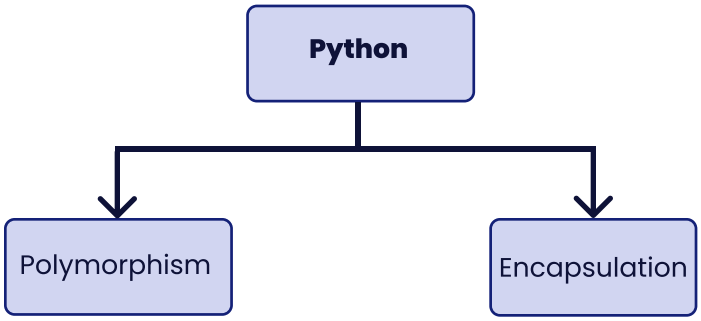
PW SKILLS

Lesson Plan

Polymorphism and Encapsulation

Topic to covered:

- Polymorphism in Python
- Encapsulation in Python



Polymorphism in Python: In OOP, polymorphism refers to an object's capacity to assume several forms. Simply said, polymorphism enables us to carry out a single activity in a variety of ways

From the Greek words poly (many) and morphism (forms), we get polymorphism. Polymorphism is the capacity to assume several shapes.

Ex:

```
[2] print(len('iNeuron'))
    print(len(['iNeuron','PwSkills']))

    7
    2
```

Inheritance is the primary application of polymorphism. The traits and methods of a parent class are passed down to a child class through inheritance. A subclass, child class, or derived class is a new class that is created from an existing class, which is referred to as a base class or parent class.

Method overriding polymorphism enables us to define child class methods with the same names as parent class methods. The act of overriding an inherited method in a child class is referred to as method overriding.

In python, polymorphism is achieved through method overloading and method overriding.

1. Method overloading : Method overloading is the practice of invoking the same method more than once with different parameters. Method overloading is not supported by Python. Even if you overload the method, Python only takes into account the most recent definition. If you overload a method in Python, a TypeError will be raised.

```
[6] def mul(x,y):
      z = x*y
      print(z)
    def mul(p,q,r):
      s = p*q*r
      print("Output: ",s)
    # mul(5,6)  error
    mul(5,2,3)

    Output:  30
```

Ex:

2. Method overriding : In Python, method overriding is the process of providing a different implementation for a method that is already defined in the superclass within a subclass. It enables the subclass to define its own version of a method with the same name and parameters as the method in the superclass. When a method is overridden, the subclass implements the method in its own way, which overrides the behaviour defined in the superclass. The subclass can then alter or expand the functionality of the inherited method.

The name, parameters, and return type of the overridden method in the subclass must match those of the method in the superclass. Method overriding occurs only when the subclass has a method with the same name and signature as the superclass method.

When a subclass object is used to call the overridden method during runtime, the subclass implementation is invoked rather than the superclass implementation. A crucial element of polymorphism is the dynamic dispatch of methods based on the actual object type.

```
class Animal:
    def sound(self):
        print("Animal sound")

class Cat(Animal):
    def sound(self):
        print("Cat meows")

# Creating objects
animal = Animal()
cat = Cat()

# Calling the overridden method
animal.sound()
cat.sound()
```

```
Animal sound
Cat meows
```

Encapsulation in Python : Encapsulation is a Python technique for combining data and functions into a single object. A class, for instance, contains all the data (methods and variables). Encapsulation refers to the broad concealment of an object's internal representation from areas outside of its specification.

Assume, for instance, that you combine methods that provide read or write access with an attribute that is hidden from view on the exterior of an object. Then, you may limit who has access to the object's internal state and hide particular pieces of information. Without giving the program complete access to all of a class's variables, encapsulation provides a mechanism for us to obtain the necessary variable. This method is used to shield an object's data from other objects.

Access Modifier in python : A class's data members and methods can be made private or protected in order to achieve encapsulation. Direct access modifiers like public, private, and protected don't exist in Python, though. Single and double underscores can be used to accomplish this.

Modifiers for access control restrict use of a class's variables and methods. Private, public, and protected are the three different access modifier types that Python offers.

Public Member : from outside of class, anywhere accessible.

Private Member : Within the class, accessible.

Protected Member : Within the class and its subclasses, accessible.

Public Member : Both inside and outside of a class, public data members are accessible. By default, the class's member variables are all public.

```
[1] class Student:
      def __init__(self,name,degree):
        #Public Data Member
        self.name = name
        self.degree = degree

      def show(self) :
        # Accessing Public Data Memeber
        print('Name: ',self.name,'Degree: ',self.degree)

    stud = Student('Rohit','B.tech')
    stud.show()
```

```
Name:  Rohit Degree:  B.tech
```

Private Member : By designating class variables as private, we may protect them. Add two underscores as a prefix to the beginning of a variable name to define it as a private variable.
Private members can only be accessed within the class; they are not directly available from class objects.
We can access private members from outside of a class using the following two approaches.

- Create a public method to access private members of a class.
- By using name mangling

```
[4] class Student:
      def __init__(self,name,degree):
        #Public  Member
        self.name = name
        # Private Member
        self.__degree = degree

    stud = Student('Rohit','B.tech')
    print('Degree: ',stud._Student__degree)
```

```
Degree:  B.tech
```

Protected Member : Within the class and to its sub-classes, protected members can be accessed. Add a single underscore (_) before the member name to define it as a protected member.
When you implement inheritance and wish to restrict access to data members to just child classes, you use protected data members.

```
[5] class College:
      def __init__(self):
        # Protected member
        self._college_name='PwSkills'

    class Student(College):
      def __init__(self,name):
        self.name=name
        College.__init__(self)

      def show(self):
        print('Student Name: ',self.name)
        print('He is studying at ',self._college_name)

    stud = Student('Rocky')
    stud.show()
```

```
Student Name:  Rocky
He is studying at  PwSkills
```

PW SKILLS

THANK YOU !