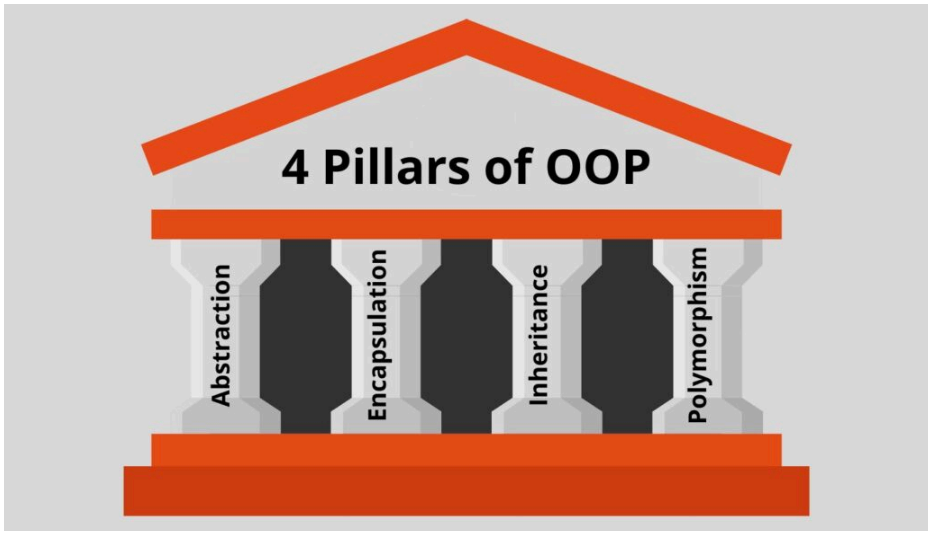
Agenda

OOPS: Introduction to inheritance and polymorphism, Abstract Class, Introduction to Abstraction and Encapsulation

OOPs

Object Oriented Programming is a concept that consists of Object, Class, Methods, and the 4 pillars of OOP.

Pillars of OOPs



Let's discuss this theoretically first and then we will implement it through programs.

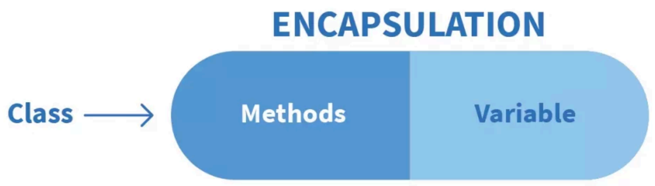
Abstraction:

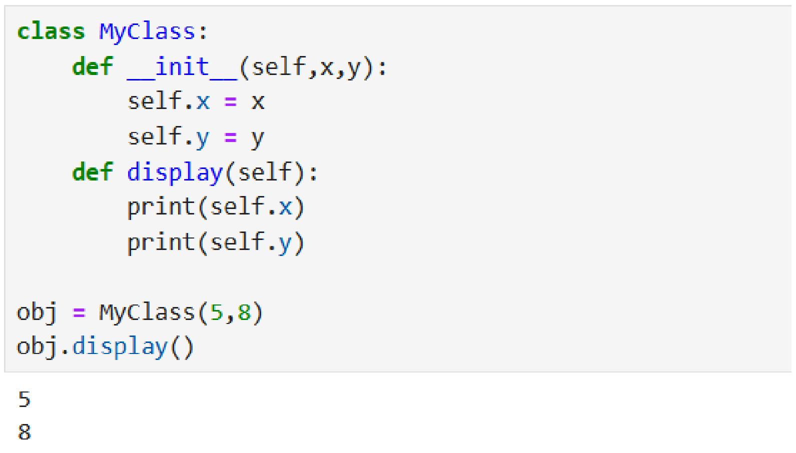
- It is a property by which we hide unnecessary details from the user.
- For example, if a user wants to order food, he/she should only have to give order details and get the order acknowledgment, they do not need to know about how the order is processed internally or what mechanisms we are using.

- Abstraction does the work of simplifying the interface and the user only has to think about the input and output and not about the process.
- We achieve abstraction through Abstract classes.

Encapsulation:

- Encapsulation is the process of binding the data (variable) and the method into a class so that the data is not available externally or to unauthorized personnel.
- This is done by restricting access using access modifiers.



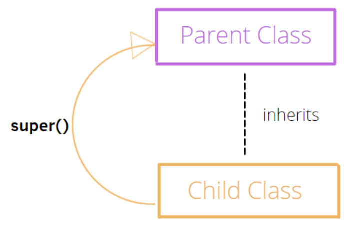


```
class MyClass:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def display(self):
        print(self.x)
        print(self.y)

obj = MyClass(5,8)
obj.display()
```

```
5
8
```

Inheritance:

- Inheritance is the property by which a class is able to access the data and methods of another class.
- The class inheriting these fields is called the Subclass or Child class and the class whose fields are inherited is called the Base class or Parent class or Super class.
- The sub-class can have additional fields and methods than those of the superclass.
- The purpose of inheritance is to increase code reusability and make the code more readable.



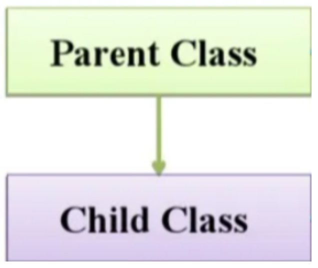
Polymorphism:

- Polymorphism is the ability to have different methods with the same name but perform different tasks.
- It can be achieved through method overriding or method overloading.
- Method overriding is a subclass having a different implementation of the method than the parent class.
- Method overloading is a function having different parameter types or number of parameters.

Let's Start with Inheritance first.

Inheritance

- The mechanism of deriving a new class from an old one (existing class) such that the new class inherits all the members (variables and methods) of the old class is called inheritance or derivation.
- **Parent Class** is the class being inherited from. This is also called **base class**.
- **Child Class** is the class that inherits from another class. This is also called derived class.



- All classes in python are built from a single super class called **‘object’** so whenever we create a class in python, object will become super class for them internally.

  class Mobile(object): is same as class Mobile:

- The main advantage of inheritance is code reusability.
- Python also allows the classes to inherit commonly used attributes and methods from other classes through inheritance.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Syntax

```
# define a ParentClass
class ParentClass:
    # attributes and method definition

# inheritance
class ChildClass(ParentClass):
    # attributes and method of ParentClass
    # attributes and method of ChildClass
```

Example:

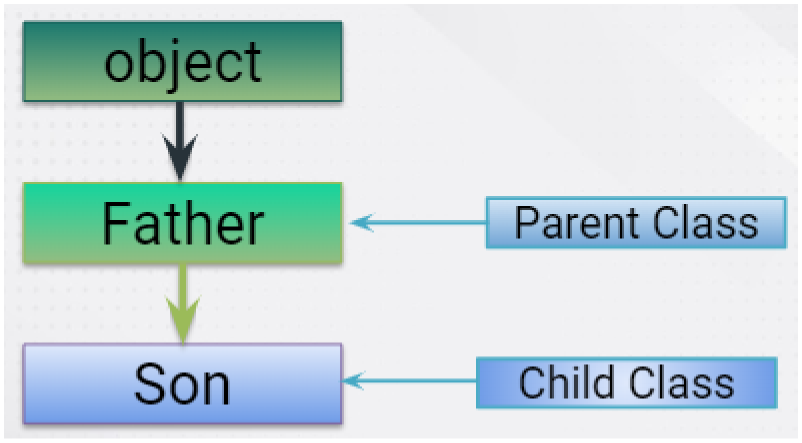
```python
class Animal:

    # attribute and method of the parent class
    name = ""

    def eat(self):
        print("I can eat")

# inherit from Animal
class Dog(Animal):

    # new method in subclass
    def display(self):
        # access name attribute of superclass using self
        print("My name is ", self.name)

# create an object of the subclass
labrador = Dog()

# access superclass attribute and method
labrador.name = "Rohu"
labrador.eat()

# call subclass method
labrador.display()
```

```
I can eat
My name is  Rohu
```

Types of Inheritance

- Single Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Multiple Inheritance

Single Inheritance



- Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.
- If a class is derived from one base class (Parent Class), it is called Single Inheritance.

Syntax:

```
class Father:
         members of class Father

class Son (Father):
        members of class Son
```

- We can access Parent Class Variables and Methods using Child Class Object.
- We can also access Parent Class Variables and Methods using Parent Class Object.
- We can not access Child Class Variables and Methods using Parent Class Object.

```
class Father:
    money = 1000
    def show(self):
        print("Parent class instance method")
    @classmethod
    def moneyshow(cls):
        print("Parent Class Class Method: Total money = ", cls.money)
    @staticmethod
    def stat_method():
        a = 5
        print("Parent Class Static Method: the value of a is ",a)

class Son(Father):
    def son_display(self):
        print("Child class instance method")

s = Son()
s.show()
s.moneyshow()
s.stat_method()
s.son_display()
```

```
Parent class instance method
Parent Class Class Method: Total money =  1000
Parent Class Static Method: the value of a is  5
Child class instance method
```

Constructor in Inheritance

- By default, The constructor in the parent class is available to the child class.

```
class Person:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname
        print("First name is {} and last name is {}".format(self.fname, self.lname))

class Student(Person):
    pass

s  = Student("Sachin", "Kumar")
```

```
First name is Sachin and last name is Kumar
```

Constructor Overriding

- If we write a constructor in both classes, parent class and child class then the parent class constructor is not available to the child class.
- In this case only child class constructor is accessible which means child class constructor is replacing parent class constructor.
- Constructor overriding is used when programmers want to modify the existing behavior of a constructor.

Example:

```python
class Person:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname
        print("First name is {} and last name is {}".format(self.fname, self.lname))
class Student(Person):
    def __init__(self, fname, lname):
        print("Inside Student class init")

s  = Student("Mohit", "Kumar")
```

```
Inside Student class init
```

One more Example of constructor overriding:

```python
class Father:
    def __init__(self):
        self.money = 2000
        print("Father Class Constructor")
class Son(Father):
    def __init__(self):
        self.money = 5000
        print("Son Class Constructor")
    def display(self):
        print(self.money)
s = Son()
s.display()
```

```
Son Class Constructor
5000
```

Constructor with super( ) Method

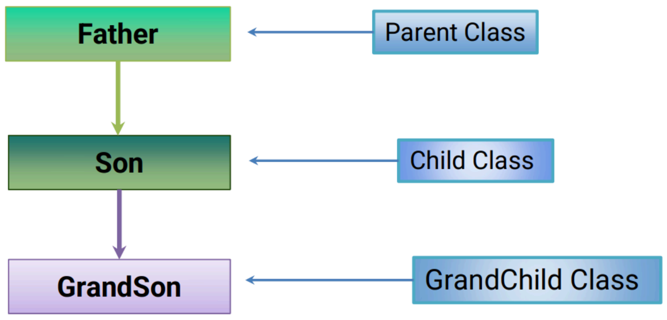
- If we write a constructor in both classes, parent class and child class then the parent class constructor is not available to the child class.
- In this case only child class constructor is accessible which means child class constructor is replacing parent class constructor.
- **super ( )** method is used to call parent class constructors or methods from the child class.

```python
class Person:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname
        print("First name is {} and last name is {}".format(self.fname, self.lname))

class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
        print("Inside Student class init")

s  = Student("Mohit", "Kumar")
```

```
First name is Mohit and last name is Kumar
Inside Student class init
```

Multi Level Inheritance

- In multi-level inheritance, the class inherits the feature of another derived class (Child Class).



Syntax:

```
class ParentClassName:
        members of Parent Class

class ChildClassName(ParentClassName):
        members of Child Class

class GrandChildClassName(ChildClassName):
        members of Grand Child Class
```

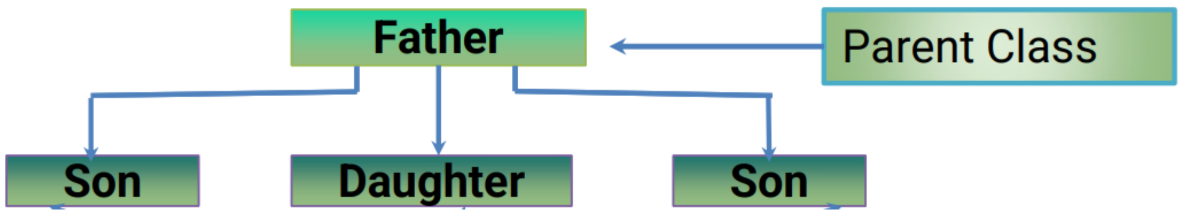
Example:

```
class Father:
    def __init__(self):
        print("Father class constructor")
    def show_father(self):
        print("Father class method")

class Son(Father):
    def __init__(self):
        print("Son class constructor")
    def show_son(self):
        print("Son class method")

class GrandSon(Son):
    def __init__(self):
        print("Grandson class constructor")
    def show_grand_son(self):
        print("Grandson class method")

g = GrandSon()
g.show_grand_son()
g.show_son()
g.show_father()
```

```
Grandson class constructor
Grandson class method
Son class method
Father class method
```

Hierarchical inheritance

- When more than one derived class is created from a single base class, this type of inheritance is called hierarchical inheritance.



Syntax:

```
class ParentClassName():
        members of Parent Class

class ChildClassName1(ParentClassName):
        members of Child Class 2

class ChildClassName2(ParentClassName):
        members of Child Class 2
```

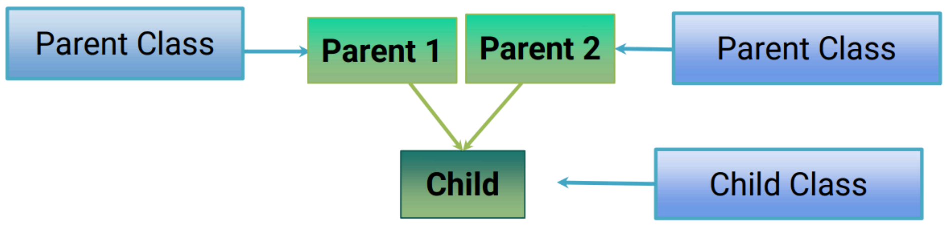
Example:

```python
class Father:
    def __init__(self):
        print("Father class constructor")
    def show_father(self):
        print("Father class method")

class Son(Father):
    def __init__(self):
        print("Son class constructor")
    def show_son(self):
        print("Son class method")

class Daughter(Father):
    def __init__(self):
        print("Daughter class constructor")
    def show_daughter(self):
        print("Daughter class method")

s = Son()
s.show_son()
s.show_father()

d = Daughter()
d.show_daughter()
d.show_father()
```

```
Son class constructor
Son class method
Father class method
Daughter class constructor
Daughter class method
Father class method
```

Multiple Inheritance

- If a class is derived from more than one parent class, then it is called multiple inheritance.



Syntax:

```
class ParentClassName1():
          members of Parent Class

class ParentClassName2():
          members of Parent Class

classChildClassName(ParentClassName1, ParentClassName2):
          members of Child Class
```

Example:

```
class Father:
    def __init__(self):
        print("Father class constructor")
    def show_father(self):
        print("Father class method")

class Mother():
    def __init__(self):
        print("Mother class constructor")
    def show_mother(self):
        print("Mother class method")

class Son(Father, Mother):
    def __init__(self):
        print("Son class constructor")
    def show_son(self):
        print("Son class method")

s = Son()
s.show_father()
s.show_mother()
s.show_son()
```

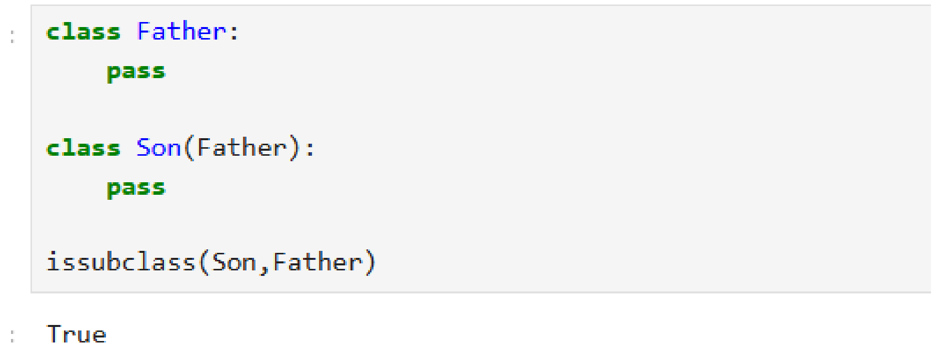
```
Son class constructor
Father class method
Mother class method
Son class method
```

issubclass() Function

- In Python issubclass() is a built-in function used to check if a class is a subclass of another class or not.
- This function returns True if the given class is the subclass of the given class, else it returns False.

Syntax:

issubclass(class1,class2)



```
class Father:
    pass

class Son(Father):
    pass

issubclass(Son,Father)
```

```
True
```

Encapsulation

- Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.
- In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.
- In Python, we denote private attributes using double underscore as the prefix.

```
class Computer:
    def __init__(self):
        self.__maxprice = 900
    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))
    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()
c.setMaxPrice(1000)
c.sell()
```

```
Selling Price: 900
Selling Price: 1000
```

- In the above program, we defined a Computer class.
- We used __init__() method to store the maximum selling price of Computer.
- If we tried to modify the price through __maxprice attribute, we can't change it because Python treats the __maxprice as private attributes.
- As shown, to change the value, we have to use a setter function i.e setMaxPrice() which takes price as a parameter.

Access Modifiers in Python

- Access modifiers are used to restrict access to the variables and methods of the class.
- Python uses '_' symbol to determine the access control for a specific data member or a member function of a class.
- A Class in Python has three types of access modifiers:
  - Public
  - Protected
  - Private

Public Access Modifiers

- The members of a class that are declared public are easily accessible from any part of the program.

- All data members and member functions of a class are public by default.

Protected Access Modifiers

- The members of a class that are declared protected are only accessible to a class derived from it.
- Data members of a class are declared protected by adding a **single underscore** ‘_’ symbol before the data member of that class.

Example:

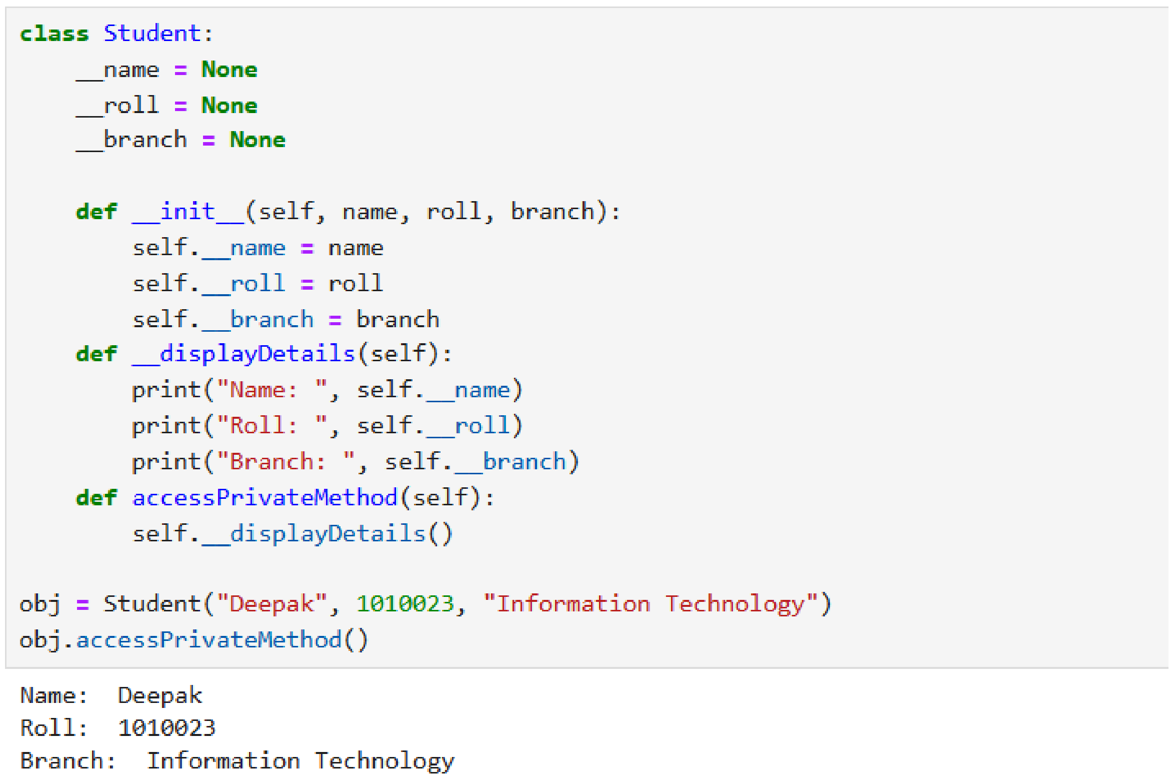
```python
class Student:
    _name = None
    _roll = None
    _branch = None

    def __init__(self, name, roll, branch):
        self._name = name
        self._roll = roll
        self._branch = branch

    def _displayInfo(self):
        print("Roll: ", self._roll)
        print("Branch: ", self._branch)

class Learnowx(Student):
    def __init__(self, name, roll, branch):
        Student.__init__(self, name, roll, branch)

    def displayDetails(self):
        print("Name: ", self._name)
        self._displayInfo()

obj = Learnowx("Aditya", 101203, "Data Science")
obj.displayDetails()
```

```
Name:  Aditya
Roll:  101203
Branch:  Data Science
```

Private Access Modifiers

- The members of a class that are declared private are accessible within the class only, private access modifier is the most secure access modifier.
- Data members of a class are declared private by adding a **double underscore** ‘__’ symbol before the data member of that class.

Example



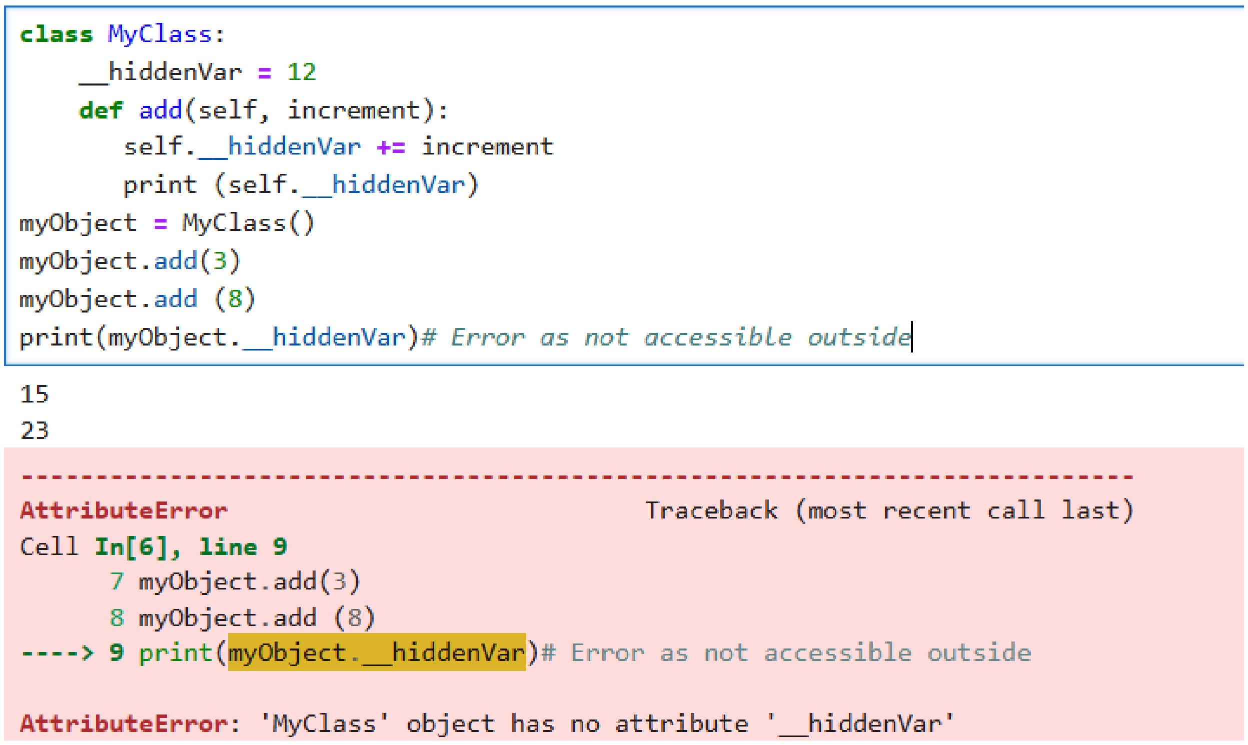
```python
class Student:
    __name = None
    __roll = None
    __branch = None

    def __init__(self, name, roll, branch):
        self.__name = name
        self.__roll = roll
        self.__branch = branch
    def __displayDetails(self):
        print("Name: ", self.__name)
        print("Roll: ", self.__roll)
        print("Branch: ", self.__branch)
    def accessPrivateMethod(self):
        self.__displayDetails()

obj = Student("Deepak", 1010023, "Information Technology")
obj.accessPrivateMethod()
```

```
Name:   Deepak
Roll:   1010023
Branch:   Information Technology
```

Data Hiding in Python

- Data hiding is generally used to hide the data information from the user.
- It includes internal object details such as data members, internal working.
- It maintained the data integrity and restricted access to the class member.
- The main working of data hiding is that it combines the data and functions into a single unit to conceal data within a class.
- We cannot directly access the data from outside the class.

Example:



```
class MyClass:
    __hiddenVar = 12
    def add(self, increment):
       self.__hiddenVar += increment
       print (self.__hiddenVar)
myObject = MyClass()
myObject.add(3)
myObject.add (8)
print(myObject.__hiddenVar)# Error as not accessible outside
```

```
15
23
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[6], line 9
      7 myObject.add(3)
      8 myObject.add (8)
----> 9 print(myObject.__hiddenVar)# Error as not accessible outside

AttributeError: 'MyClass' object has no attribute '__hiddenVar'
```

Method Resolution Order (MRO)

- In the multiple inheritance scenario members of class are searched first in the current class. If not found, the search continues into parent classes in depth-first, left to right manner without searching the same class twice.
- Search for the child class before going to its parent class.
- When a class is inherited from several classes, it searches in the order from left to right in the parent classes.
- It will not visit any class more than once which means a class in the inheritance hierarchy is traversed only once exactly.

Example:

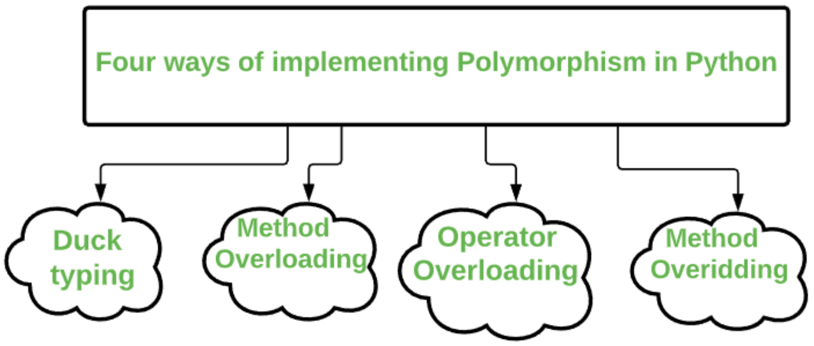
```python
class A:
    def method(self):
        print("Method in A")

class B(A):
    def method(self):
        print("Method in B")

class C(A):
    def method(self):
        print("Method in C")

class D(B, C):
    pass

# Example usage
d = D()
d.method()

# Print MRO
print(D.mro())
```

```
Method in B
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

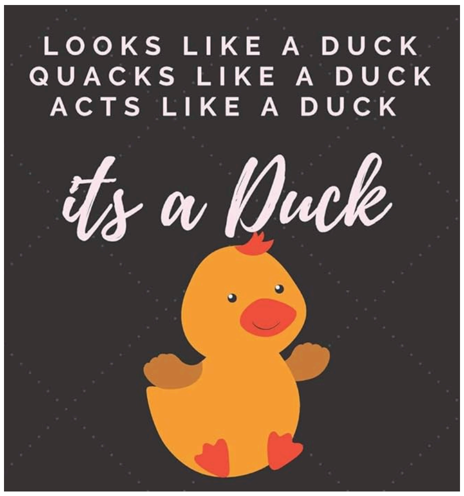
This means first the method will be looked into D → B → C → A → Object class.

Polymorphism

- Polymorphism is a word that came from two Greek words, poly means many and morphos means forms.
- If a variable, object or method performs different behavior according to the situation, it is called polymorphism.
    - Duck Typing
    - Method Overloading
    - Operator Overloading
    - Method Overriding



DuckTyping:



It's a very simple and obvious concept that if the same method or variable is in two different classes, the object will use a method or variable of its own class.

Example:

```python
class Duck:
    def quack(self):
        print("Quack!")

class Person:
    def quack(self):
        print("I'm quacking like a duck!")

# Both Duck and Person can be passed to make_it_quack
duck = Duck()
person = Person()

duck.quack()    # Output: Quack!
person.quack()  # Output: I'm quacking like a duck!
```

```
Quack!
I'm quacking like a duck!
```

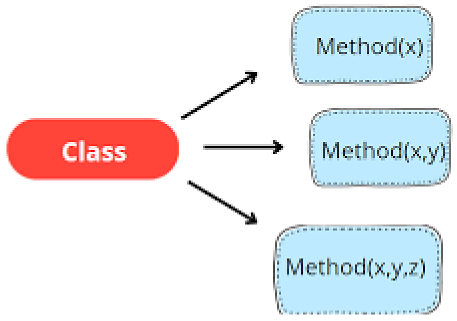
Example:

```python
class Bird:
    def fly(self):
        print("fly with wings")

class Airplane:
    def fly(self):
        print("fly with fuel")

class Fish:
    def swim(self):
        print("fish swim in sea")

# Attributes having same name are
# considered as duck typing
for obj in Bird(), Airplane(), Fish():
    obj.fly()
```

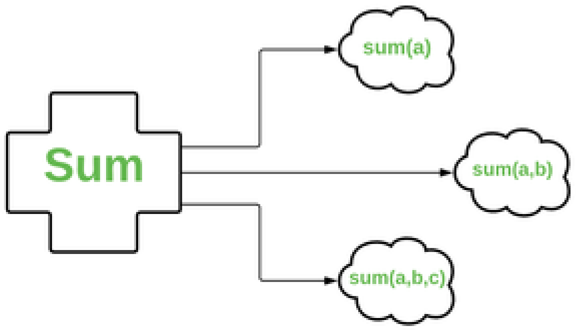
```
fly with wings
fly with fuel
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[14], line 16
     13 # Attributes having same name are
     14 # considered as duck typing
     15 for obj in Bird(), Airplane(), Fish():
---> 16     obj.fly()

AttributeError: 'Fish' object has no attribute 'fly'
```

Method Overloading



- Method Overloading is the class having methods that are the same name with different arguments.
- In a class, a single method can be used with different parameters.
- Like other languages (C+=, JAVA), python does not support method overloading by default. But there are different ways to achieve method overloading in Python. Like through default argument, variable length argument.



Example:

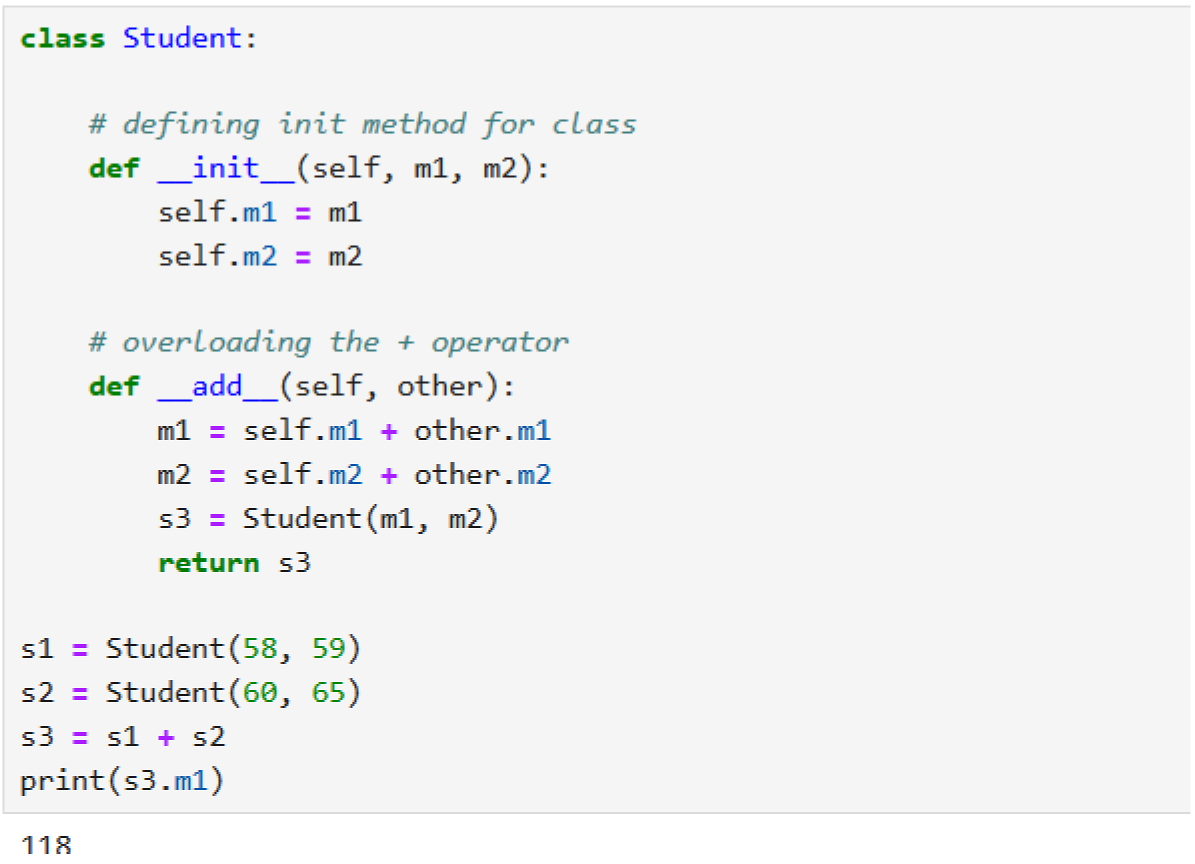
```python
class MyClass:
    def sum(self, a = None, b = None, c = None):
        s = 0
        if a != None and b != None and c != None:
            s = a + b + c
        elif a != None and b != None:
            s = a + b
        else:
            s = a
        return s

s = MyClass()

# sum of 1 integer
print(s.sum(1))

# sum of 2 integers
print(s.sum(3, 5))

# sum of 3 integers
print(s.sum(1, 2, 3))
```

```
1
8
6
```

Operator Overloading

- The + operator is used for adding numbers and at the same time to concatenate strings.
- Operator overloading in Python is the ability of a single operator to perform more than one operation based on the class (type) of operands. So, basically defining methods for operators is known as operator overloading. For e.g: To use the + operator with custom objects you need to define a method called __add__.
- It is possible because the + operator is overloaded by both int class and str class. The operators are actually methods defined in respective classes.

Example:



```
class Student:

    # defining init method for class
    def __init__(self, m1, m2):
        self.m1 = m1
        self.m2 = m2

    # overloading the + operator
    def __add__(self, other):
        m1 = self.m1 + other.m1
        m2 = self.m2 + other.m2
        s3 = Student(m1, m2)
        return s3

s1 = Student(58, 59)
s2 = Student(60, 65)
s3 = s1 + s2
print(s3.m1)
```

```
118
```

Example: Perform vector operations like addition, subtract and dot product.

(Try Yourself before looking into solution)

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

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, other):
        # Dot product of two vectors
        return self.x * other.x + self.y * other.y

    def __str__(self):
        return f"({self.x}, {self.y})"

# Example usage
v1 = Vector(2, 3)
v2 = Vector(4, 5)

v3 = v1 + v2
v4 = v1 - v2
dot_product = v1 * v2

print("v1:", v1)                    # Output: v1: (2, 3)
print("v2:", v2)                    # Output: v2: (4, 5)
print("v1 + v2:", v3)               # Output: v1 + v2: (6, 8)
print("v1 - v2:", v4)               # Output: v1 - v2: (-2, -2)
print("v1 * v2 (dot product):", dot_product)  # Output: v1 * v2 (dot product): 23
```

```
v1: (2, 3)
v2: (4, 5)
v1 + v2: (6, 8)
v1 - v2: (-2, -2)
v1 * v2 (dot product): 23
```

Method overriding

- Applicable with Inheritance only.

- Method overriding in Python occurs when a child class provides an implementation of a method that is already defined in its parent class. This allows a child to modify or extend the behavior of that method.

Example:

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

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

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

# Example usage
animal = Animal()
dog = Dog()
cat = Cat()

animal.speak()  # Output: The animal makes a sound
dog.speak()     # Output: The dog barks
cat.speak()     # Output: The cat meows
```

```
The animal makes a sound
The dog barks
The cat meows
```

Example: Create a base class BankAccount and two derived classes SavingsAccount and CurrentAccount. The derived classes will override a method calculate_interest to provide specific implementations for calculating interest for savings and current accounts.

```python
class BankAccount:
    def __init__(self, account_number, balance):
        self.account_number = account_number
        self.balance = balance

    def calculate_interest(self):
        # Base method for calculating interest, might be overridden in subclasses
        return self.balance * 0.01  # Default interest rate of 1%

    def display(self):
        print(f"Account Number: {self.account_number}, Balance: {self.balance}")

class SavingsAccount(BankAccount):
    def calculate_interest(self):
        # Specific interest calculation for SavingsAccount
        return self.balance * 0.04  # Savings accounts get 4% interest

class CurrentAccount(BankAccount):
    def calculate_interest(self):
        # Specific interest calculation for CurrentAccount
        return self.balance * 0.02  # current accounts get 2% interest

# Example usage
savings = SavingsAccount("S123", 1000)
Current = CurrentAccount("C456", 2000)

savings.display()  # Output: Account Number: S123, Balance: 1000
Current.display() # Output: Account Number: C456, Balance: 2000

print(f"Savings Account Interest: {savings.calculate_interest()}")  # Output: 40.0
print(f"Current Account Interest: {Current.calculate_interest()}")  # Output: 40.0
```

```
Account Number: S123, Balance: 1000
Account Number: C456, Balance: 2000
Savings Account Interest: 40.0
Current Account Interest: 40.0
```

Method Overloading vs Overriding

Method Overloading	Method Overriding

Methods or functions must have the same name and different parameters.	Methods or functions must have the same name and same parameters.
It is an example of compile time polymorphism.	It is an example of run time polymorphism.
In the method overloading, inheritance may or may not be required.	Whereas in method overriding, inheritance always required.
It is performed between methods within the class.	It is done between parent class and child class methods.
It is used in order to add more to the behavior of methods.	Whereas it is used in order to change the behaviour of exist methods.
There is no need of more than one class.	There is need of at least of two classes.

Abstract Class

- A class is called an **Abstract class** if it contains one or more abstract methods.
- An **abstract method** is a method that is declared, but contains no implementation.
- Abstract classes may not be instantiated.
- Its abstract methods must be implemented by its subclasses.
- In Python, Abstract classes are derived **from ABC** class which belongs to **abc** module.
- **@abstractmethod** decorator is used to declare abstract method.

Syntax:

```python
from abc import ABC, abstractmethod

class MyAbstractClass(ABC):
    @abstractmethod
    def my_abstract_method(self):
        pass
```

- Abstract Class must have **abstract methods** but they can also have some **concrete methods** (Concrete methods are the methods in Abstract class that are implemented in abstract class itself).

```python
from abc import ABC, abstractmethod
class Father(ABC):
    @abstractmethod
    def disp(self):
        pass
    def show(self):  #Concrete method
        print("Concrete Method")
```

Example

```python
from abc import ABC, abstractmethod
class Computer(ABC):
    @abstractmethod
    def process(self):
        pass
    def message(self):
        print("Computer is an electronic device")
class Laptop(Computer):
    def process(self):
        print("Executing Laptop Process")
class Desktop(Computer):
    def process(self):
        print("Executing Desktop Process")

com1 = Laptop()
com2 = Desktop()
com1.process()
com2.message()
com2.process()
```

```
Executing Laptop Process
Computer is an electronic device
Executing Desktop Process
```

Example:

```python
from abc import ABC, abstractmethod

# Abstract Class
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

    # Concrete method
    def description(self):
        return "This is a shape."

# Concrete Class: Circle
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

    def perimeter(self):
        return 2 * 3.14 * self.radius
```

```python
# Concrete Class: 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)

# Example usage
circle = Circle(5)
rectangle = Rectangle(4, 6)

print(f"Circle area: {circle.area()}")
print(f"Circle perimeter: {circle.perimeter()}")
print(f"Circle description: {circle.description()}")

print(f"Rectangle area: {rectangle.area()}")
print(f"Rectangle perimeter: {rectangle.perimeter()}")
print(f"Rectangle description: {rectangle.description()}")
```

```
Circle area: 78.5
Circle perimeter: 31.400000000000002
Circle description: This is a shape.
Rectangle area: 24
Rectangle perimeter: 20
Rectangle description: This is a shape.
```