Interfaces and Abstract Classes

Abstract Classes

Abstract classes in Python are classes that cannot be instantiated on their own and are meant to be subclassed by other classes. Abstract classes may contain abstract methods, which are declared but not implemented in the abstract class. Subclasses must provide concrete implementations for these abstract methods.

Why use an abstract class?

- It is used to achieve abstraction.
- It can have methods with an implementation and methods without an implementation.
- When you don't want to provide the implementation of a method, you can make it abstract.
- When you don't want to allow the instantiation of a class, you can make it abstract.

How to create an abstract class?

Let us create an abstract class for a Person. In Python, you can create an abstract class by inheriting from the `ABC` class in the `abc` module. Similarly, you can create an abstract method by using the `abstractmethod` decorator.

```python
from abc import ABC, abstractmethod

class Person(ABC):

    @abstractmethod
    def get_name(self):
        pass

    @abstractmethod
    def get_email(self):
        pass
```

Abstract Methods

Python does not have built-in support for abstract methods like Java, where you do not have to provide an implementation for abstract methods. In Python, you can either add a dummy implementation by using the `pass` statement or raise a `NotImplementedError` exception.

Another interesting way is to use the `...` ellipsis, which is a special literal that can be used in place of an expression or a statement. It is used to indicate incomplete code. When the interpreter finds an ellipsis, it raises a `SyntaxError` exception.

Now let's create a class that extends the Person abstract class:

```python
class User(Person):
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def get_name(self):
        return self.name

    def get_email(self):
        return self.email
```

In the above example, we have extended from the `Person` class by adding it to the class declaration. We have also provided implementations for the abstract methods `get_name` and `get_email`.

You can create an instance of the `User` class and call the `get_name` and `get_email` methods:

```
user = User("Tantia Tope", "t@t.in")
print(user.get_name())
print(user.get_email())
```

Interfaces

> It can be thought of as a blueprint of behavior. It is used to achieve abstraction.

An interface is a reference type in Java. It is similar to a class, but it cannot be instantiated. It can contain only constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Why use an interface?

- It is used to achieve abstraction.
- Define a common behavior for related classes.

In Python, interfaces are not explicitly defined as a language construct, as in some other programming languages like Java or C#. Instead, interfaces are typically represented by classes that define a set of methods without providing any implementation. A class in Python can be considered to implement an interface if it provides implementations for all the methods declared in that interface.

Why does Python not have interfaces?

There are multiple reasons why Python does not have interfaces as a language construct:

- `Duck Typing` - Python uses duck typing, which means that the type or the

class of an object is less important than the methods it defines. If an object has all the methods defined in an interface, then it can be considered to implement that interface.

- `Multiple Inheritance` - Java uses interfaces to achieve multiple inheritance since normal classes cannot inherit from multiple classes. Python, on the other hand, allows multiple inheritance, so interfaces are not needed to achieve multiple inheritance.

If you want to create a base class that is a blueprint for other classes, you can create them similar to abstract classes. Let's say we want to create a class `EmailSender`. To make it extensible to other forms of communication, we can create an abstract class `NotificationSender`:

```
from abc import ABC, abstractmethod

class NotificationSender(ABC):
    @abstractmethod
    def send(self, person: Person, message: str):
        pass
```

This class can be extended by other classes that want to implement the `send` method. Let's create a class `EmailSender` that extends the `NotificationSender` class:

```
class EmailSender(NotificationSender):
    def send(self, person: Person, message: str):
        print(f"Sending email to {person.get_email()}")
```

This how you can use abstract classes in Python for the same purpose as interfaces in Java.

Association

> Association represents a relationship between two or more classes.

Association is a relationship where all objects have their own lifecycle and there is no owner. Let's take an example of a Car and an Engine. A car can have an engine, but an engine can

also be used in multiple cars. Both have their own lifecycle and there is no ownership between the objects and both can exist without each other.

```
class Engine:
    def __init__(self, capacity: int):
        self.capacity = capacity

    def start(self):
        print("Engine started")

    def stop(self):
        print("Engine stopped")

class Car:
    def __init__(self, engine: Engine):
        self.engine = engine

    def start(self):
        self.engine.start()

    def stop(self):
        self.engine.stop()
```

In the above example, the `Car` class has a reference to the `Engine` class. The `Car` class can use the `Engine` class to start and stop the engine. The `Engine` class can also be used by other classes like `Truck` or `Bus`.

When a class contains a reference to another class, we call it an association. In the above example, the `Car` class is associated with the `Engine` class. There are two types of association:

Composition

Composition is a type of association where one class, known as the "composite" class, contains an object of another class within its own structure. The composed class has a strong relationship with the component class, meaning that the component's lifecycle is managed by the composite. **If the composite is destroyed, the component is typically also destroyed.**

In the provided example, the Car class has a composition relationship with the Engine class. The Car class encapsulates an instance of the Engine class as one of its attributes. This composition allows the Car class to delegate certain functionalities, such as starting and stopping the engine, to the encapsulated Engine object.

```
class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        self.engine.start()

    def stop(self):
        self.engine.stop()
```

Aggregation

Aggregation is another form of association where one class, the "aggregate" class, contains a reference to another class, the "component" class. However, in aggregation, the component's lifecycle is not necessarily dependent on the aggregate. The component can exist independently of the aggregate.

Unlike composition, aggregation represents a weaker relationship. If the aggregate is destroyed, the component can still exist. In Python, aggregation is often implemented using references or attributes.

```
class Department:
    def __init__(self, name):
        self.name = name
        self.employees = []

    def add_employee(self, employee):
        self.employees.append(employee)

class Employee:
    def __init__(self, name):
        self.name = name

# Aggregation example
engineering_department = Department("Engineering")
alice = Employee("Alice")
bob = Employee("Bob")

engineering_department.add_employee(alice)
engineering_department.add_employee(bob)
```

In this example, the Department class aggregates Employee objects. If the Department is destroyed, the Employee objects can still exist independently.