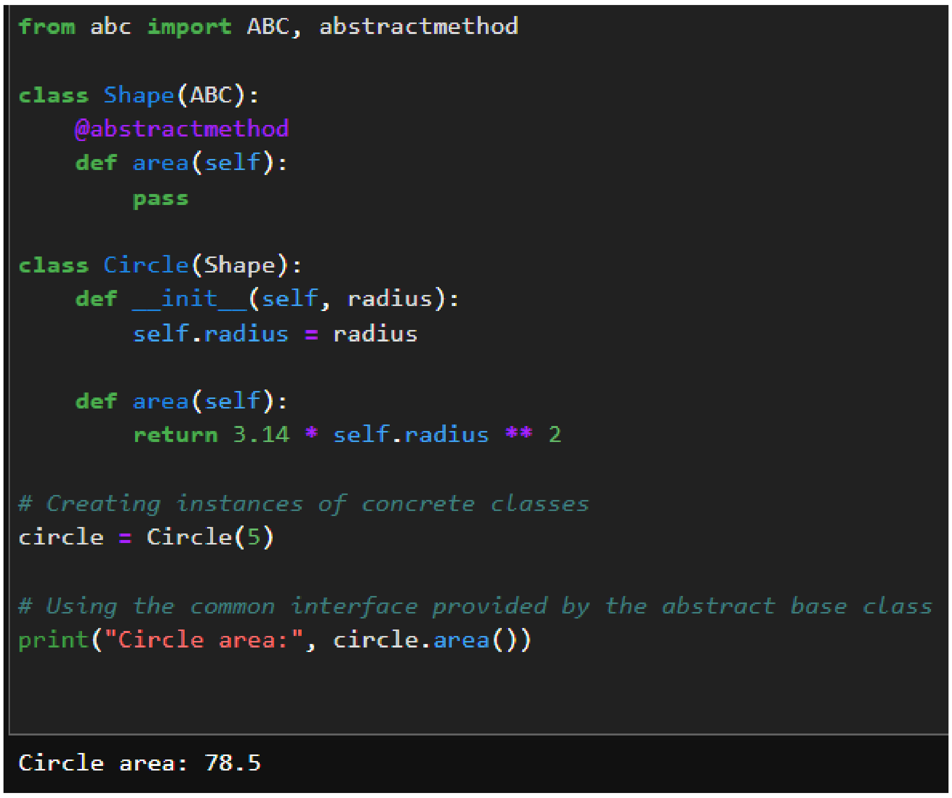
Q1. What is Abstraction in OOps? Explain with an example.

Abstraction is one of the four fundamental principles of object-oriented programming (OOP), along with encapsulation, inheritance, and polymorphism. Abstraction is the concept of simplifying complex systems by modelling classes based on their essential properties and behaviours, while hiding unnecessary details. It involves defining a clear and concise interface for interacting with objects, focusing on what an object does rather than how it does it.



```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

# Creating instances of concrete classes
circle = Circle(5)

# Using the common interface provided by the abstract base class
print("Circle area:", circle.area())
```

```
Circle area: 78.5
```

Q2. Differentiate between Abstraction and Encapsulation. Explain with an example.

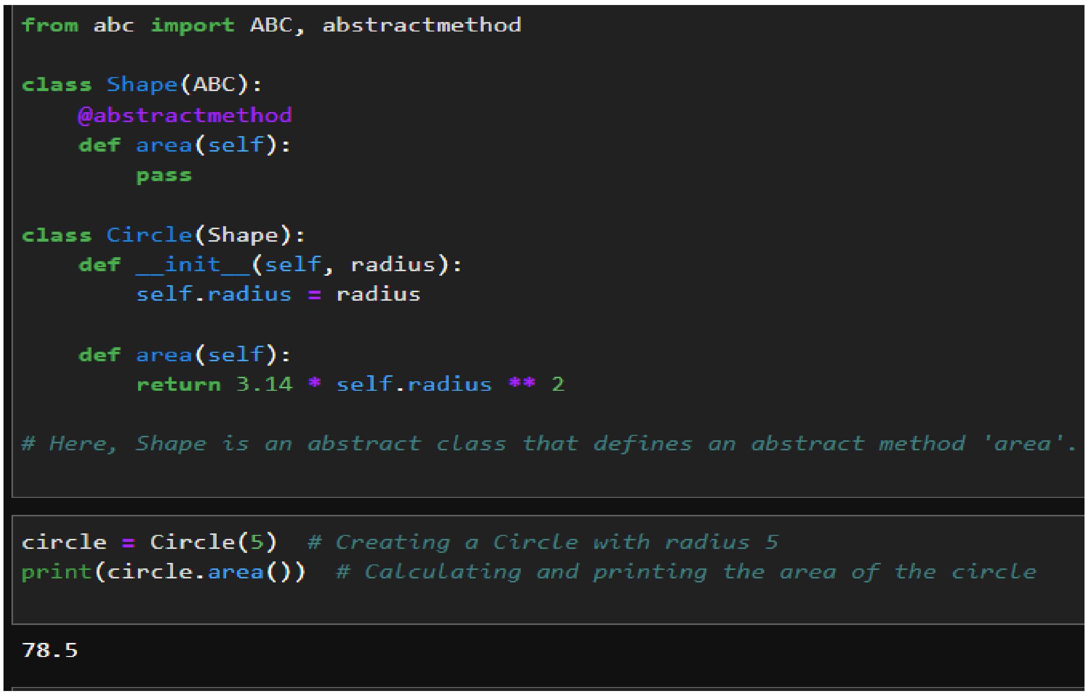
- Abstraction is more focused on defining a clear and concise interface, emphasizing what an object does at a high level, while encapsulation is about bundling data and methods, controlling access to the internal state of an object.

- Abstraction is achieved through abstract classes and interfaces, while encapsulation is achieved through access control and data hiding mechanisms.

- Abstraction abstracts away implementation details, allowing you to work with objects at a higher level without knowing how they are implemented, while encapsulation protects the integrity of an object's internal state by controlling access to it.

Abstraction and encapsulation are complementary concepts in OOP. Abstraction provides a high-level view of an object's behaviour, while encapsulation ensures that the internal details of an object are hidden and can only be accessed through well-defined interfaces. Both concepts contribute to the organization and maintainability of code.

Abstraction:

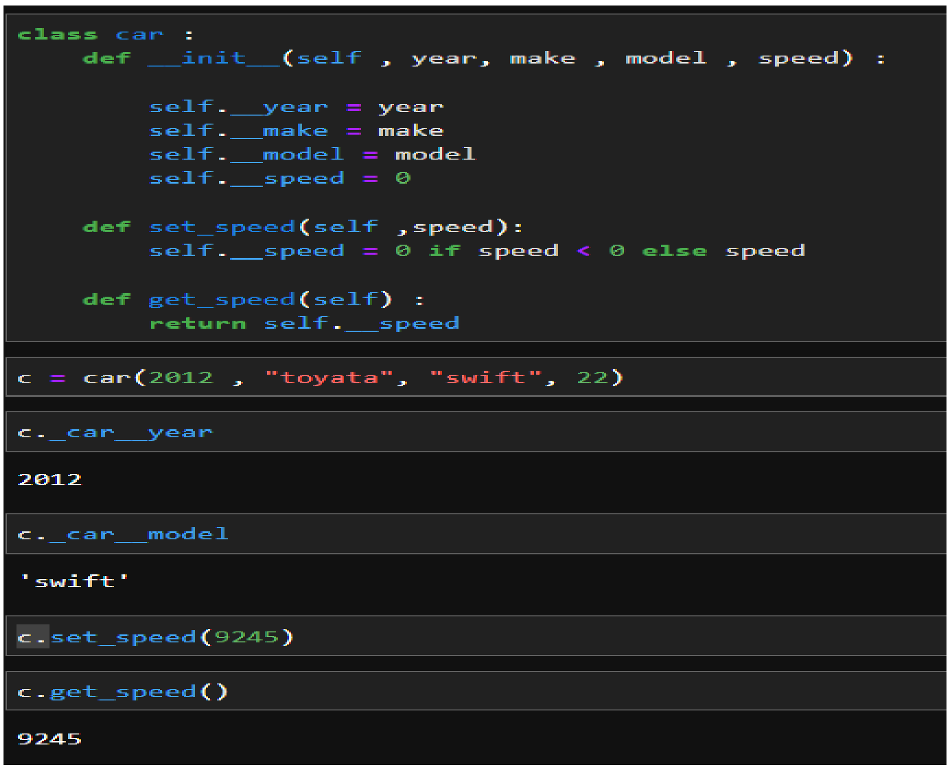


```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

# Here, Shape is an abstract class that defines an abstract method 'area'.
```

```python
circle = Circle(5)  # Creating a Circle with radius 5
print(circle.area())  # Calculating and printing the area of the circle
```

```
78.5
```

Encapsulation:



```
class car :
    def __init__(self , year, make , model , speed) :

        self.__year = year
        self.__make = make
        self.__model = model
        self.__speed = 0

    def set_speed(self ,speed):
        self.__speed = 0 if speed < 0 else speed

    def get_speed(self) :
        return self.__speed
```

```
c = car(2012 , "toyata", "swift", 22)
```

```
c._car__year
```

```
2012
```

```
c._car__model
```

```
'swift'
```

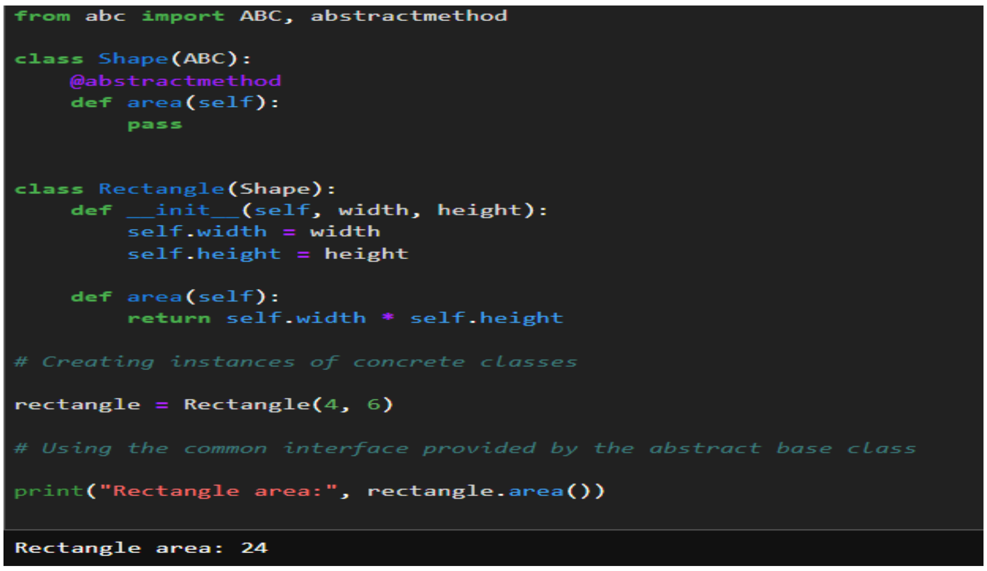
```
c.set_speed(9245)
```

```
c.get_speed()
```

```
9245
```

Q3. What is abc module in python? Why is it used?

The `abc` module in Python stands for "Abstract Base Classes." It is a module that provides a way to define abstract base classes in Python and enforce a level of structure on derived classes. Abstract base classes are used to create a blueprint for other classes and ensure that certain methods or attributes are implemented in those derived classes.



```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass


class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

# Creating instances of concrete classes

rectangle = Rectangle(4, 6)

# Using the common interface provided by the abstract base class

print("Rectangle area:", rectangle.area())
```

```
Rectangle area: 24
```

Q4. How can we achieve data abstraction?

To achieve data abstraction:

1. Define an abstract data type (ADT) that represents the data and operations on it.

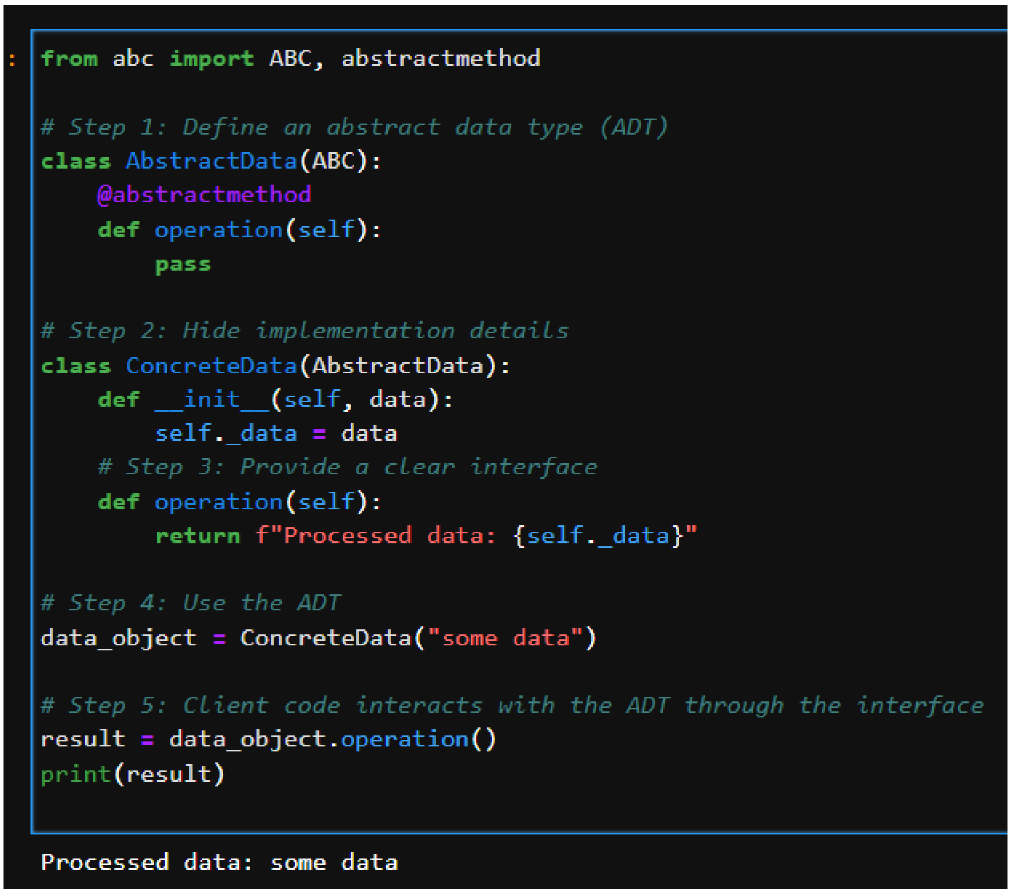
2. Hide implementation details using access control mechanisms (e.g., private attributes).

3. Provide a clear interface with methods that define how to interact with the data.

4. Use abstract classes or interfaces to define the ADT.

5. Implement concrete classes that inherit from the abstract class or interface.

6. Write client code that interacts with the ADT through the defined interface, avoiding direct access to internal data.



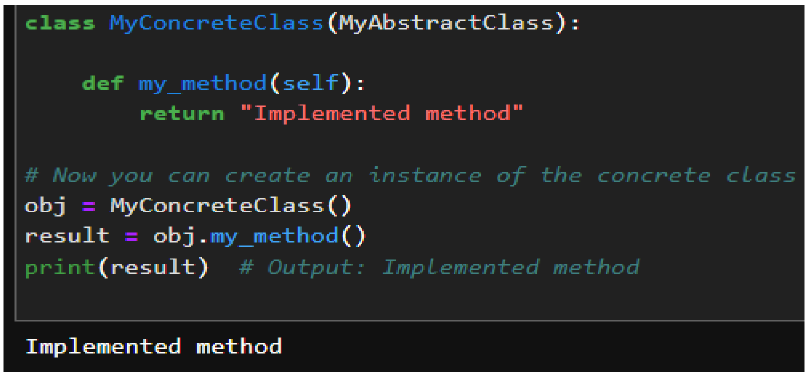
```python
from abc import ABC, abstractmethod

# Step 1: Define an abstract data type (ADT)
class AbstractData(ABC):
    @abstractmethod
    def operation(self):
        pass

# Step 2: Hide implementation details
class ConcreteData(AbstractData):
    def __init__(self, data):
        self._data = data
    # Step 3: Provide a clear interface
    def operation(self):
        return f"Processed data: {self._data}"

# Step 4: Use the ADT
data_object = ConcreteData("some data")

# Step 5: Client code interacts with the ADT through the interface
result = data_object.operation()
print(result)
```

```
Processed data: some data
```

Q5. Can we create an instance of an abstract class? Explain your answer.

No, you cannot create an instance of an abstract class in most object-oriented programming languages, including Python. Abstract classes are meant to be incomplete and serve as blueprints for other classes to inherit from. They define common attributes and method signatures that should be implemented by their concrete (non-abstract) subclasses. Abstract classes exist to provide a common interface and enforce a contract for their subclasses, but they themselves are not meant to be instantiated.

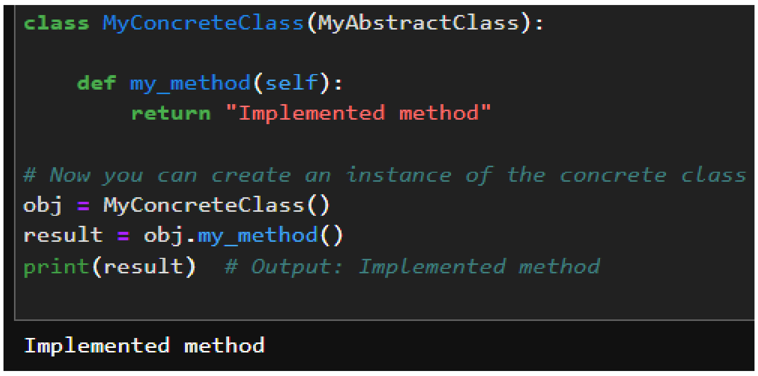
In Python, you can create an abstract class using the `abc` module, and an abstract method within an abstract class is marked with the `@abstractmethod` decorator. Attempting to create an instance of an abstract class will result in a `TypeError`.



```python
class MyConcreteClass(MyAbstractClass):

    def my_method(self):
        return "Implemented method"

# Now you can create an instance of the concrete class
obj = MyConcreteClass()
result = obj.my_method()
print(result)  # Output: Implemented method
```

```
Implemented method
```

You can create instances of concrete classes that inherit from the abstract class and provide implementations for all the abstract methods. This is the intended use of abstract classes—to serve as a foundation for creating concrete subclasses that adhere to a specific interface and contract.



```python
class MyConcreteClass(MyAbstractClass):

    def my_method(self):
        return "Implemented method"

# Now you can create an instance of the concrete class
obj = MyConcreteClass()
result = obj.my_method()
print(result)  # Output: Implemented method
```

```
Implemented method
```