Lecture 2.10

Abstraction - abstract class (Continued)

Agenda

- 1. Recap of Abstraction Concepts
- 2. The Python 'abc' Module
- 3. Defining Abstract Base Classes (ABCs)
- 4. Abstract Methods Explained
- 5. Abstract Properties in Python
- 6. Implementing Abstract Methods & Properties
- 7. Concrete vs. Abstract Classes
- 8. Real-World Applications
- 9. Common Pitfalls and Errors

Recap: What is Abstraction?

- Abstraction hides complex implementation details, showing only the essential features of an object.
- It focuses on 'what' an object does, rather than 'how' it does it.
- In Python, abstraction is achieved using Abstract Classes and Abstract Methods.
- Unlike statically typed languages (e.g., Java), Python does not have built-in support for abstract classes natively—it uses the 'abc' module.

The 'abc' Module in Python

- Python provides the Abstract Base Classes (abc) module to define abstract classes.
- An abstract class is a class that contains one or more abstract methods.
- Key components of the abc module:
- - 'ABC': The helper base class for defining abstract classes.
- - '@abstractmethod': A decorator indicating abstract methods.
- Instantiating an abstract class directly will raise a TypeError.

Defining an Abstract Base Class (ABC)

- “‘python
- from abc import ABC, abstractmethod
-
- class Shape(ABC):
- def __init__(self, name):
- self.name = name
-
- @abstractmethod
- def area(self):
- pass
-
- def display(self):
- print(f’This is a {self.name}’)
- “‘

Abstract Methods Explained

- An abstract method is a method that is declared, but contains no implementation.
- They enforce a contract: derived classes MUST implement these methods.
- If a child class does not implement all abstract methods of the parent, the child class also becomes abstract.
- Abstract methods typically contain just a 'pass' statement or a docstring.

Abstract Properties in Python

- Just like methods, properties can also be made abstract.
- Use the ‘@property‘ decorator combined with ‘@abstractmethod‘.
- This enforces that child classes must implement a specific property (getter, and optionally setter).
- The order of decorators matters: ‘@property‘ must be the outer decorator (on top).

Implementing Abstract Methods

- ‘‘‘python
- class Rectangle(Shape):
- def __init__(self, width, height):
- super().__init__('Rectangle')
- self.width = width
- self.height = height
-
- def area(self): # Must be implemented
- return self.width * self.height
-
- rect = Rectangle(10, 5)
- print(rect.area()) # Output: 50
- rect.display() # Inherited method
- ‘‘‘

Implementing Abstract Properties

- “‘python
- class Animal(ABC):
- @property
- @abstractmethod
- def diet(self): pass
-
- class Lion(Animal):
- @property
- def diet(self):
- return 'Carnivore'
-
- simba = Lion()
- print(simba.diet) # Output: Carnivore
- “‘

Why Use Abstract Classes?

- 1. Blueprint for Subclasses: Defines a strict template for other classes.
- 2. Code Maintainability: Centralizes shared logic (concrete methods) while enforcing specific behaviors.
- 3. API Design: Ideal for designing plugins or large frameworks where users provide custom implementations.
- 4. Polymorphism: Allows functions to accept objects of the abstract base type, confident that they implement specific methods.

Concrete vs Abstract Classes

- **Abstract Class:**
- - Cannot be instantiated directly.
- - Contains at least one abstract method or property.
- - Intended to be subclassed and extended.
-
- **Concrete Class:**
- - Can be safely instantiated.
- - Provides full implementation for all abstract members inherited.
- - Represents actual, working objects in the system.

Real-world Use Case: Payment Gateway

- “‘python
- class PaymentProcessor(ABC):
- @abstractmethod
- def pay(self, amount): pass
-
- class CreditCardPayment(PaymentProcessor):
- def pay(self, amount):
- print(f'Processing credit card payment of \{*amount*}')
-
- class PayPalPayment(PaymentProcessor):
- def pay(self, amount):
- print(f'Redirecting to PayPal for \{*amount*}')
- “‘

Real-world Use Case: Database Connector

- “‘python
- class DBConnector(ABC):
- @abstractmethod
- def connect(self): pass
-
- @abstractmethod
- def execute(self, query): pass
-
- class MySQLConnector(DBConnector):
- def connect(self): print('Connecting to MySQL')
- def execute(self, query): print(f'Executing {query} in MySQL')
- “‘

Common Errors with Abstraction

- **Error 1: Instantiating an Abstract Class**
- ‘shape = Shape('Generic')‘ -¿ TypeError: Can't instantiate abstract class Shape with abstract method area
-
- **Error 2: Missing Implementation in Subclass**
- “‘python
- class Circle(Shape):
- def __init__(self, radius):
- self.radius = radius
- “‘
- ‘circle = Circle(5)‘ -¿ TypeError: Can't instantiate abstract class Circle with abstract method area

Summary & Next Lecture Preview

- **Key Takeaways:**
- - Abstraction is implemented using the ‘abc‘ module in Python.
- - Inherit from ‘ABC‘ and use ‘@abstractmethod‘.
- - Abstract classes provide a blueprint and enforce method implementation.
- - Concrete classes must implement all abstract methods to be instantiated.
-
- **Next Lecture:** Stack and Queues - Overview of Stack Operations.