Al-Mustaqbal University
College of Sciences
Intelligent Medical System Department

جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

كلية العلوم
قسم الانظمة الطبية الذكية

Lecture: (8)

Abstraction and Interfaces in OOP

Subject: Object oriented programming I
Class: Second

Dr. Maytham N. Meqdad

Abstraction and Interfaces in OOP

Data abstraction is one of the most essential concepts of Python OOPs which is used to hide irrelevant details from the user and show the details that are relevant to the users.

A simple example of this can be a car. A car has an accelerator, clutch, and break and we all know that pressing an accelerator will increase the speed of the car and applying the brake can stop the car but we don't know the internal mechanism of the car and how these functionalities can work this detail hiding is known as data abstraction.

To understand data abstraction we should be aware of the below basic concepts:

- OOP concepts in Python
- Classes in Python
- Abstract classes in Python

Importance of Data Abstraction

It enables programmers to hide complex implementation details while just showing users the most crucial data and functions. This abstraction makes it easier to design modular and well-organized code, makes it simpler to understand and maintain, promotes code reuse, and improves developer collaboration.

Data Abstraction in Python

Data abstraction in Python is a programming concept that hides complex implementation details while exposing only essential information and functionalities to users. In Python, we can achieve data abstraction by using abstract classes and abstract classes can be created using abc (abstract base class) module and abstract method of abc module.

Abstraction classes in Python

Abstract class is a class in which one or more abstract methods are defined. When a method is declared inside the class without its implementation is known as abstract method.

Abstract Method: In Python, abstract method feature is not a default feature. To create abstract method and abstract classes we have to import the "**ABC**" and "**abstractmethod**" classes from abc (Abstract Base Class) library. Abstract method of base class force its child class to write the implementation of the all abstract methods defined in base class. If we do not implement the abstract methods of base class in the child class then our code will give error. In the below code **method_1** is a abstract method created using @abstractmethod decorator.

```
from abc import ABC, abstractmethod
class BaseClass(ABC):
    @abstractmethod
    def method_1(self):
         #empty body
         pass
```

Concrete Method: Concrete methods are the methods defined in an abstract base class with their complete implementation. Concrete methods are required to avoid reprication of code in subclasses. For example, in abstract base class there may be a method that implementation is to be same in all its subclasses, so we write the implementation of that method in abstract base class after which we do not need to write implementation of the concrete method again and again in every subclass. In the below code **startEngine** is a concrete method**.**

```
class Car(ABC):
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
        self.engine_started = True

    def startEngine(self):
        if not self.engine_started:
            print(f"Starting the {self.model}'s engine.")
            self.engine_started = True
        else:
            print("Engine is already running.")
```

Steps to Create Abstract Base Class and Abstract Method:

1. Firstly, we import ABC and abstractmethod class from abc (Abstract Base Class) library.

2. Create a BaseClass that inherits from the ABC class. In Python, when a class inherits from ABC, it indicates that the class is intended to be an abstract base class.

3. Inside BaseClass we declare an abstract method named "method_1" by using "abstractmethod" decorater. Any subclass derived from BaseClass must implement this **method_1** method. We write pass in this method which indicates that there is no code or logic in this method.

```
from abc import ABC, abstractmethod
class BaseClass(ABC):
    @abstractmethod
    def method_1(self):
         #empty body
         pass
```

Implementation of Data Abstraction in Python

In the below code, we have implemented data abstraction using abstract class and method. Firstly, we import the required modules or classes from abc library then we create a base class 'Car' that inherited from 'ABC' class that we have imported. Inside base class we create init function, abstract function and non-abstract functions. To declare abstract function printDetails we use "**@abstractmethod**" decorator. After that we create child class hatchback and suv. Since, these child classes inherited from abstract class so, we need to write the implementation of all abstract function declared in the base class. We write the implementation of abstract method in both child class. We create an instance of a child class and call the printDetails method. In this way we can achieve the data abstraction.

```
# Import required modules
from abc import ABC, abstractmethod

# Create Abstract base class
class Car(ABC):
    def __init__(self, brand, model, year):
        self.brand = brand
```

```
        self.model = model
        self.year = year

    # Create abstract method
    @abstractmethod
    def printDetails(self):
        pass

    # Create concrete method
    def accelerate(self):
        print("Speed up ...")

    def break_applied(self):
        print("Car stopped")

# Create a child class
class Hatchback(Car):
    def printDetails(self):
        print("Brand:", self.brand)
        print("Model:", self.model)
        print("Year:", self.year)

    def sunroof(self):
        print("Not having this feature")

# Create a child class
class Suv(Car):
    def printDetails(self):
        print("Brand:", self.brand)
        print("Model:", self.model)
        print("Year:", self.year)

    def sunroof(self):
        print("Available")

# Create an instance of the Hatchback class
car1 = Hatchback("Maruti", "Alto", "2022")

# Call methods
car1.printDetails()
car1.accelerate()
car1.sunroof()
```

Output

```
Brand: Maruti
Model: Alto
Year: 2022
speed up ...
```

```
from abc import ABC, abstractmethod

# كلاس مجرد
class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

# كلاس يرث من Animal ويطبق الدالة sound
class Dog(Animal):
    def sound(self):
        return "Woof"

class Cat(Animal):
    def sound(self):
        return "Meow"

# استخدام التجريد
animals = [Dog(), Cat()]

for animal in animals:
    print(animal.sound())
```

```
from abc import ABC, abstractmethod

# كلاس مجرد يمثل واجهة الدفع
class Payment(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass

# كلاس الدفع باستخدام البطاقة الائتمانية
class CreditCardPayment(Payment):
    def process_payment(self, amount):
        print(f"Processing credit card payment of ${amount}")

# كلاس الدفع عبر PayPal
class PayPalPayment(Payment):
    def process_payment(self, amount):
        print(f"Processing PayPal payment of ${amount}")

# كلاس الدفع عبر المحفظة الإلكترونية
class WalletPayment(Payment):
    def process_payment(self, amount):
        print(f"Processing wallet payment of ${amount}")

# استخدام التجريد
def make_payment(payment_method, amount):
```

```
    payment_method.process_payment(amount)

# تجريب طرق الدفع المختلفة
credit_card = CreditCardPayment()
paypal = PayPalPayment()
wallet = WalletPayment()

make_payment(credit_card, 100)   # الدفع عن طريق البطاقة الائتمانية
make_payment(paypal, 200)        # الدفع عبر PayPal
make_payment(wallet, 50)         # الدفع عبر المحفظة الإلكترونية
```

https://www.programiz.com/online-compiler/6dsr5GUN2h36L

```
from abc import ABC, abstractmethod

# كلاس مجرد يمثل المركبة
class Vehicle(ABC):
    @abstractmethod
    def move(self):
        pass

# كلاس يمثل السيارة ويرث من Vehicle
class Car(Vehicle):
    def move(self):
        print("The car is driving on the road.")

# كلاس يمثل الدراجة ويرث من Vehicle
class Bicycle(Vehicle):
    def move(self):
        print("The bicycle is being pedaled on the path.")

# كلاس يمثل القارب ويرث من Vehicle
class Boat(Vehicle):
    def move(self):
        print("The boat is sailing on the water.")

# دالة تستخدم التجريد لنقل المركبات المختلفة
def transport(vehicle):
    vehicle.move()

# تجريب أنواع المركبات المختلفة
car = Car()
bicycle = Bicycle()
boat = Boat()

transport(car)      # السيارة
transport(bicycle)  # الدراجة
transport(boat)     # القارب
```

https://www.programiz.com/online-compiler/1tHom2CeSMDwT

```
from abc import ABC, abstractmethod

# كلاس مجرد يمثل الموظف
class Employee(ABC):
    @abstractmethod
    def calculate_salary(self):
        pass

# كلاس يمثل الموظف بدوام كامل ويرث من Employee
class FullTimeEmployee(Employee):
    def __init__(self, monthly_salary):
        self.monthly_salary = monthly_salary

    def calculate_salary(self):
        return f"Full-time employee salary is: ${self.monthly_salary}"

# كلاس يمثل الموظف المستقل ويرث من Employee
class Freelancer(Employee):
    def __init__(self, hourly_rate, hours_worked):
        self.hourly_rate = hourly_rate
        self.hours_worked = hours_worked

    def calculate_salary(self):
        return f"Freelancer salary is: ${self.hourly_rate * self.hours_worked}"

# دالة تستخدم التجريد لحساب رواتب الموظفين المختلفة
def print_salary(employee):
    print(employee.calculate_salary())

# تجريب أنواع الموظفين المختلفة
full_time_employee = FullTimeEmployee(3000)
freelancer = Freelancer(25, 120)

print_salary(full_time_employee)  # موظف بدوام كامل
print_salary(freelancer)          # موظف مستقل
```

https://www.programiz.com/online-compiler/68tzUg5cxDBFo

```
from abc import ABC, abstractmethod

# كلاس مجرد يمثل المريض
class Patient(ABC):
    @abstractmethod
    def assess_health(self):
        pass

# كلاس يمثل مريض السكري ويرث من Patient
class DiabetesPatient(Patient):
    def __init__(self, blood_sugar_level):
        self.blood_sugar_level = blood_sugar_level

    def assess_health(self):
        if self.blood_sugar_level < 70:
            return "Diabetes patient condition: Hypoglycemia (low blood sugar)"
        elif self.blood_sugar_level > 180:
            return "Diabetes patient condition: Hyperglycemia (high blood sugar)"
        else:
            return "Diabetes patient condition: Normal blood sugar level"

# كلاس يمثل مريض ضغط الدم ويرث من Patient
class HypertensionPatient(Patient):
    def __init__(self, systolic, diastolic):
        self.systolic = systolic
        self.diastolic = diastolic

    def assess_health(self):
        if self.systolic > 140 or self.diastolic > 90:
            return "Hypertension patient condition: High blood pressure"
        elif self.systolic < 90 or self.diastolic < 60:
            return "Hypertension patient condition: Low blood pressure"
        else:
            return "Hypertension patient condition: Normal blood pressure"

# دالة تستخدم التجريد لتقييم حالة المرضى المختلفة
def evaluate_patient(patient):
    print(patient.assess_health())

# تجريب أنواع المرضى المختلفة
diabetes_patient = DiabetesPatient(160)
```

```
hypertension_patient = HypertensionPatient(130, 85)

evaluate_patient(diabetes_patient)      # تقييم حالة مريض السكري
evaluate_patient(hypertension_patient)  # تقييم حالة مريض ضغط الدم
```

https://www.programiz.com/online-compiler/2pgSM2odqYlr1

==========

Python-interface module

In object-oriented languages like Python, the interface is a collection of method signatures that should be provided by the implementing class. Implementing an interface is a way of writing an organized code and achieve abstraction.

The package **zope.interface** provides an implementation of "object interfaces" for Python. It is maintained by the Zope Toolkit project. The package exports two objects, 'Interface' and 'Attribute' directly. It also exports several helper methods. It aims to provide stricter semantics and better error messages than Python's built-in abc module.

Declaring interface

In python, interface is defined using python class statements and is a subclass of **interface.Interface** which is the parent interface for all interfaces.

Syntax :

```
class IMyInterface(zope.interface.Interface):
    # methods and attributes
```

Example

Output :

```
<class zope.interface.interface.InterfaceClass>
__main__
MyInterface
```

```
<zope.interface.interface.Attribute object at 0x00000270A8C74358>
<class 'zope.interface.interface.Attribute'>
```

Implementing interface

Interface acts as a blueprint for designing classes, so interfaces are implemented using **implementer** decorator on class. If a class implements an interface, then the instances of the class provide the interface. Objects can provide interfaces directly, in addition to what their classes implement.

Syntax :

```
@zope.interface.implementer(*interfaces)
class Class_name:
    # methods
```

Example

```
import zope.interface


class MyInterface(zope.interface.Interface):
    x = zope.interface.Attribute("foo")
    def method1(self, x):
        pass
    def method2(self):
        pass

@zope.interface.implementer(MyInterface)
class MyClass:
    def method1(self, x):
        return x**2
    def method2(self):
        return "foo"
```

We declared that MyClass implements MyInterface. This means that instances of MyClass provide MyInterface.

Methods

- **implementedBy(class) –** returns a boolean value, True if class implements the interface else False

- **providedBy(object) –** returns a boolean value, True if object provides the interface else False
- **providedBy(class) –** returns False as class does not provide interface but implements it
- **list(zope.interface.implementedBy(class)) –** returns the list of interfaces implemented by a class
- **list(zope.interface.providedBy(object)) –** returns the list of interfaces provided by an object.
- **list(zope.interface.providedBy(class)) –** returns empty list as class does not provide interface but
  implements it.

```
import zope.interface


class MyInterface(zope.interface.Interface):
    x = zope.interface.Attribute('foo')
    def method1(self, x, y, z):
        pass
    def method2(self):
        pass

@zope.interface.implementer(MyInterface)
class MyClass:
    def method1(self, x):
        return x**2
    def method2(self):
        return "foo"
obj = MyClass()

# ask an interface whether it
# is implemented by a class:
print(MyInterface.implementedBy(MyClass))

# MyClass does not provide
# MyInterface but implements it:
print(MyInterface.providedBy(MyClass))

# ask whether an interface
# is provided by an object:
print(MyInterface.providedBy(obj))

# ask what interfaces are
```

```
# implemented by a class:
print(list(zope.interface.implementedBy(MyClass)))

# ask what interfaces are
# provided by an object:
print(list(zope.interface.providedBy(obj)))

# class does not provide interface
print(list(zope.interface.providedBy(MyClass)))
```

Output :

```
True
False
True
[<InterfaceClass __main__.MyInterface>]
[<InterfaceClass __main__.MyInterface>]
[]
```

Interface Inheritance

Interfaces can extend other interfaces by listing the other interfaces as base interfaces.

Functions

- **extends(interface) –** returns boolean value, whether one interface extends another.
- **isOrExtends(interface) –** returns boolean value, whether interfaces are same or one extends another.
- **isEqualOrExtendedBy(interface) –** returns boolean value, whether interfaces are same or one is extended by another.

```
import zope.interface


class BaseI(zope.interface.Interface):
    def m1(self, x):
        pass
    def m2(self):
        pass
```

```
class DerivedI(BaseI):
    def m3(self, x, y):
        pass

@zope.interface.implementer(DerivedI)
class cls:
    def m1(self, z):
        return z**3
    def m2(self):
        return 'foo'
    def m3(self, x, y):
        return x ^ y

# Get base interfaces
print(DerivedI.__bases__)

# Ask whether baseI extends
# DerivedI
print(BaseI.extends(DerivedI))

# Ask whether baseI is equal to
# or is extended by DerivedI
print(BaseI.isEqualOrExtendedBy(DerivedI))

# Ask whether baseI is equal to
# or extends DerivedI
print(BaseI.isOrExtends(DerivedI))

# Ask whether DerivedI is equal
# to or extends BaseI
print(DerivedI.isOrExtends(DerivedI))
```

Output :

```
(<InterfaceClass __main__.BaseI>, )
False
True
False
True
```