**Abstract Classes in Python**

An abstract class can be considered a blueprint for other [classes](https://www.geeksforgeeks.org/python-classes-and-objects/). It allows you to create a set of methods that must be created within any child classes built from the abstract class.

A class that contains one or more abstract methods is called an **abstract class**. An **abstract method** is a method that has a declaration but does not have an implementation.

We use an abstract class while we are designing large functional units or when we want to provide a common interface for different implementations of a component.

**Abstract Base Classes in Python**

By defining an abstract base class, you can define a common **Application Program Interface(API)** for a set of subclasses. This capability is especially useful in situations where a third party is going to provide implementations, such as with plugins, but can also help you when working in a large team or with a large code base where keeping all classes in your mind is difficult or not possible.

**Working on Python Abstract classes**

By default, [Python](https://www.geeksforgeeks.org/python-programming-language/) does not provide **abstract classes**. Python comes with a module that provides the base for defining A**bstract Base classes(ABC)**and that module name is **ABC**.

**ABC** works by decorating methods of the base class as an abstract and then registering concrete classes as implementations of the abstract base. A method becomes abstract when decorated with the keyword @abstractmethod.

**Example 1:**

This code defines an abstract base class called “**Polygon**” using the ABC (Abstract Base Class) module in Python. The “Polygon” class has an abstract method called “**noofsides**” that needs to be implemented by its subclasses.

There are four subclasses of “Polygon” defined: “Triangle,” “Pentagon,” “Hexagon,” and “Quadrilateral.” Each of these subclasses overrides the “noofsides” method and provides its own implementation by printing the number of sides it has.

In the driver code, instances of each subclass are created, and the “noofsides” method is called on each instance to display the number of sides specific to that shape.

Python

*# Python program showing*

*# abstract base class work*

**from** **abc** **import** ABC, abstractmethod

**class** **Polygon**(ABC):

@abstractmethod

**def** noofsides(self):

**pass**

**class** **Triangle**(Polygon):

*# overriding abstract method*

**def** noofsides(self):

print("I have 3 sides")

**class** **Pentagon**(Polygon):

*# overriding abstract method*

**def** noofsides(self):

print("I have 5 sides")

**class** **Hexagon**(Polygon):

*# overriding abstract method*

**def** noofsides(self):

print("I have 6 sides")

**class** **Quadrilateral**(Polygon):

*# overriding abstract method*

**def** noofsides(self):

print("I have 4 sides")

*# Driver code*

R = Triangle()

R.noofsides()

K = Quadrilateral()

K.noofsides()

R = Pentagon()

R.noofsides()

K = Hexagon()

K.noofsides()

**Output**

I have 3 sides

I have 4 sides

I have 5 sides

I have 6 sides

**Example 2:**

Here, This code defines an abstract base class called “Animal” using the ABC (Abstract Base Class) module in Python. The “Animal” class has a non-abstract method called “move” that does not have any implementation. There are four subclasses of “Animal” defined: “Human,” “Snake,” “Dog,” and “Lion.” Each of these subclasses overrides the “move” method and provides its own implementation by printing a specific movement characteristic.

Python

*# Python program showing*

*# abstract base class work*

**from** **abc** **import** ABC, abstractmethod

**class** **Animal**(ABC):

**def** move(self):

**pass**

**class** **Human**(Animal):

**def** move(self):

print("I can walk and run")

**class** **Snake**(Animal):

**def** move(self):

print("I can crawl")

**class** **Dog**(Animal):

**def** move(self):

print("I can bark")

**class** **Lion**(Animal):

**def** move(self):

print("I can roar")

*# Driver code*

R = Human()

R.move()

K = Snake()

K.move()

R = Dog()

R.move()

K = Lion()

K.move()

**Output**

I can walk and run

I can crawl

I can bark

I can roar

**Implementation of Abstract through Subclass**

By subclassing directly from the base, we can avoid the need to register the class explicitly. In this case, the Python class management is used to recognize **Plugin implementation** as implementing the **abstract PluginBase**.

Python

*# Python program showing*

*# implementation of abstract*

*# class through subclassing*

**import** **abc**

**class** **parent**:

**def** geeks(self):

**pass**

**class** **child**(parent):

**def** geeks(self):

print("child class")

*# Driver code*

print( issubclass(child, parent))

print( isinstance(child(), parent))

**Output**

True

True

A side-effect of using direct subclassing is, it is possible to find all the implementations of your plugin by asking the base class for the list of known classes derived from it.

**Concrete Methods in Abstract Base Classes**

Concrete classes contain only concrete (normal) methods whereas abstract classes may contain both concrete methods and abstract methods.

The concrete class provides an implementation of abstract methods, the abstract base class can also provide an implementation by invoking the methods via super(). Let look over the example to invoke the method using super():

Python

*# Python program invoking a*

*# method using super()*

**from** **abc** **import** ABC

**class** **R**(ABC):

**def** rk(self):

print("Abstract Base Class")

**class** **K**(R):

**def** rk(self):

super().rk()

print("subclass ")

*# Driver code*

r = K()

r.rk()

**Output**

Abstract Base Class

subclass

In the above program, we can invoke the methods in abstract classes by using **super().**

**Abstract Properties in Python**

Abstract classes include attributes in addition to methods, you can require the attributes in concrete classes by defining them with @abstractproperty.

Python

*# Python program showing*

*# abstract properties*

**import** **abc**

**from** **abc** **import** ABC, abstractmethod

**class** **parent**(ABC):

@abc.abstractproperty

**def** geeks(self):

**return** "parent class"

**class** **child**(parent):

@property

**def** geeks(self):

**return** "child class"

**try**:

r = parent()

print(r.geeks)

**except** **Exception** **as** err:

print(err)

r = child()

print(r.geeks)

**Output**

Can't instantiate abstract class parent with abstract methods geeks

child class

In the above example, the Base class cannot be instantiated because it has only an abstract version of the property-getter method.

**Abstract Class Instantiation**

Abstract classes are incomplete because they have methods that have nobody. If Python allows creating an object for abstract classes then using that object if anyone calls the abstract method, but there is no actual implementation to invoke.

So, we use an abstract class as a template and according to the need, we extend it and build on it before we can use it. Due to the fact, an abstract class is not a concrete class, it cannot be instantiated. When we create an object for the abstract class it **raises an error**.

Python

*# Python program showing*

*# abstract class cannot*

*# be an instantiation*

**from** **abc** **import** ABC,abstractmethod

**class** **Animal**(ABC):

@abstractmethod

**def** move(self):

**pass**

**class** **Human**(Animal):

**def** move(self):

print("I can walk and run")

**class** **Snake**(Animal):

**def** move(self):

print("I can crawl")

**class** **Dog**(Animal):

**def** move(self):

print("I can bark")

**class** **Lion**(Animal):

**def** move(self):

print("I can roar")

c=Animal()

**Output:**

Traceback (most recent call last):  
 File "/home/ffe4267d930f204512b7f501bb1bc489.py", line 19, in   
 c=Animal()  
TypeError: Can't instantiate abstract class Animal with abstract methods move

**Abstract Classes in Python – FAQs**

**What is the Purpose of Abstract Classes in Python?**

*Abstract classes in Python are used to define a common interface for a group of subclasses. They enforce certain methods to be implemented by any subclass, ensuring a consistent API and encouraging code reuse. Abstract classes themselves cannot be instantiated and are meant to be inherited by other classes that provide concrete implementations of the abstract methods.*

**How to Define and Use an Abstract Class in Python?**

*To define an abstract class in Python, you typically use the abc (Abstract Base Class) module. An abstract class is defined by inheriting from ABC and using the @abstractmethod decorator to define abstract methods that must be implemented by subclasses.*

***Example:***

*from abc import ABC, abstractmethod  
  
class Animal(ABC):  
 @abstractmethod  
 def sound(self):  
 pass  
  
class Dog(Animal):  
 def sound(self):  
 return "Bark"  
  
dog = Dog()  
print(dog.sound()) # Output: Bark*

**What is the ‘abc' Module and How is It Used in Python?**

*The abc module in Python provides tools for defining abstract base classes. It includes the ABC class and the abstractmethod decorator, which are used to create abstract classes and methods.*

***Using the abc Module:***

*from abc import ABC, abstractmethod  
  
class Shape(ABC):  
 @abstractmethod  
 def area(self):  
 pass  
  
class Rectangle(Shape):  
 def area(self):  
 return 10 \* 20  
  
rectangle = Rectangle()  
print(rectangle.area()) # Output: 200*

**How to Define Abstract Methods in Python?**

*Abstract methods in Python are defined using the @abstractmethod decorator from the abc module. These methods do not provide any implementation in the abstract class and must be overridden in any concrete subclass.*

***Example:***

*from abc import ABC, abstractmethod  
  
class Vehicle(ABC):  
 @abstractmethod  
 def start\_engine(self):  
 pass  
  
class Car(Vehicle):  
 def start\_engine(self):  
 return "Car engine started"  
  
car = Car()  
print(car.start\_engine()) # Output: Car engine started*

**Why Should Abstract Classes Be Used in Python?**

*Abstract classes should be used in Python for several reasons:*

1. ***Enforcing a Contract****: Ensure that all subclasses implement certain methods, providing a consistent interface.*
2. ***Code Reusability****: Define common functionality in the abstract class and reuse it in all subclasses.*
3. ***Encouraging Design Principles****: Promote the use of design patterns and principles like SOLID.*
4. ***Reducing Code Duplication****: Define shared methods and properties once in the abstract class.*
5. ***Improved Readability and Maintainability****: Provide clear guidelines for developers about the structure and behavior expected from subclasses*