OOP(Polymorphism and Encapsulation)

Polymorphism:

The word polymorphism means having many forms. In programming, polymorphism means the same function name (but different signatures) being used for different types.

Example of inbuilt polymorphic functions:

```
# len() being used for a string
print(len("Python"))

# len() being used for a list
print(len([10, 20, 30]))
```

Output:

```
6
3
```

Example of user defined polymorphic functions:

```
def add(x, y, z = 0):
    return x + y+z

# Driver code
print(add(2, 3))
print(add(2, 3, 4))
```

Output:

```
5
9
```

Polymorphism with class methods:

The below code shows how Python can use two different class types, in the same way. We create a for loop that iterates through a tuple of objects. Then call the methods without being concerned about which class type each object is. We assume that these methods actually exist in each class.

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of
India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")

    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()
    country.type()
```

Output:

```
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.
```

Polymorphism with Inheritance:

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**.

```
class Bird:
  def intro(self):
    print("There are many types of birds.")

  def flight(self):
    print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
  def flight(self):
    print("Sparrows can fly.")

class ostrich(Bird):
  def flight(self):
    print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

Output:

```
There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.
```

Polymorphism with a Function and objects:

It is also possible to create a function that can take any object, allowing for polymorphism. In this example, let's create a function called "func()" which will take an object which we will name "obj". Though we are using the name 'obj', any instantiated object will be able to be called into this function. Next, let's give the function something to do that uses the 'obj' object we passed to it. In this case, let's call the three methods, viz., capital(), language() and type(), each of which is defined in the two classes 'India' and 'USA'. Next, let's create instantiations of both the 'India' and 'USA' classes if we don't have them already. With those, we can call their action using the same func() function:

```
def func(obj):
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

Code: Implementing Polymorphism with a Function

```
class India():
    def capital(self):
        print("New Delhi is the capital of India.")

    def language(self):
        print("Hindi is the most widely spoken language of
India.")

    def type(self):
        print("India is a developing country.")

class USA():
    def capital(self):
        print("Washington, D.C. is the capital of USA.")
    def language(self):
        print("English is the primary language of USA.")

    def type(self):
        print("USA is a developed country.")

def func(obj):
    obj.capital()
    obj.language()
    obj.type()

obj_ind = India()
obj_usa = USA()

func(obj_ind)
func(obj_usa)
```

Output:
New Delhi is the capital of India.
Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.

Encapsulation:

- It describes the idea of wrapping data and the methods that work on data within one unit.
- This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.
- To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as **private variable**.
- A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

## Protected members

- Protected members (in C++ and JAVA) are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses.
- To accomplish this in Python, just follow **the convention** by prefixing the name of the member by a **single underscore "_"**.

  Note: The __init__ method is a constructor and runs as soon as an object of a class is instantiated.

```
# Creating a base class
class Base:
    def __init__(self):

        # Protected member
        self._a = 2

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling protected member of base class: ")
        print(self._a)

obj1 = Derived()

obj2 = Base()

# Calling protected member
# Outside class will  result in
# AttributeError
print(obj2.a)
```

Output:

Calling protected member of base class:

```
2

Traceback (most recent call last):

  File "C:/Users/RITI/AppData/Local/Programs/Python/Python39/reve.py", line 25, in <module>

    print(obj2.a)

AttributeError: 'Base' object has no attribute 'a'
```

Private members

Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class. In Python, there is no existence of **Private** instance variables that cannot be accessed except inside a class. However, to define a private member prefix the member name with double underscore "__".

Note: Python's private and protected member can be accessed outside the class through

```
# Python program to

# demonstrate private members

# Creating a Base class

class Base:

    def __init__(self):

        self.a = "BennettUniversity"

        self.__c = "BennettUniversity"


# Creating a derived class

class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__c)
# Driver code
obj1 = Base()
print(obj1.a)
```

Output:

BennettUniversity

Q1. Predict the output

```
class Demo:
    def __init__(self):
        self.a = 1
        self.__b = 1

    def display(self):
        return self.__b
obj = Demo()
print(obj.a)

Sol.
    1
```

Q2. Predict the output

```
class Demo:
    def __init__(self):
        self.a = 1
        self.__b = 1

    def display(self):
        return self.__b
obj = Demo()
print(obj.__b)

Sol.
AttributeError: 'Demo' object has no attribute '__b'
```

Explanation: Variables beginning with two underscores are said to be private members of the class and they can’t be accessed directly.

Q3. Predict the output

```
class Demo:
     def __init__(self):
         self.a = 1
         self.__b = 1
     def get(self):
         return self.__b

obj = Demo()
print(obj.get())

Sol.
     1
```

Q4. Predict the output

```
class Demo:
     def __init__(self):
         self.a = 1
         self.__b = 1
     def get(self):
         return self.__b
obj = Demo()
obj.a=45
print(obj.a)

Sol.
    45
```

Q5. Predict the output

```
class student:
    def __init__(self):
        self.marks = 97
        self.__cgpa = 8.7
    def display(self):
        print(self.marks)
obj=student()
print(obj._student__cgpa)

Sol.
    8.7
```

Q6. Predict the output

```
class objects:
    def __init__(self):
        self.colour = None
        self._shape = "Circle"

    def display(self, s):
        self._shape = s
obj=objects()
print(obj._objects_shape)
```

```
Sol.
AttributeError: 'objects' object has no attribute '_objects_shape'
```

Explanation: Protected members begin with one underscore and they can only be accessed within a class or by subclasses.

Q7. Predict the output

```
class A:
    def __str__(self):
        return '1'
class B(A):
    def __init__(self):
        super().__init__()
class C(B):
    def __init__(self):
        super().__init__()
def main():
    obj1 = B()
    obj2 = A()
    obj3 = C()
    print(obj1, obj2,obj3)
main()
```

```
Sol.
    1 1 1
```

Q8. Predict the output

```
class A:
    def __repr__(self):
        return "1"
class B(A):
    def __repr__(self):
        return "2"
class C(B):
    def __repr__(self):
        return "3"
obj1 = A()
obj2 = B()
obj3 = C()
print(obj1, obj2, obj3)

Sol.
    1 2 3
```

Q9. Predict the output

```
class Demo:
    def check(self):
        return " Demo's check "
    def display(self):
        print(self.check())
class Demo_Derived(Demo):
    def check(self):
        return " Derived's check "
Demo().display()
Demo_Derived().display()

Sol.
    Demo's check
    Derived's check
```

Q10. Predict the output

```
class A:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __str__(self):
        return 1
    def __eq__(self, other):
        return self.x * self.y == other.x * other.y
obj1 = A(5, 2)
obj2 = A(2, 5)
print(obj1 == obj2)
```

Sol.

```
    True
```

Q11. Predict the output

```
class A:
    def one(self):
        return self.two()
    def two(self):
        return 'A'
class B(A):
    def two(self):
        return 'B'
obj2=B()
print(obj2.two())
```

```
Sol.
    B
```

Q12. Predict the output

```
class Base:
    def __init__(self):
        self.x = 1

    def convert(self):
        self.x = 10

class Derived(Base):
    def convert(self):
        self.x = self.x  + 1
        return self.x

def main():
    obj = Derived()
    print(obj.convert())

main()

Sol.
   2
```

Q13. Predict the output

```
class A:
    def __init__(self):
        self.multiply(10)
        print(self.i)

    def multiply(self, i):
        self.i = 4 * i;

class B(A):
    def __init__(self):
        super().__init__()

    def multiply(self, i):
        self.i = 2 * i;
```

```
Sol.
   20
```

Q14. Predict the output

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

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

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

```
Sol.
    Vegetable
    Red
    Fruit
    Red
```

Q15. Predict the output

```
class India():
     def capital(self):
       print("New Delhi")

     def language(self):
       print("Hindi and English")

class USA():
     def capital(self):
        print("Washington, D.C.")

      def language(self):
        print("English")


obj_ind = India()
obj_usa = USA()
for country in (obj_ind, obj_usa):
    country.capital()
    country.language()

Sol.

New Delhi
Hindi and English
Washington, D.C.
English
```

Q16. Predict the output

```
class BaseClass:
    def __init__(self):
        self._x = 10

class DerivedClass(BaseClass):
    def __init__(self):
        BaseClass.__init__(self)
        print(self._x)

obj1 = DerivedClass()
obj2 = BaseClass()
print(obj2.x)

Sol.
   10
   AttributeError: 'BaseClass' object has no attribute 'x'
```