# Report: Object-Oriented Programming
Abstract Base Classes(ABC)

Nguyen Do Thanh Phuc

February 2026

## 1 Introduction

Abstraction is one of the fundamental principles of Object-Oriented Programming (OOP). It allows developers to define common interfaces while hiding implementation details. In Python, Abstract Base Classes (ABCs), provided by the `abc` module, offer a formal mechanism for defining such interfaces. This report discusses the concept, purpose, and implementation of Abstract Base Classes, along with their relationship to duck typing.

## 2 Abstract Base Class

### 2.1 Defnition

In Python, an abstract base class (ABC) is a class that can't be instantiated on its own and is designed to be a blueprint for other classes, allowing you to define a common interface for a group of related classes.

### 2.2 Why Are Abstract Base Classes Needed?

Abstract classes are useful for enforcing method implementation and defining a clear interface for subclasses. By declaring abstract methods, a base class ensures that all derived classes provide their own implementations, reducing the risk of incomplete or inconsistent behavior.

In addition, abstract classes can include concrete methods that promote code reuse and reduce duplication, supporting the DRY (Do Not Repeat Yourself) principle. They also enhance readability and maintainability by establishing a consistent structure. Furthermore, abstract classes facilitate polymorphism, allowing developers to write generalized code that operates on any subclass of the abstract base class.

### 2.3 How Abstract Base Classes Work in Python

An Abstract Base Class in Python is created by inheriting from `ABC` in the `abc` module. Required methods are declared using the `@abstractmethod` decorator. A class containing abstract methods cannot be instantiated. Only when a subclass implements all abstract methods does it become a concrete class that can be instantiated.

# 3 Example Code Simulation

The following example demonstrates the use of an Abstract Base Class to define a common interface for geometric shapes:

```
from abc import ABC, abstractmethod

class Shape(ABC):

    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Circle(Shape):

    def __init__(self, radius):
        self.radius = radius

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

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

class Square(Shape):
    pass # This does not implement abstract methods area() and
       perimeter()

# shape = Shape()  # This would raise TypeError
# square = Square() # This would raise TypeError

circle = Circle(5)
print(circle.area())
print(circle.perimeter())
```

In this example, the class `Shape` defines the abstract method `area()`. The subclass `Circle` provides a concrete implementation and can therefore be instantiated successfully.

However, attempting to instantiate `Shape` directly raises a `TypeError` because it contains an abstract method. Similarly, the class `Square` does not override the abstract method `area()`, so Python also raises a `TypeError` when instantiation is attempted. This behavior demonstrates how Abstract Base Classes enforce interface compliance.