attributes
   # class definition - methods
```

class - is a keyword to define a new **class object**
ClassName - is the user-defined name of the class followed by colon (:)
class definition - area to define data attributes and method behaviors; The statements inside a class are any of these following

1. Variable definitions
2. Method definitions
   a. Sequential instructions
   b. Decision control statements
   c. Loop statements

Create a class named phone:

```
class phone:
    # Data or Attributes
    price = 1000
    qty = 1
```

Object: An object is an instance of a class. Each object is a **combination of data and methods** that operate on that data.

- Creating an object of a class is known as class instantiation.
- Once a class is defined, we can create an object of that class.
- The object can then **access class variables** and **class methods** using a **dot operator**

CIT Students Only

Syntax: To Create an Object

```
objectVar = className()
```

objectVar - is a user-defined object-name for an **instance object**
className - the name of the **class object**

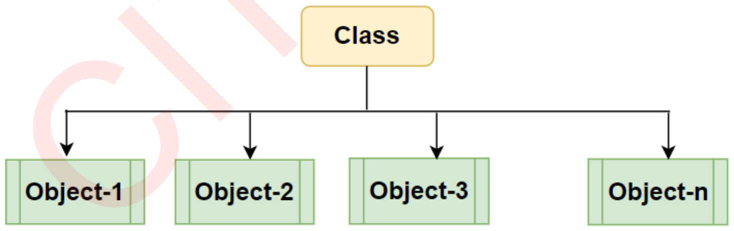
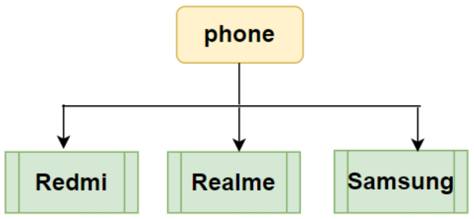
Syntax: To Access Class-Member through an Object

```
objectVar . Class-Member
```

objectVar - is a user-defined variable/object-name
Class-Member - variable or method

Create an object:

```
Samsung = phone()
```

Many Instance Objects can be Instantiated with a Class	Example
	

Example: Create a Class, Create an Object, Invoke Class-Members

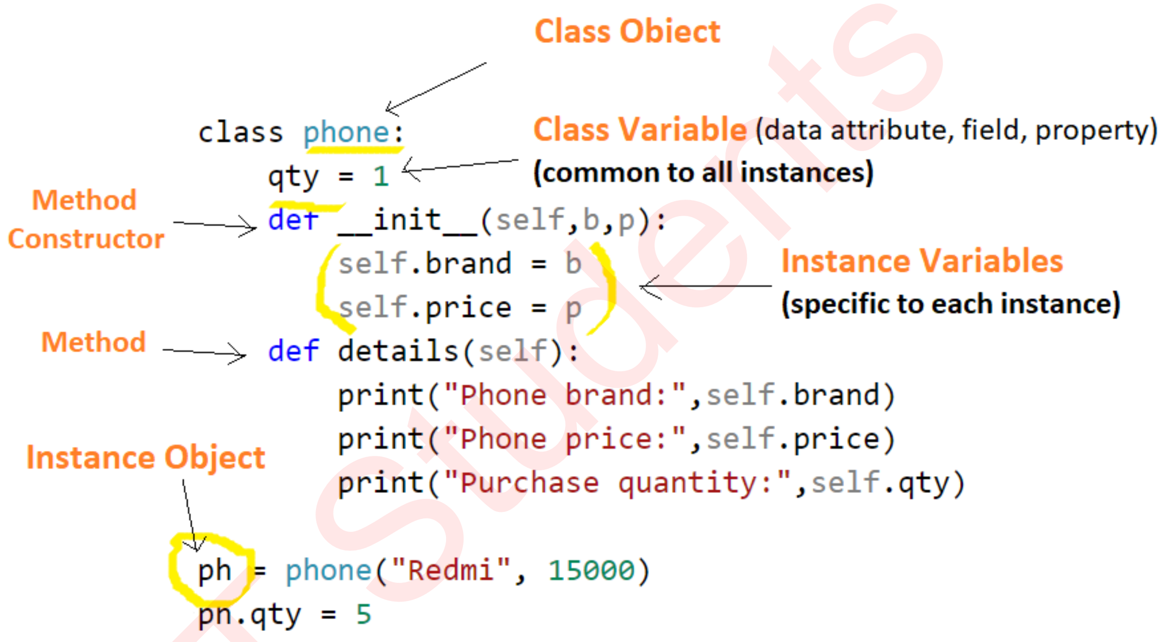
```
class phone:
    # Data or Attributes
    price = 10000
    qty = 1
#Main program
Redmi = phone()    #Create object "Redmi"
print(Redmi.price)#Invoke class-member "price" through object "Redmi"
```

Output:

```
10000
```

CIT Students Only

OOP Terminology


Class Object
Class Variable (data attribute, field, property)
(common to all instances)
Method
Constructor
Instance Variables
(specific to each instance)
Method
Instance Object

```
class phone:
    qty = 1
    def __init__(self,b,p):
        self.brand = b
        self.price = p
    def details(self):
        print("Phone brand:",self.brand)
        print("Phone price:",self.price)
        print("Purchase quantity:",self.qty)

ph = phone("Redmi", 15000)
pn.qty = 5
```

Class methods using the argument " self ":

★ **self** refers to the object itself (self is a pointer to the instance of a class)

★ A method in a class should have its first parameter as "**self**" and then declare rest of the parameters

★ We must pass self to a member function even if it doesn't take any parameter or argument

★ We don't need to pass a value for this parameter. Python will pass when we call a method.

★ The "**self**" in Python is similar to the "**this**" pointer in C++

Application: Class method using argument self

```
class phone:
    # Data or Attributes
    price = 10000
    qty = 1
    # Methods or Behaviours
    def details(self):
        print("Price: ",self.price)
        print("Quantity: ",self.qty)
```

CIT Students Only

```
#Main program
Realme = phone()    #Create object "Realme"
Realme.details()    #Invoke the method details()
Samsung = phone()
Samsung.price=25000
Samsung.qty = 5

Samsung.details()
```

Output:

```
Price:  10000
Quantity:  1
Price:  25000
Quantity:  5
```

What is the constructor and destructor in a class? Demonstrate them with an example

A constructor is a special type of method (function) that is called when we instantiate an object of a class. The constructors are normally used to initialize (assign values) the instance variables.

Creating a constructor:

- The name of the constructor method is always the **_ _init_ _()**
- The first argument of the **__init__()** method must always be the current object instance that is being constructed.
- While creating an object, a constructor can accept arguments if needed.
- When we create a class without a constructor, Python automatically creates a default constructor that doesn't do anything.
- Every class must have a constructor, even if it simply relies on the default constructor.

Syntax:

```
def __init__(self):
   # body of constructor
```

CIT Students Only

Creating a destructor:

- A destructor is usually called when an object gets destroyed.
- Destructors are not much needed in Python unlike C/C++ because Python has Garbage Collector that handles memory management automatically.
- The name of the destructor method is always the **_ _del_ _()**
- The **__del__()** method is automatically called when all references to an object have been deleted i.e when an object is garbage collected.

Note: A reference to an object is also deleted when that object goes out of reference or when the program ends.

Syntax:

```
def __del__(self):
   # body of destructor
```

Application: Illustrate Constructor and Destructor methods in a Class

```
class phone:
    qty = 1
    # Initializing (Calling constructor automatically)
    def __init__(self,b,p):
        print("Constructor Initialized")
        self.brand = b
        self.price = p
    # Deleting (Calling destructor)
    def __del__(self):
        print("Destructor destroyed the object")
    def details(self):
        print("Phone brand:",self.brand)
        print("Phone price:",self.price)
        print("Purchase quantity:",self.qty)
# Create/instantiate object ph
ph = phone("Redmi", 15000)
ph.qty = 5
ph.details()
```

CIT Students Only

```
# calling destructor method
del ph
```

Output:

```
Constructor Initialized
Phone brand: Redmi
Phone price: 15000
Purchase quantity: 5
Destructor destroyed the object
```

What is Inheritance? Describe types of Inheritance with examples.

Inheritance:
Inheritance enables the creation of a new class (**derived** or **child** or **sub class**) from an existing class (**base** or **parent** or **super class**). With inheritance, the derived class gains access to all the data members and methods defined in the base class. A derived class may also offer its own method implementation of the base class's methods.

A new class copies all the existing class data attributes and methods without rewriting the syntax in the new class. These new classes are called **derived (child) classes**, and existing classes are called **base (parent) classes**.

Syntax:

```
# define a parent/super class
class BaseClass:
    # data attributes & methods definitions

# inheritance
class DerivedClass(BaseClass):
    # data attributes & methods of BaseClass
    # data attributes & methods of DerivedClass
```

BaseClass - an existing class that will serve as a **parent or super class**
DerivedClass - a **new class** that will inherit from **BaseClass**

CIT Students Only

Application: Inheritance

```
class Person:   # Base class
    def __init__(self, name, college):  # Constructor
        self.name = name
        self.college = college
    def introduce(self):
        print(f"My name is {self.name} and I am at {self.college}")

# Student class inherits from Person class
class Student(Person):  # Derived class
    def __init__(self, name, college, semester):  # Constructor
        super().__init__(name, college)  # Call parent class constructor
        self.semester = semester
    def introduce(self):
        super().introduce()  # Call parent class introduce method
        print(f"I am a student in semester {self.semester}.")

# Create an instance from base class
person = Person("Venkatesh", "CIT")
person.introduce()

# Create an instance from derived class
student = Student("Divya", "CIT", 2)
student.introduce()
```

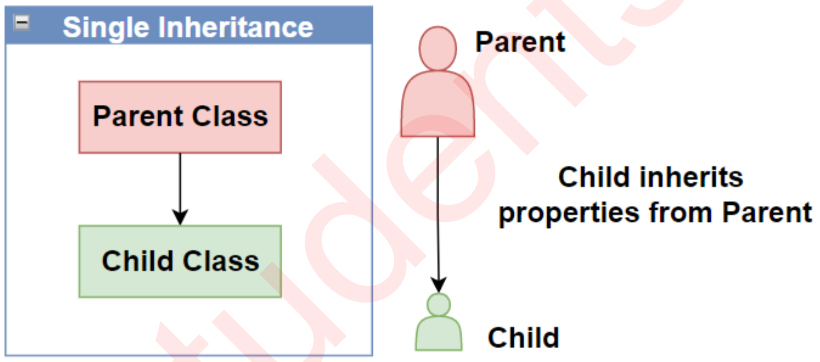
Output:

```
My name is Venkatesh and I am at CIT college.

My name is Divya and I am at CIT college.
I am a student in semester 2.
```

CIT Students Only

Types of Inheritance in Python

1. **Single Inheritance** - This is the simplest form of inheritance in Python. This is also known as Simple Inheritance. In this inheritance, the child class inherits properties from a single-parent class.



Application: Single Inheritance

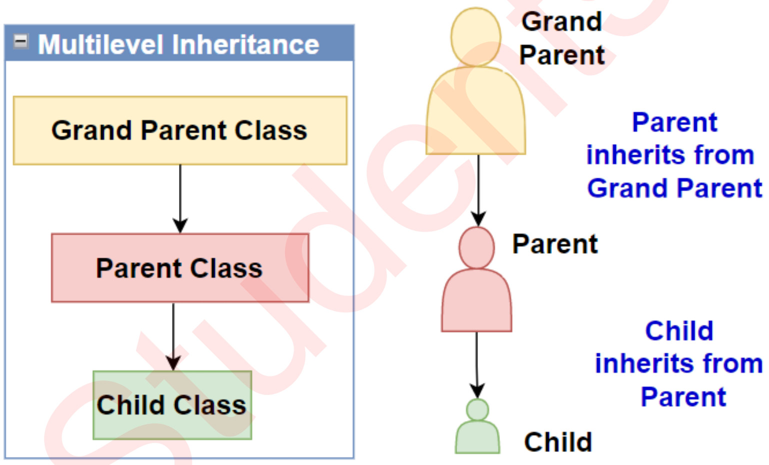
```
# parent class
class Human:
    def hinfo(self):
        print("I am a Human (parent class)")

# child class Male inherits from parent class Human
class Male(Human):
    def minfo(self):
        print("I am a male (child class)")

# main program
m = Male()   # Instance object of child class
m.hinfo() # method in parent class
m.minfo() # method in child class
```

Output:

```
I am a Human (parent class)
I am a male (child class)
```

CIT Students Only

2. **Multilevel Inheritance -** The child class inherits properties from its immediate parent class and the parent class in turn inherits properties from its parent (grandparent) class. The class at each level is in relation to a class at the next level.



Application: Multilevel Inheritance

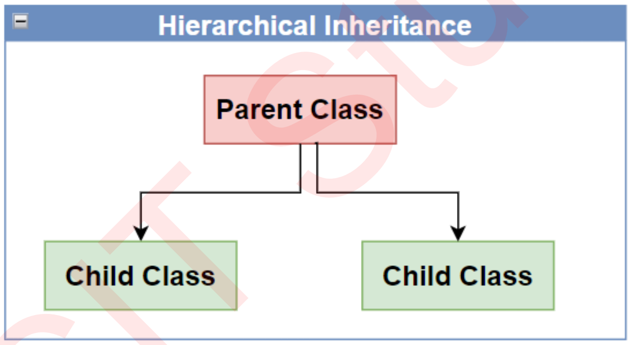
```
# grand parent class
class Human:
    def hinfo(self):
        print("I am a human (grand parent class)")

# parent class
class Person(Human):
    def pinfo(self):
        print("I am a person (parent class)")

# child class Male inherits from parent calss Human
class Male(Person):
    def minfo(self):
        print("I am a male (child class)")

# main program
m = Male()  # Instance object of child class
m.hinfo() # method in grand parent class
```

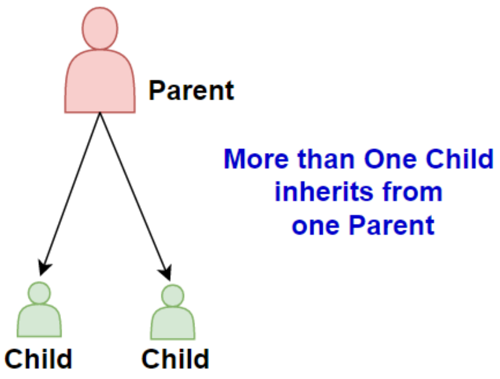
CIT Students Only

```
m.pinfo() # method in parent class
m.minfo() # method in child class
```

Output:
I am an animal (grandparent class)
I am a human (parent class)
I am a male (child class)

3. **Hierarchical Inheritance -** More than one child classes inherit properties from a single parent class. It is quite the opposite of Multiple Inheritance. One parent class has a relation with many child classes.





Application: Hierarchical Inheritance

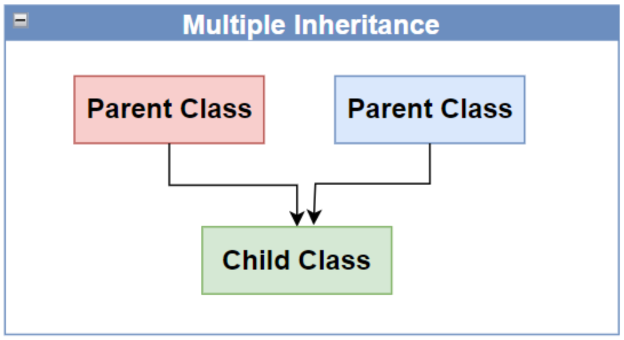
```
# parent class
class Person():
    def pinfo(self):
        print("I am a person (parent class)")

# child class Male inherits from parent calss Human
class Male(Person):
    def minfo(self):
        print("I am a male (child class-1)")

# child class Male inherits from parent calss Human
class Female(Person):
    def finfo(self):
        print("I am a female (child class-2)")
```

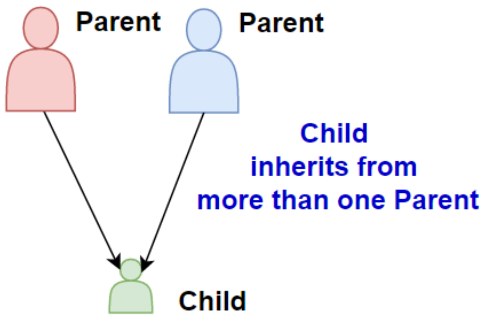
```
# main program
m = Male()   # Instance object of child class Male
m.pinfo() # method in parent class Person
m.minfo() # method in child class Male
print()
f = Female()   # Instance object of child class Female
f.pinfo() # method in parent class Person
f.finfo() # method in child class Female
```

Output:

```
I am a person (parent class)
I am a male (child class-1)

I am a person (parent class)
I am a female (child class-2)
```

4. **Multiple Inheritance -** One child class inherits properties from more than one parent class. It is quite the opposite of Hierarchical Inheritance. One child class has a relation with many parent classes.





CIT Students Only

CIT Students Only

Syntax: Multiple Inheritance

```
# define a parent class1
class BaseClass1:
    # data attributes & methods definitions

# define a parent class2
class BaseClass2:
    # data attributes & methods definitions

# Multiple inheritance
class DerivedClass(BaseClass1, BaseClass2, . . .):
    # data attributes & methods of BaseClass1 & 2
    # data attributes & methods of DerivedClass
```

Application: Multiple Inheritance

```
# parent class
class Father:
    def Finfo(self):
        print("From Father (parent class)")

# parent class
class Mother:
    def Minfo(self):
        print("From Mother (parent class)")

# child class Male inherits from parent calss Human
class Son(Father, Mother):
    def Sinfo(self):
        print("From Son (child class)")

# main program
s = Son()  # Instance object of child class Son
```

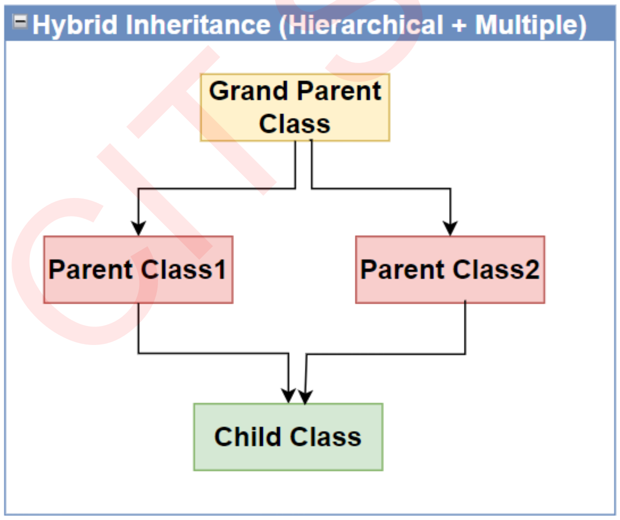
CIT Students Only

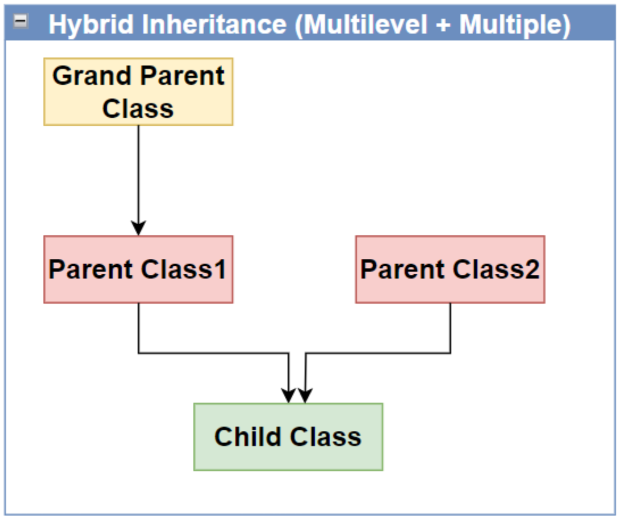
```
s.Finfo() # method in parent class Father
s.Minfo() # method in child class Mother
s.Sinfo() # method in child class Mother
```

Output:

```
From Father (parent class)
From Mother (parent class)
From Son (child class)
```

5. **Hybrid Inheritance** - This is the combination of 2 or more types of inheritance. The hybrid inheritance allows to have many relations among several parent and child classes at different levels.





Application: Hybrid Inheritance (Star structure)

```
# grand parent class
class A:
    def Ainfo(self):
        print("From A (grand parent class)")

# parent class-1 Hierarchical
class B(A):
    def Binfo(self):
```

CIT Students Only

```
        print("From B (parent class-1)")

# parent class-2 Hierarchical
class C(A):
    def Cinfo(self):
        print("From C (parent class-1)")

# child Multiple parents
class D(B,C):
    def Dinfo(self):
        print("From D (child multiple parents)")

d1 = D()
d1.Ainfo()
d1.Binfo()
d1.Cinfo()
d1.Dinfo()
```

Output:

```
From A (grand parent class)
From B (parent class-1)
From C (parent class-1)
From D (child multiple parents)
```

Benefits of Inheritance:

- Better representation of real-world relationships,
- Code reusability without rewriting the same code,
- Add more features to a class without modifying it,
- Create specialized child classes based on more general parent classes.

CIT Students Only

Demonstrate the usage of super(), issubclass() and isinstance() methods in Python.

➢ **super() method:**

The **super()** method is used to refer to the parent class in a subclass. It allows you to invoke methods and access data attributes defined in the superclass from the subclass.

Syntax:

```
super().method()  # call the Name of the method in parent class
```

Application: super() Inheritance

```
class Person:
    def __init__(self, name):
        self.name = name
    def introduce(self):
        print(f"My name is {self.name}")
class Student(Person):
    def __init__(self, name, semester):
        super().__init__(name)  # Call parent method
        self.semester = semester
    def introduce(self):
        super().introduce()  # Call parent method
        print(f"I am a student in semester {self.semester}.")

# Create an instance from derived class
student = Student("Divya", 2)
student.introduce()
```

Output:

```
My name is Divya
I am a student in semester 2.
```

Explanation:

In the above example, the **super().__init__(name)** call in the **Student** class invokes the constructor of the **Person** class, allowing the **name** attribute to be set.

Similarly, **super().introduce()** invokes the **introduce()** method of the **Person** class before printing "I am a student in semester xx.".

CIT Students Only

➢ **issubclass() function:**

The **issubclass()** function is used to check if a class is a subclass of another class.
It returns **True** if the **first class** is a subclass of the **second class**; otherwise, it returns **False**.

Syntax:

```
issubclass(first class, second class)  #
```

➢ **isinstance() function:**

The **isinstance()** function is used to check if an object is an instance of a particular class.
It returns **True** if the **object** is an instance of the **class** or any of its subclasses;
otherwise, it returns **False**.

Syntax:

```
isinstance(object_name, class_name)  #
```

Application: issubclass(), isinstance()

```
class Person:
    pass
class Student(Person):
    pass
class Teacher:
    pass

s1 = Student()
t1 = Teacher()

print(issubclass(Student, Person))  # True
print(issubclass(Teacher, Person))  # False
print(isinstance(s1, Person))  # True
print(isinstance(t1, Person))  # False
```

CIT Students Only

Explanation:

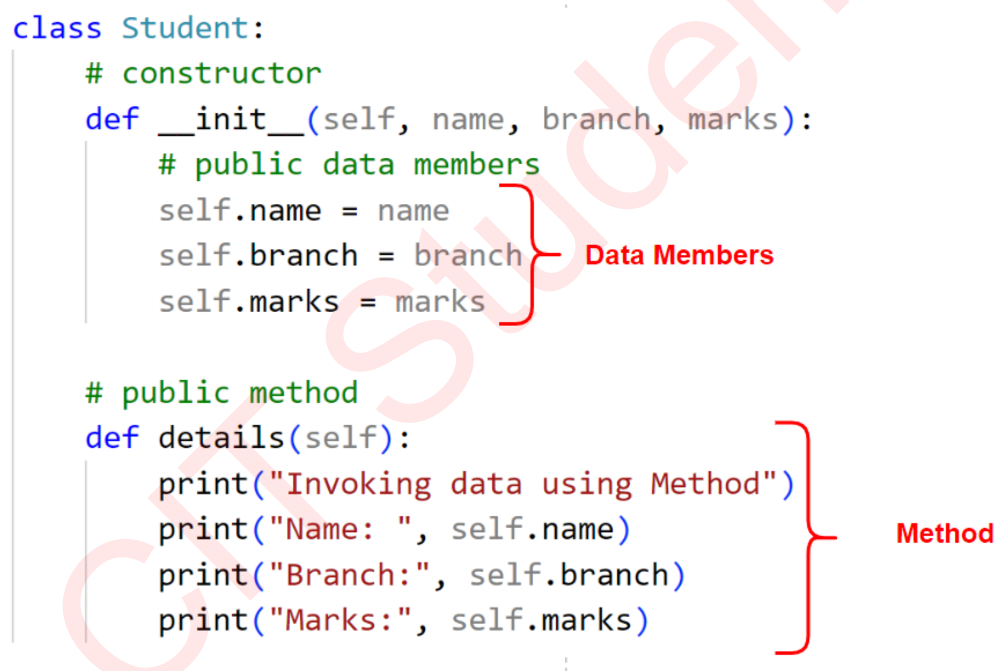
issubclass(Student, Person) returns **True** because **Student** is a subclass of **Person**. However, **issubclass(Teacher, Person)** returns **False** because **Teacher** is not a subclass of **Person**.
isinstance(s1, Person) returns **True** because **s1** is an instance of the **Student** class, which is a subclass of **Person**. However, **isinstance(t1, Person)** returns **False** because **t1** is not an instance of the **Person** class or any of its subclasses.

CIT Students Only

What is Encapsulation? What are access modifiers? Explain them with examples.

(How to Create and Invoke Public, Protected, and Private members of a class?)

Encapsulation: Encapsulation is the practice of wrapping data members and methods within a single unit of class to protect the data from external interference. This adds restrictions to directly access the members. A class is an example of encapsulation as it binds all the data members and methods into a single unit.



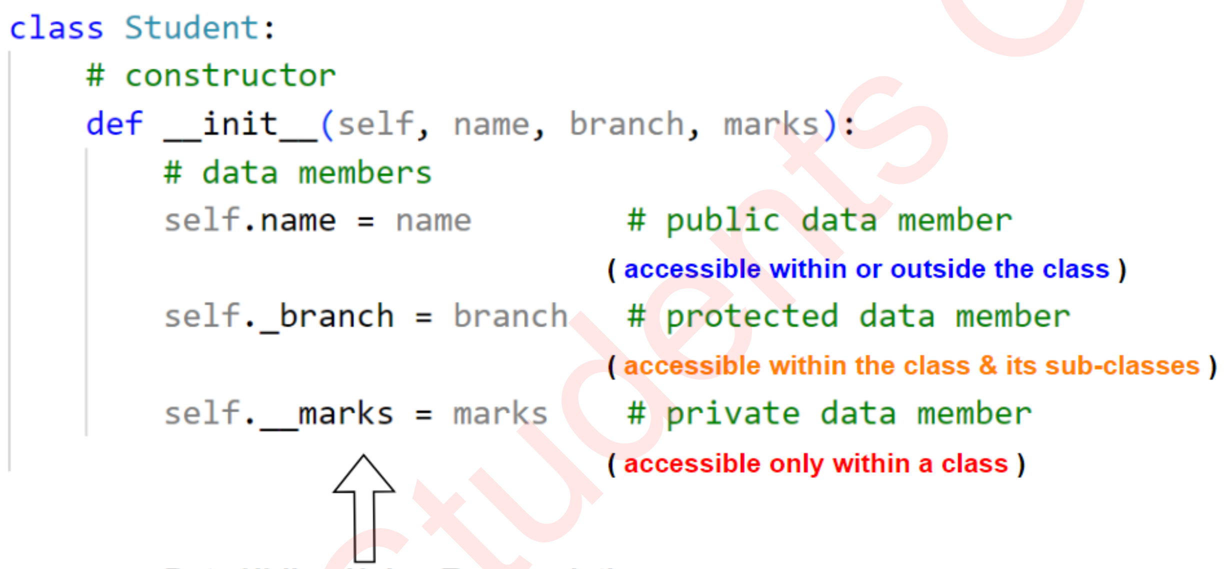
```python
class Student:
    # constructor
    def __init__(self, name, branch, marks):
        # public data members
        self.name = name
        self.branch = branch
        self.marks = marks

    # public method
    def details(self):
        print("Invoking data using Method")
        print("Name:  ", self.name)
        print("Branch:", self.branch)
        print("Marks:", self.marks)
```

Access modifiers:
Encapsulation can be achieved by declaring the data members and methods of a class using access modifiers. The access modifiers limit access to the variables and methods of a class.

In Python, we don't have direct access modifiers like **public**, **private**, **and protected**.
However, Python provides the following 3 types of access modifiers.

Type	Access Modifiers	Definition
Public access	normal members	Accessible **anywhere from outside the class**
Protected access	Members preceded by **_ (single underscore)**	Accessible **within the class and its sub-classes**. ● The methods of the same or sub-classes can only change/invoke protected members.
Private access	Members preceded by **__ (double underscore)**	Accessible **within the class**. ● The methods of the same class can only change/invoke private members.

CIT Students Only



```
class Student:
    # constructor
    def __init__(self, name, branch, marks):
        # data members
        self.name = name            # public data member
                                   ( accessible within or outside the class )
        self._branch = branch       # protected data member
                                   ( accessible within the class & its sub-classes )
        self.__marks = marks        # private data member
                                   ( accessible only within a class )
```

Data Hiding Using Encapsulation

➢ **Public access:**
In Python, class variables are by default public; it means, they can be accessed and modified from both within and outside the class.

Application: public access

```
class Student:
    # constructor
    def __init__(self, name, branch, marks):
        # public data members
        self.name = name
        self.branch = branch
        self.marks = marks

    # public method
    def details(self):
        print("Invoking data using Method")
        print("Name:  ", self.name)
        print("Branch:", self.branch)
        print("Marks:", self.marks)
# creating object of the class
std = Student('Dinesh', 'CSE', 70)
```

CIT Students Only

```
# calling public method of the class
std.details()

# direct access to public data members
print("Invoking data directly")
print(std.name, std.branch, std.marks)
```

Output:

```
Invoking data using Method
Name:  Dinesh
Branch: CSE
Marks: 70
Invoking data directly
Dinesh CSE 70
```

➢ **Protected access:**

Protected members are accessible **within the class and also available to its sub-classes**.
To define a protected member, prefix the member name with a single underscore _.

During inheritance, the Protected data members are useful if we want to provide access only to the **current class and sub-classes** but not outside the class.

Application: protected access _ (double underscore members)

```
class College:
    def __init__(self):
        self._branch="CSE"      # protected data member
class Student(College):
    def __init__(self, name):
        self.name = name        # public data member
        College.__init__(self)
    def show(self):
        print("Name:  ", self.name)
        print("Branch:", self._branch)
# creating object of the class
std1 = Student('Dinesh')
```

CIT Students Only

```
# Direct access to protected data member
print("Direct access to protected member")
print("Branch:", std1._branch)
# Using public method to access protected member
print("Using Method to access protected member")
std1.show()
```

Output:

Direct access to protected member
Branch: CSE
Using Method to access protected member
Name: Dinesh
Branch: CSE

➢ **Private access:**

In Python, we can create private members to secure them. Private members are accessible only within the class, and we can't access them directly from the class objects. This provides a way to encapsulate data and restrict direct access to the data from external code.

To define a private member, Prefix the member name with double underscores (__).
Note: This internally changes the variable's name to include the class name (Mangling), making it private and harder to access from outside the class.

Accessing Private Members:
We cannot directly access the private members. The following 2 methods are used to access the private members.

1. Using **public methods** to access private members
2. Using **Name MANGLING** to access private members - We can directly access private and protected variables from outside of a class through name mangling. The name mangling is created on an identifier by adding one leading underscore and two trailing underscores, like this **_classname__dataMember**, where **classname** is the current class, and **dataMember** is the private variable name.
   Syntax:

   object . _classname__dataMember
   Ex: print(std **.** _Student__marks)

CIT Students Only

Application: private access __ (double underscore members)

```
class Student:
    # constructor
    def __init__(self, name, branch, marks):
        # data members
        self.name = name          # public data member
        self._branch = branch     # protected data member
        self.__marks = marks      # private data member
    # public method
    def details(self):
        print("Name:  ", self.name)
        print("Branch:", self._branch)
        print("Marks:", self.__marks)
# creating object of the class
std = Student('Dinesh', 'CSE', 70)
# NO Direct access to private data members
""" AttributeError: 'Student' object has no attribute '__marks'   """
#print("Marks:", std.__marks)

# Accessing Private members
# 1. Using public method to invoke private & protected members
print("Invoking data using Method")
std.details()
# 2. Using Name MANGLING to invoke private members
print("Invoking data using Name Mangling")
print("Marks:", std._Student__marks)
```

Output:

```
Invoking data using Method
Name:  Dinesh
Branch: CSE
Marks: 70
Invoking data using Name Mangling
Marks: 70
```

CIT Students Only

Demonstrate Data modeling using classes with an example.

A data model means arranging various related data elements in a format that is more clean and readable or processed further. Multiple classes can be used in Python to structure the data elements with data attributes ad methods. The following application demonstrates how to model and process Employee and Manager data.

Application: Data modeling using classes

```
class Employee:
    def __init__(self, emp_id, name, dept,sal,bonus):
        self.emp_id = emp_id
        self.name = name
        self.dept = dept
        self.sal = sal
        self.bonus = bonus
    def get_salary(self):
        return (self.sal+self.bonus)
    def display_info(self):
        print(f"Employee ID: {self.emp_id}")
        print(f"Name: {self.name}")
        print(f"Department: {self.dept}")
        print(f"Total Salary: {self.get_salary()}")
class Manager(Employee):
    def __init__(self, emp_id, name, dept,sal, bonus, team_size):
        super().__init__(emp_id, name, dept,sal,bonus)
        self.team_size = team_size
    def display_info(self):
        super().display_info()
        print(f"Team Size: {self.team_size}")
# Create instances of Employee and Manager
employee1 = Employee("E001", "Chandrika", "HR",10000,500)
employee2 = Employee("E002", "Chaitanya", "IT",15000,500)
manager1 = Manager("M001", "Sree Veda", "Finance",17000,500, 10)
manager2 = Manager("M002", "Jyothi", "Support",9000,500, 5)
# Display information of employees and managers
```

CIT Students Only

```
employee1.display_info()
print("-----------------------")
employee2.display_info()
print("-----------------------")
manager1.display_info()
print("-----------------------")
manager2.display_info()
```

Output:

```
Employee ID: E001
Name: Chandrika
Department: HR
Total Salary: 10500
-----------------------
Employee ID: E002
Name: Chaitanya
Department: IT
Total Salary: 15500
-----------------------
Employee ID: M001
Name: Sree Veda
Department: Finance
Total Salary: 17500
Team Size: 10
-----------------------
Employee ID: M002
Name: Jyothi
Department: Support
Total Salary: 9500
Team Size: 5
```

Explanation:

Two classes are defined in this data modeling program. **Employee** and **Manager**.

- The **Employee class** contains data attributes **emp_id, name, dept, sal,** and **bonus**. It also has methods like **display_info()** to display employee information and **get_salary()** to calculate the salary.
- The **Manager class** is a subclass of Employee and inherits its attributes and methods. It adds an additional data attribute **team_size** specific to managers and **overrides the display_info()** method to include the team size information along with the inherited employee information.
- Later, we created **instances of Employee** and **Manager classes** and displayed their information using the **display_info()** method.

CIT Students Only

What is Polymorphism? Explain Method Overriding and Method Overloading with examples.

Polymorphism:
Polymorphism basically means having many forms. In Python, it means the ability to use an object of a derived class wherever an object of the base class is expected.

Polymorphism is of two types
1. **Compile-time Polymorphism (Overloading) and**
2. **Run-time Polymorphism (Overriding).**

[**Note**: Python does not support method overloading or compile-time polymorphism. If there are multiple methods with the same name in a class, the last method defined will override the earlier one in a class.]

Polymorphism in Python is achieved through the following techniques.
A. **Method Overriding -** using the same function name with different definitions in inherited classes
B. **Method Overloading -** a method used for different numbers of arguments. (Not supported directly in Python; but can be achieved with default & variable-length keyword arguments)
C. **Operator Overloading -** allows the same operator to have different meanings according to the **data type of the given values**

A. Method Overriding

Method Overriding is redefining a parent class method in the derived class. Overriding requires inheritance for implementation. In Python, Polymorphism lets us define methods in the child class that have the same name as the methods in the parent class. In inheritance, the child class inherits the methods from the parent class. However, it is possible to modify a method in a child class that it has inherited from the parent class. This is particularly useful in cases where the method inherited from the parent class doesn't quite fit the child class. In such cases, we re-implement the method in the child class. This process of re-implementing a method in the child class is known as **Method Overriding**.

The overriding method in the subclass provides a different or specialized implementation. This allows the subclass to provide its own behavior while still maintaining the same method signature in the superclass.

Application-1: Method Overriding in Polymorphism

```
class Shape:
    def draw(self):
        print("Drawing a shape.")
class Circle(Shape):
    def draw(self):
        print("Drawing a circle.")
```

CIT Students Only

```
s = Shape()
c = Circle()
s.draw()  # Drawing a shape.
c.draw()  # Drawing a circle.
# c.draw() calls draw() method in Circle class. The draw() method in the
# Circle class is overriding the draw() method in the Shape class
```

Explanation:
The **Shape** class has **draw()** method.
The **Circle** class inherits from **Shape** and **overrides the draw() method** to provide its own implementation.
When we call the **draw() method** on a **Shape object**, it normally executes the method defined in the Shape class.
However, when we call the **draw() method** on a **Circle object**, it executes the method defined in the Circle class, overriding the implementation in the Shape class.

Application-2: Method Overriding in Polymorphism

```
class Animals:
    def Intro(self):
        print("\nWe have many types of animals")
    def Eat(self):
        print("Some animals eat Non-Veg & some eat only Veg")
        #print("Some animals are Carnivorous & some are Herbivorous")
class Tigers(Animals):
    def Eat(self):
        print("Tigers eat meat.")
class Cows(Animals):
    def Eat(self):
        print("Cows do not eat meat.")
obj_tiger = Tigers()
obj_cow = Cows()
obj_tiger.Intro()
obj_tiger.Eat()
obj_cow.Intro()
obj_cow.Eat()
```

CIT Students Only

Output:

```
We have many types of animals
Tigers eat meat.

We have many types of animals
Cows do not eat meat.
```

Explanation:

- **Eat()** method is defined in all classes **Animals, Tigers, Cows**
- When called from Tigers object, **Eat() method in Tigers class overrides Eat() method in Animals class.**
- When called from Cows object, **Eat() method in Cows class overrides Eat() method in Animals class.**

B. Method Overloading

Method overloading refers to defining multiple methods with the same name but with **different numbers of parameters**.

In Python, method overloading is not supported directly, as strict data type declarations for method arguments are not required in Python. However, we can achieve similar behavior using default arguments or variable-length argument lists.

Application: Indirect - Method Overloading in Polymorphism (Emulation)

```python
class Calculator:
    def add(self, a, b, c=0):
        return a + b + c

calc = Calculator()
print(calc.add(3, 5))  # Output: 8
print(calc.add(3, 5, 7))  # Output: 15
```

CIT Students Only

C. Operator Overloading

Does Python Support Operator overloading? Justify with an example program.

Yes, Python does support Operator Overloading. **Operator overloading** is Redefining or changing the default behavior of built-in operators depending on the operands (values) that we use. This means we can use the same operator for multiple purposes.

Application-1: Operator Overload in Polymorphism (int, str, list objects)
The + operator will do arithmetic addition on two numbers or concatenate 2 strings or merge 2 lists.

```
# Operator Overload using built-in int, str, list objects
print(50 + 25)                  # Add 2 numbers
print('Software ' + 'Engineer')     # Concatenates 2 strings
# Merges two lists
print([501, 502, 503] + ['Rani', 'Suj', 'Madhu'])
```

Output:

```
75
Software Engineer
[501, 502, 503, 'Rani', 'Suj', 'Madhu']
```

Example-2: Operator overload using two custom objects (user-defined objects)
We cannot directly add 2 custom objects; it will throw a TypeError.
However, we can overload **+ operator** to work with custom objects with the **MAGIC** method **__add__()**.

- When we use the + operator, the magic method __add__() is automatically invoked.
- Internally + operator is implemented by using __add__() method.
- We have to override this method in our class if you want to add two custom objects.

(Note: See the Reference section at the end for all available MAGIC methods in Python.)

CIT Students Only

Application-2: Operator Overloading in Polymorphism using Two Custom Objects

```
# Operator Overload using custom objects
"""
Two objects cannot be added directly,
but we can overload + operator using __add__() magic method
"""
class Section:
    def __init__(self, s):
        self.students = s
    def __add__(self, other):
        return self.students + other.students

# Create two objects
A = Section(60)
B = Section(61)

# Add two objects
print("Total students: ", A + B)
```

```
# OR Actual addition of objects is done as follows
print("Total students: ", Section.__add__(A, B))
print("Total students: ", A.__add__(B))
```

Output:

```
Total students:  121
Total students:  121
Total students:  121
```

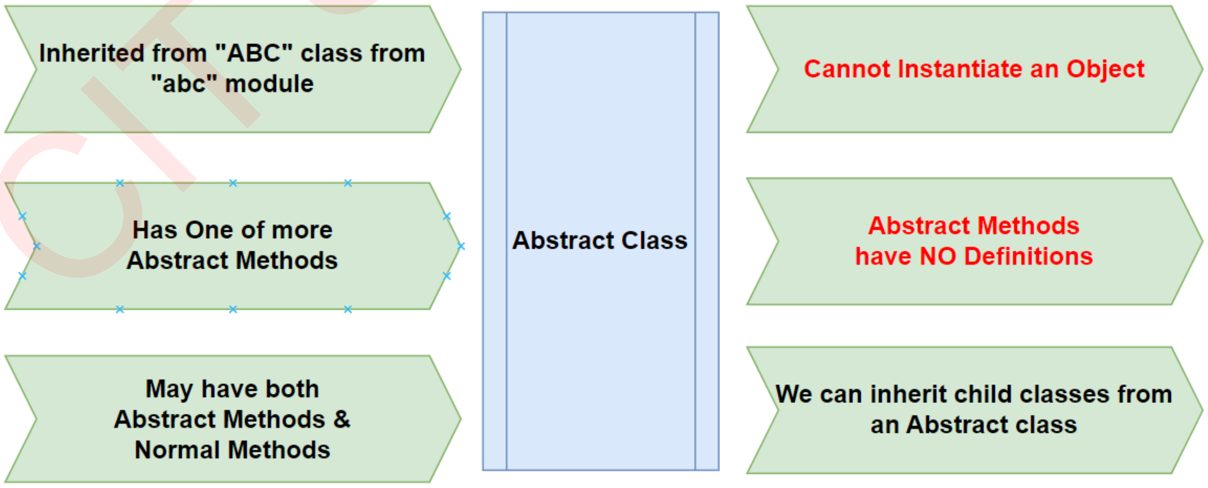
CIT Students Only

What is Abstraction in Python? Explain Types and Properties of Abstraction with an Example.

Abstraction is one of the 4 core principles of OOP languages. It is the process of handling complexity by hiding unnecessary information from the user. This means users know **"what the method is"** (name of the method and how to use the method) but do not know **"how that method functions"**. Hence, Abstraction helps to hide the complexity of coding from the users.

Types and Properties of Abstraction:

1. **Abstract Classes -** Abstract classes are used to create a blueprint for other classes.
   a. A class with one or more abstract methods is called an abstract class.
   b. An abstract class can have both abstract methods and concrete(normal) methods.
   c. We ***cannot create/instantiate an object using an Abstract class*** (because methods have NO definition)
   d. We can inherit child classes from an abstract class
2. **Abstract Methods**
   a. Abstract methods do not have a definition or implementation in the Abstract class.
   b. Abstract methods are REDEFINED with definitions in the derived class.
   c. Thus the method in the child class overrides the abstract method in the parent class.


Inherited from "ABC" class from "abc" module
Has One of more Abstract Methods
May have both Abstract Methods & Normal Methods
Abstract Class
Cannot Instantiate an Object
Abstract Methods have NO Definitions
We can inherit child classes from an Abstract class

➢ **Abstract class:** A class that contains one or more abstract methods.
➢ **Abstract method:** A method that has a declaration but has NO implementation in Abstract class.

Requirements:

- Python provides the "**abc**" module to use the abstraction in the Python program.
- We must import the "**abc**" module:

```
from abc import ABC, abstractmethod
```

- Abstract class is inherited from "**ABC**" class (Abstract Base Class) from "**abc**" module.

CIT Students Only

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

Syntax:

```
from abc import ABC, abstractmethod # Abstract Base Classs
class abstract_class_name(ABC):
    @abstractmethod
    def abstract_method_name(self, other parameters):
        pass
```

Application: Abstraction of OOPs in Python

```
from abc import ABC, abstractmethod

# Abstract base class for shapes
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass
# Concrete subclass representing a rectangle
class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width
    def area(self):
        return self.length * self.width
    def perimeter(self):
        return 2 * (self.length + self.width)
# Concrete subclass representing a circle
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
```

CIT Students Only

```
    def area(self):
        return 3.1415 * self.radius * self.radius
    def perimeter(self):
        return 2 * 3.1415 * self.radius

# Create instances of the classes and demonstrate abstraction
r = Rectangle(5, 3)
c = Circle(4)

print("Rectangle:")
print("Area:", r.area())
print("Perimeter:", r.perimeter())
print()

print("Circle:")
print("Area:", c.area())
print("Perimeter:", c.perimeter())
```

Output:

```
Rectangle:
Area: 15
Perimeter: 16

Circle:
Area: 50.264
Perimeter: 25.132
```

Explanation:
The methods **area()** and **perimeter()** in **Shape** class are Abstract methods
These methods are redefined in each of the subclasses **Rectangle** and **Circle**
Hence, actual definitions of **area()** and **perimeter()** are hidden from users.

CIT Students Only

What are built-in class attributes in Python? Explain adding and retrieving dynamic class attributes.

In Python, class attributes are variables that are associated with a class rather than with instances (objects) of the class. They are shared among all instances of the class and can be accessed using the class name or any instance of the class.

Built-in class attributes:
Built-in class attributes are predefined attributes that exist for every class in Python. Here are some commonly used built-in class attributes:

1. **__name__**: name of the class as a string.
2. **__module__**: module in which the class is defined.
3. **__dict__**: a dictionary that contains the class's namespace.
4. **__doc__**: the docstring (documentation string) for the class.

Dynamic Class Attribute/variable:
In Python, you can dynamically add class variables at runtime, even if they are not initially defined in the class definition. Simply **assign a value to a variable using the class name.**

Syntax:

```
class_name.variable = value
```

Dynamic Instance Attribute/variable:
In Python, you can dynamically add instance variables at runtime, even if they are not initially defined in the class definition. Simply **assign a value to a variable using the object name**.

Syntax:

```
object_name.variable = value
```

Application: Dynamic Class Attributes or Instance Attributes

```
# Adding Dynamic Attributes to a Class
class Movies:
    name = "Superman"
```

CIT Students Only

```
Movies.year = 2023  # Dynamic Class Attribute added at runtime
m1=Movies()
m1.sales = 2500000  # Dynamic Instance Attribute added at runtime

print("Name: ", m1.name)
print("Year: ", m1.year)
print("Sales: ", m1.sales)
del(m1.sales)       # deletes the dynamic attribute
```

Output:

```
Name:  Superman
Year:  2023
Sales:  2500000
```

CIT Students Only

Demonstrate the design of a case study with classes.

We can use classes in Python to design solutions to a given problem. The following case study demonstrates the design of a solution using classes.

Case Study-1: Design with One Class

Aim: Design a Python program using a class to calculate total cost of purchase of a given product with discounts based on number of items ordered. The details are given below.
(Note: Lab program #28)

Write a class called **Product**.

- The class should have fields called **name, amount**, and **price**, holding the product's name, the number of items of that product in stock, and the regular price of the product.
- There should be a method **get_price** that **receives the number of items to be bought** and **returns the cost of buying that many items**, where
    - the **regular price** is charged for **orders of less than 10 items**,
    - a **10% discount** is applied for **orders of between 10 and 99 items**, and
    - a **20% discount** is applied for **orders of 100 or more items**.

Explanation: Design Product Case Study

1. **Product**
    a. **name** (Name of the product)
    b. **amount** (Number in stock)
    c. **price** (MRP of product)
2. Method **get_price()**
    a. Receive argument:
        i. **n** (Number products to purchase)
    b. Process:
        i. If n<10, **Regular Price**
        ii. If 10 <= n <= 99, **10% discount** on price
        iii. If n >= 100, **20% discount** on price
    c. Output:
        i. **cost** (Total cost of buying n number of items)

CIT Students Only

Case Study-1: Design Product Operations using the class

```
class product:
    def __init__(this,name,items,price):
        this.name=name
        this.items=items
        this.price=price
    def get_price(this,n):
        if n<10:
            cost=n*this.price
            print("Regular price:",cost)
            print("Discount:",cost)
        elif n>=10 and n<100:
            cost=n*this.price
            discount=round((cost*10)/100)
            costAfterDiscount = cost-discount
            print("Regular price:",cost)
            print("Discount:",costAfterDiscount)
        else:
            cost=n*this.price
            discount=round((cost*20)/100)
            costAfterDiscount = cost-discount
            print("Regular price:",cost)
            print("Discount:",costAfterDiscount)
pname = input("Enter product name: ")
n = int(input("Number of items to be bought: "))
pprice = int(input("Enter the price of single product: "))
p=product(pname,n,pprice)
p.get_price(n)
```

Output:

```
Enter product name: Car
Number of items to be bought: 15
Enter the price of single product: 15000
Regular price: 225000
Discount: 202500
```

CIT Students Only

Case Study-2: Design with Multiple Classes (using Multiple Inheritance)

Aim: Design a Python application using multiple inheritance to show the details of student, marks, and average of a given subject.

Explanation: Design Student Case Study

We use three classes: **Student, Subject**, and **MarkSheet**.

1. The **Student** class represents a student. It contains,
   a. Attributes
      - **name**
      - **age**
      - **roll_number**
   b. Methods
      - **display_student_info()** to display the student's information.
2. The **Subject** class represents a subject. It contains,
   a. Attributes
      - **subject_name**
      - **max_marks**.
   b. Methods
      - **display_subject_info()** to display the subject's information.
3. The **MarkSheet** class inherits from both the **Student** and **Subject** classes using multiple inheritance. It represents the mark sheet of a student for a particular subject. It contains
   a. Attributes
      - **obtained_marks**
   b. Methods
      - **display_marksheet()** to display the mark sheet information.
      - **calculate_percentage()** to calculate the percentage obtained by the student in the subject.

In the application section, we create an instance of the **MarkSheet** class called **student1** with the provided information. We then display the mark sheet using the **display_marksheet()** method and calculate the percentage using the **calculate_percentage()** method. We repeated the same process for an instance **student2. Hence, we can apply this design of classes for any number of student instances.**

CIT Students Only

Case Study-2: Design Student Operations using Classes

```
class Student:
    def __init__(self, name, age, roll_number):
        self.name = name
        self.age = age
        self.roll_number = roll_number
    def display_student_info(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Roll Number: {self.roll_number}")

class Subject:
    def __init__(self, subject_name, max_marks):
        self.subject_name = subject_name
        self.max_marks = max_marks

    def display_subject_info(self):
        print(f"Subject Name: {self.subject_name}")
        print(f"Maximum Marks: {self.max_marks}")

class MarkSheet(Student, Subject):
    def __init__(self, name, age, roll_number, subject_name, max_marks,
                                                        obtained_marks):
        Student.__init__(self, name, age, roll_number)
        Subject.__init__(self, subject_name, max_marks)
        self.obtained_marks = obtained_marks

    def display_marksheet(self):
        print("________________________")
        self.display_student_info()
        self.display_subject_info()
        print(f"Obtained Marks: {self.obtained_marks}")
```

CIT Students Only

```
    def calculate_percentage(self):
        percentage = (self.obtained_marks / self.max_marks) * 100
        return percentage

# Apply for student1 object
student1 = MarkSheet("Pras Lakshmi", 19, "22HT1A587", "Python", 100, 75)
student1.display_marksheet()
percentage = student1.calculate_percentage()
print(f"Percentage: {percentage}%")

# Apply for student2 object
student1 = MarkSheet("Yashwant", 19, "22HT1A581", "Maths", 100, 87)
student1.display_marksheet()
percentage = student1.calculate_percentage()
print(f"Percentage: {percentage}%")
```

Output:

```
________________________
Name: Pras Lakshmi
Age: 19
Roll Number: 22HT1A587
Subject Name: Python
Maximum Marks: 100
Obtained Marks: 75
Percentage: 75.0%

________________________
Name: Yashwant
Age: 19
Roll Number: 22HT1A581
Subject Name: Maths
Maximum Marks: 100
Obtained Marks: 87
Percentage: 87.0%
```

CIT Students Only

Case Study-3: Design a Python Application for BANKING Operations using Classes

Aim: Design a Python application for basic BANKING operations such as Login, Deposit and Withdrawal using classes.

Explanation: Design BANKING Case Study
Classes: Bank and **Transactions**
The **Bank** and **Transactions** classes represent customer log in, deposit, and withdraw money. These classes consist of the following Data members and Method members.

Class: Bank

Data members	Method members
● **balance -** data variable that holds the available balance	● **Login( ) -** Checks for 3 tries for a correct pin; if not then exit

Class: Transactions

Data members	Method members
none	● **Deposit( )** - Input the amount to deposit; Increase the balance accordingly ● **Withdraw( )** - Input the amount to withdraw; Check for sufficient balance; if so, then decrement the balance; if not, then show “Insufficient balance” ● **Show_Balance( ) -** Shows the net available balance ● **Logout( ) -** Exit from the BANKING application

Note: As of Python 3.10, the **match-case** statement is available, similar to the switch-case in C/C++.

Case Study-3: Design BANKING Operations using classes & math-case statement in Python

```
import sys
class Bank:
    def __init__(self):
        self.balance = 0
        print("Welcome to BANKING")
    def Login(self):
        tries = 0
        while tries <3:
            ac = input("Account#(12345): ")
            pin= input("PIN#    (9797) : ")
```

CIT Students Only

```
            if ac=="12345" and pin == "9797":
                print("\nLogged in Successfully")
                return True
            else:
                print("\nInvalid A/C# or PIN")
                tries += 1
        else:
            print("Too many incorrect tries")
            print("Your account will be blocked for 24 hours")
            sys.exit()
class Transactions(Bank):
    def Menu(self):
        print("___________")
        print("1. Deposit")
        print("2. Withdraw")
        print("3. Balance")
        print("4. Logout")
        choice = int(input("Select Your Choice:"))
        match(choice):
            case 1:
                self.Deposit()
                self.Menu()
            case 2:
                self.Withdraw()
                self.Menu()
            case 3:
                self.Show_Balance()
                self.Menu()
            case 4:
                self.Logout()
            # _ is the default case if no given case is matched
            case _:
                print("Incorrect choice.")
                self.Menu()
    def Deposit(self):
```

CIT Students Only

```
        amount = float(input("Enter deposit amount: "))
        self.balance += amount
        print("\nDeposited amount: ",amount)
    def Withdraw(self):
        amount = float(input("Enter withdrawal amount: "))
        if self.balance>=amount:
            self.balance -= amount
            print("\nWithdrawn amount: ",amount)
        else:
            print("\nInsufficient balance")
    def Show_Balance(self):
        print("\nNet available balance: ",self.balance)
    def Logout(self):
        print("Logged out!")
        sys.exit()
customer_obj = Transactions()
customer_obj.Login()
customer_obj.Menu()
```

Output-1:

```
Welcome to BANKING
Account#(12345): 12345
PIN#    (9797) : 9797
Logged in Successfully
____________
1. Deposit
2. Withdraw
3. Balance
4. Logout
Select Your Choice:1
Enter deposit amount: 7000
Deposited amount:  7000.0
____________
1. Deposit
2. Withdraw
3. Balance
4. Logout
Select Your Choice:2
```

CIT Students Only

```
Enter withdrawal amount: 2000
Withdrawn amount:  2000.0
______________
1. Deposit
2. Withdraw
3. Balance
4. Logout
Select Your Choice:1
Enter deposit amount: 10000
Deposited amount:  10000.0
______________
1. Deposit
2. Withdraw
3. Balance
4. Logout
Select Your Choice:3
Net available balance:  15000.0
______________
1. Deposit
2. Withdraw
3. Balance
4. Logout
Select Your Choice:4
Logged out!
```

Output-2:

```
Welcome to BANKING
Account#(12345): 1234
PIN#    (9797) : 6789

Invalid A/C# or PIN
Account#(12345): 3456
PIN#    (9797) : 9809

Invalid A/C# or PIN
Account#(12345): 34567
PIN#    (9797) : 09

Invalid A/C# or PIN
Too many incorrect tries
Your account will be blocked for 24 hours
```

CIT Students Only

Reference

Magic Methods in Python
The magic methods are built-in methods in Python and are used to perform operator overloading. The below list of magic methods overloads the mathematical operators, assignment operators, and relational operators in Python.

Operator Name	Symbol	Magic method
Addition	+	__add__(self, other)
Subtraction	-	__sub__(self, other)
Multiplication	*	__mul__(self, other)
Division	/	__div__(self, other)
Floor Division	//	__floordiv__(self,other)
Modulus	%	__mod__(self, other)
Power	**	__pow__(self, other)
Increment	+=	__iadd__(self, other)
Decrement	-=	__isub__(self, other)
Product	*=	__imul__(self, other)
Division	/+	__idiv__(self, other)
Modulus	%=	__imod__(self, other)
Power	**=	__ipow__(self, other)
Less than	<	__lt__(self, other)
Greater than	>	__gt__(self, other)
Less than or equal to	<=	__le__(self, other)
Greater than or equal to	>=	__ge__(self, other)
Equal to	==	__eq__(self, other)
Not equal	!=	__ne__(self, other)

CIT Students Only

Application: Polymorphism Operator Overloading
Using __mul__ magic method for Multiplication of objects

```python
# Operator Overload using custom objects - Polymorphism
"""
Two objects cannot be multiplied
but we can overload * operator using __mul__ magic method.
"""
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
    def __mul__(self, wrk):
        return self.salary * wrk.days

class Work:
    def __init__(self, name, days):
        self.name = name
        self.days = days

emp = Employee("Sowmya", 100)
wrk = Work("Sowmya", 30)
print("Total Salary: ", emp * wrk)
```

Output:

```
Total Salary:  3000
```

CIT Students Only

Application: Polymorphism Method Overriding - Animals

```
class Animals:
    def Intro(self):
        print("\nWe have many types of animals")
    def Eat(self):
        print("Some animals eat Non-Veg & some eat only Veg")
        #print("Some animals are Carnivorous & some are Herbivorous")
class Tigers(Animals):
    def Eat(self):
        print("Tigers eat meat.")
class Cows(Animals):
    def Eat(self):
        print("Cows do not eat meat.")
obj_tiger = Tigers()
obj_cow = Cows()

obj_tiger.Intro()
obj_tiger.Eat()

obj_cow.Intro()
obj_cow.Eat()
```

Output:

```
We have many types of animals
Tigers eat meat.
We have many types of animals
Cows do not eat meat.
```

Explanation:

- Eat() method is in all base & derived classes: Animals , Tigers, Cows
- obj_tiger.Eat() executes Eat() method in Tigers class by Overriding Eat() method in Animals class
- obj_cow.Eat() executes Eat() method in Cows class by Overriding Eat() method in Animals class

CIT Students Only

#Application: Data Modeling using Classes

```
class Students:
    def __init__(self, std_id, name, branch, Py, Math, DS):
        self.std_id = std_id
        self.name = name
        self.branch = branch
        self.Py = Py
        self.Math = Math
        self.DS = DS
    def Total(self):
        return (self.Py+self.Math+self.DS)
    def Show(self):
        print("Reg# ",self.std_id)
        print("Name ",self.name)
        print("Branch ", self.branch)
        print("Total ",self.Total())

class Players(Students):
    def __init__(self, std_id, name, branch, Py, Math, DS,sport):
        super().__init__(std_id, name, branch, Py, Math, DS)
        self.sport = sport
    def Show(self):
        super().Show()
        print("Sport ",self.sport)
        print("---------------")
Vobj = Players("5B6", "Vasanthi", "CSE",100,80,70, "Kabadi")
Vobj.Show()
Sobj = Players("591", "Tirupathiah", "CSE",90,70,50, "Cricket")
Sobj.Show()
```

Output:

```
Reg#  5B6
Name  Vasanthi
Branch  CSE
```

CIT Students Only

```
Total  250
Sport  Kabadi
---------------
Reg#  411
Name  Tirupathiah
Branch  CSE
Total  210
Sport  Cricket
---------------
```

Application: Added Dynamic Class Attribute & Instance Attribute to a Class

```
class Music:
    singer = "Aniruth"

# Added Dynamic Class Attribute
Music.song = "Tere Vaste"
naveen = Music()

# Added Dynamic Instance Attribute
naveen.price = 100
print("naveen Object:")
print(naveen.singer, naveen.song, naveen.price)

print("raj Object:")
raj = Music()
print(raj.singer, raj.song)
# print(raj.singer, raj.song, raj.price)
# raj.price will show an ERROR because
# "price" is an instance attribute for naveen object
```

Output:

```
naveen Object:
Aniruth Tere Vaste 100
raj Object:
Aniruth Tere Vaste
```

CIT Students Only

Application: Polymorphism Method Overriding - Animals Lions Rabits

```
class Animals:
    def Introduction(self):
        print("We have many types of Animals")
    def Eat(self):
        print("Some are Carnivorous & Some are Herbivorous")
class Lions(Animals):
    def Eat(self):
        print("Lions are Carnivorous")

class Rabits(Animals):
    def Eat(self):
        print("Rabits are Herbivorous")

l = Lions()
r = Rabits()

l.Introduction()
l.Eat()

r.Introduction()
r.Eat()
```

Output:

We have many types of Animals (from parent class - Animals)
Lions are Carnivorous (from child class - Lions)

We have many types of Animals (from parent class - Animals)
Rabits are Herbivorous (from child class - Rabits)

CIT Students Only

Application: Encapsulation - Accessing Private & Protected Members

```
class Banking:
    ac = 12345       #Public member
    __bal = 3500000  #Private member
    _city = "Guntur" #Protected member

    def show(self):
        print("Accessing Private member from Public Method")
        print(self.ac,self.__bal)
class Customers(Banking):
    pass

obj = Customers()
# Accessing Private member using Public Method
obj.show()

# Accessing Private member using Name Mangling
print("Accessing Private member using Name MANGLING")
print(obj.ac,obj._Banking__bal)

# Accessing Protected member from child object
print("Access Protected member from Child Object")
print(obj._city)
```

Output:

```
Accessing Private member from Public Method
12345 3500000

Accessing Private member using Name MANGLING
12345 3500000

Access Protected member from Child Object
Guntur
```