Data Abstraction

The process by which data and functions are defined in such a way that only essential details can be seen and unnecessary implementations are hidden is called Data Abstraction.

Data Abstraction means to hide internal functionalities of an application (codes) and to show only essential information (class attributes).

- The users only interact with the basic implementation of the function, but inner working is hidden.
- User is familiar with that **"what function does"** but they don't know **"how it does."**

By terms abstracting, we mean something to provide a name to things so that by seeing the name, the responsible programmer may understand the idea of what the whole program is doing behind the scene.

Abstract Class

- In Python, abstraction can be achieved by using abstract classes.
- A class that consists of one or more abstract method is called the abstract class.
- **Abstract classes can have any number of abstract methods coexisting with any number of other methods. i.e. Abstract class can have both a normal method and an abstract method**
- Abstract methods do not contain their implementation, they get their definition in the Child class.
- **An abstract class cannot be instantiated, (we cannot create objects for the abstract class).**
- **An object of the derived class is used to access the methods of the base class.**

Creating Abstract Class

Python provides the **abc (abstract base class)** module to use the abstraction in the Python program.

From this module Abstract Base Class (ABC) is imported to implement abstraction.

To import the ABC class from the **abc** module:

Syntax:

```
from abc import ABC

class ClassName(ABC):
```

Example -

```
# Python program demonstrate abstract base class work
from abc import ABC
class Car(ABC):
  # abstract method
  def mileage(self):
      pass

class Tesla(Car):
   def mileage(self):
      print("The mileage is 30kmph")

class Suzuki(Car):
   def mileage(self):
      print("The mileage is 25kmph ")

class Duster(Car):
    def mileage(self):
       print("The mileage is 24kmph ")

class Renault(Car):
   def mileage(self):
        print("The mileage is 27kmph ")

# Driver code
t= Tesla ()
t.mileage()

r = Renault()
r.mileage()

s = Suzuki()
s.mileage()

d = Duster()
d.mileage()
```

Output:

The mileage is 30kmph
The mileage is 27kmph
The mileage is 25kmph
The mileage is 24kmph

Explanation -

In the above code, we have imported the **abc module** to create the abstract base class. We created the Car class that inherited the ABC class and defined an abstract method named mileage(). We have then inherited the base class from the three different subclasses and implemented the abstract method differently. We created the objects to call the abstract method.

Example -

```
# Python program to define  abstract class
from abc import ABC

class Polygon(ABC):
  # abstract method
  def sides(self):
    pass

class Triangle(Polygon):
  def sides(self):
    print("Triangle has 3 sides")

class Pentagon(Polygon):
  def sides(self):
    print("Pentagon has 5 sides")

class Hexagon(Polygon):
  def sides(self):
    print("Hexagon has 6 sides")

class square(Polygon):
  def sides(self):
    print("I have 4 sides")

# Driver code
t = Triangle()
t.sides()

s = square()
s.sides()

p = Pentagon()
p.sides()

k = Hexagon()
K.sides()
```

Output:

```
Triangle has 3 sides
Square has 4 sides
Pentagon has 5 sides
Hexagon has 6 sides
```

Explanation -

In the above code, we have defined the abstract base class named Polygon and we also defined the abstract method. This base class is inherited by the various subclasses. We implemented the abstract method in each subclass.

We created the object of the subclasses and invoke the **sides()** method. The hidden implementations for the **sides()** method inside the each subclass comes into play. The abstract method **sides()** method, defined in the abstract class, is never invoked.

Example:

```
from abc import ABC

#abstract class
class Calculate(ABC):
  def Area(self):
    pass

class Square(Calculate):
  length = 5
  def Area(self):
    return self.length * self.length


class Circle(Calculate):
  radius =4
  def Area(self):
    return 3.14 * self.radius * self.radius


sq = Square() #object created for the class 'Square'

cir = Circle() #object created for the class 'Circle'

print("Area of a Square:", sq.Area()) #call to 'calculate_area' method defined inside the class 'Square'
print("Area of a circle:", cir.Area()) #call to 'calculate_area' method defined inside the class 'Circle'.
```

Output:

```
Area of a Square: 25
Area of a circle: 50.24
An abstract class can have both a normal method and an abstract method
An abstract class cannot be instantiated, (we cannot create objects for the abstract class).
```

Additional Points:

Why Do We Need Abstraction?

- Through the process of abstraction in Python, a programmer can hide all the irrelevant data of an application.
- Abstract class just serves as a template for other classes by defining a list of methods that the classes must implement.
- An abstract class can be useful when we are designing large functions.
- An abstract class is also helpful to provide the standard interface for different implementations of components.
- It is useful while working in large teams and code-bases so that all of the classes need not be remembered and also be provided as library by third parties.
- It reduces complexity and increase efficiency.

Note:

- ✓ We need to import the abc module, which provides the base for defining Abstract Base classes (ABC). The ABC works by decorating methods of the base class as abstract. It registers concrete classes as the implementation of the abstract base.
- ✓ We use the ***@abstractmethod*** decorator to define an abstract method or if we don't provide the definition to the method, it automatically becomes the abstract method.

Let's understand the following example.

Example:

```
from abc import ABC, abstractmethod

class Parent(ABC):
  #common function
  def common_fn(self):
    print('In the common method of Parent')

 @abstractmethod
  def abs_fn(self): #is supposed to have different implementation in child classes
    pass

class Child1(Parent):
  def abs_fn(self):
    print('In the abstract method of Child1')

class Child2(Parent):
  def abs_fn(self):
    print('In the abstract method of Child2')
```

Access Modifiers/ Access Specifiers

- **Access specifiers** or access modifiers in python programming are used to limit the access of class variables and class methods outside of class while implementing the concepts of inheritance.
- Access specifiers have an important role in securing data from unauthorized access and in preventing it from being exploited.
- Python uses '_' symbol to determine the access control for a specific data member or a member function of a class.

Types of Access Specifiers / Access Modifiers

1) Public access modifier
2) Private access modifier
3) Protected access modifier

1) Public Access Modifier

- All the variables and methods (member functions) in python are by default public.
- Any instance variable in a class followed by the 'self' keyword ie. self.var_name is public accessed.

Example:

```
class Student:
        def __init__(self,age):
            self.age = age

class Subject(Student):
        pass

obj = Student(21)
obj1 = Subject(2)

print(obj.age)
print(obj1.age)
```

Output:

```
21
2
```

2) Private Access Modifier

- Private members of a class (variables or methods) are those members which are only accessible inside the class. We cannot use private members outside of class.
- It is also not possible to inherit the private members of any class (parent class) to derived class (child class).
- Syntax to declare the private member of a class are:
  - ✓ Any variable in a class followed by **self** keyword and the variable name starting with double underscore ie. **self.__varName** are the private variables.

Example:

```
class Student:
   def __init__(self):
      self.name = "Adams Boi"  # Public
      self.__age = 39        # Private

class Subject(Student):
   pass

# object creation
obj = Student()
obj1 = Subject()

# calling using object of Student class
print(obj.name)  # No Error
print(obj1.name) # No Error

# calling using object of Subject class
#obj.__age # Error
#obj1.__age # Error

print(self.__age)

print(obj._Student__age)  # To Access the Private member

print(obj1._Student__age) # To Access the Private member

print(obj._Subject__age) # Error
print(obj1._Subject__age) # Error
```

Output:

```
Adams Boi
Adams Boi
39
39
```

3) Protected Access Modifier

- 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.
- The syntax we follow to make any variable **protected** is:
  - ✓ to write variable name/ function name followed by a single underscore (_) ie. **_varName, _funcName**

Example:

```
# defining a class Employee
class Employee:
   def __init__(self, name, sal):
      self._name = name;   # protected attribute
      self._sal = sal;     # protected attribute

emp = Employee("Captain", 10000);
emp._sal;
```

Output:

1000

Example:

```
class Student:
   def __init__(self):
      self._name = "Alice"

   def _funName(self):
      return "Hello"

class Subject(Student):
   pass

obj = Student()
obj1 = Subject()

# calling by object of Student class
print(obj._name)
print(obj._funName())

# calling by object of Subject class
print(obj1._name)
print(obj1._funName())
```

Output:

Alice
Hello
Alice
Hello

Example:
#program to illustrate access modifiers of a class

```
# super class
class Super:

    # public data member
    var1 = None

    # protected data member
    _var2 = None

    # private data member
    __var3 = None

    # constructor
    def __init__(self, var1, var2, var3):
        self.var1 = var1
        self._var2 = var2
        self.__var3 = var3

   # public member function
    def displayPublicMembers(self):

        # accessing public data members
        print("Public Data Member: ", self.var1)

    # protected member function
    def _displayProtectedMembers(self):

        # accessing protected data members
        print("Protected Data Member: ", self._var2)

    # private member function
    def __displayPrivateMembers(self):

        # accessing private data members
        print("Private Data Member: ", self.__var3)

    # public member function
    def accessPrivateMembers(self):

        # accessing private member function
        self.__displayPrivateMembers()

# derived class
class Sub(Super):

     # constructor
     def __init__(self, var1, var2, var3):
            Super.__init__(self, var1, var2, var3)

     # public member function
     def accessProtectedMembers(self):

            # accessing protected member functions of super class
            self._displayProtectedMembers()

# creating objects of the derived class
obj = Sub("Python", 4, "Python !")

# calling public member functions of the class
obj.displayPublicMembers()
```

```
obj.accessProtectedMembers()
obj.accessPrivateMembers()

# Object can access protected member
print("Object is accessing protected member:", obj._var2)

# object can not access private member, so it will generate Attribute error
#print(obj.__var3)
```

Output:

```
Public Data Member:  Python
Protected Data Member:  4
Private Data Member:  Python !
Object is accessing protected member: 4
```

Python pass Statement

- The pass statement is used as a placeholder for future code.
- **When the pass statement is executed, nothing happens**.
- It can be used **when a statement is required syntactically but the program requires no action.**

- If we use an **if**, **elif, else, functions, class, for loops, while loops** we need to define their body or block of code correspond to them, if we don't the python interpreter will throw an error. So, to overcome this error you can use the **pass** keyword as a body of the corresponding statement and the statement do nothing and throw no error.

  Example

  Using the pass keyword in a function definition:

```
def myfunction():
  pass
```

  Example

  Using the pass keyword in a class definition:

```
class Person:
  pass
```

  Example

  Using the pass keyword in an if statement:

```
a = 33
b = 200

if b > a:
  pass
```