Abstract Base Classes

In this lesson, we will learn what abstract base classes are in Python and how to use them.

WE'LL COVER THE FOLLOWING

What are Abstract Base Classes?

Duck typing is useful as it simplifies the code and the user can implement the functions without worrying about the data type. But this may not be the case all the time. The user might not follow the instructions to implement the necessary steps for duck typing. To cater to this issue, Python introduced the concept of Abstract Base Classes, or **ABC**.

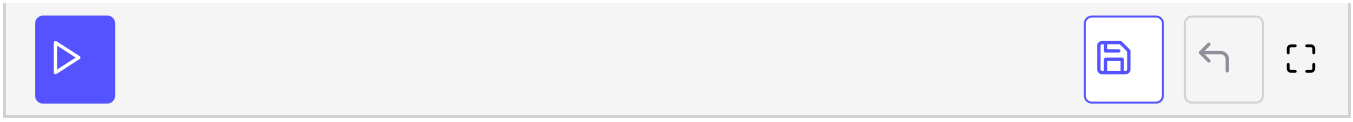
> **Abstract base classes** define a set of methods and properties that a class must implement in order to be considered a duck-type instance of that class.

Why use Abstract Base Classes?

Let's see an example below to understand why we should use abstract base classes:

```
class Shape:  # Shape is a child class of ABC
    def area(self):
        pass
```

```
    def perimeter(self):
        pass


class Square(Shape):
    def __init__(self, length):
        self.length = length

    def area(self):
        return (self.length * self.length)

    def perimeter(self):
        return (4 * self.length)


shape = Shape()
square = Square(4)
```



In the example above, you can see that an instance of `Shape` can be created even though an object from this class cannot stand on its own. `Square` class, which is the child class of `Shape`, actually implements the methods, `area()` and `perimeter()`, of the `Shape` class. `Shape` class should provide a blueprint for its child classes to implement methods in it. To prevent the user from making a `Shape` class object, we use **abstract base classes**.

Syntax

To define an abstract base class, we use the `abc` module. The abstract base class is inherited from the built-in `ABC` class. We have to use the decorator `@abstractmethod` above the method that we want to declare as an abstract method.

```
from abc import ABC, abstractmethod


class ParentClass(ABC):

    @abstractmethod
    def method(self)
```

Example

Below is a code implementation for abstract base classes:

Shape Square

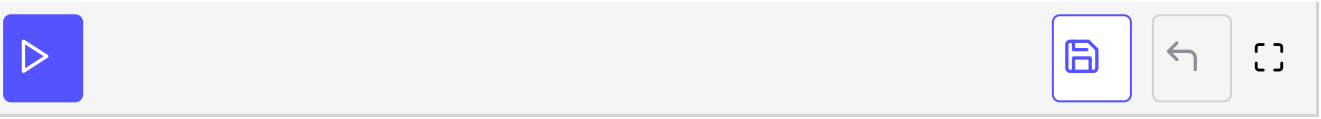
```
from abc import ABC, abstractmethod


class Shape(ABC):  # Shape is a child class of ABC
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass


class Square(Shape):
    def __init__(self, length):
        self.length = length


shape = Shape()
# this will code will not compile since Shape has abstract methods without
# method definitions in it
```

As you can see above, the code does not compile since we have not defined the abstract methods, `area` and `perimeter`, inside the parent class, `Shape`, or the child class, `Square`. Let's do it and see what happens:

Shape Square

```
from abc import ABC, abstractmethod


class Shape(ABC):  # Shape is a child class of ABC
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass


class Square(Shape):
    def __init__(self, length):
        self.length = length

    def area(self):
        return (self.length * self.length)
```

```
    def perimeter(self):
        return (4 * self.length)


shape = Shape()
# this will code will not compile since Shape has abstract methods without
# method definitions in it
```



Explanation

- Now when we define the methods, `area` and `perimeter` in the child class `Square`, an object of `Shape` cannot be instantiated, but an object of `Square` can be.
- We allow the user to have a free hand over the definition of the methods while also making sure that the methods are defined.

> **Note:** Methods with `@abstractmethod` decorators **must** be defined either in the parent class or the child class.

By using abstract base classes, we can control the classes whose objects can or cannot be created.

> Abstract methods must be defined in child classes for proper implementation of inheritance.

That is all for polymorphism in Python. Let's test your knowledge with a quick quiz!