Lecture 2.12

Abstraction - Abstract Class (Continued)

Lecture Agenda

- Recap: Abstract Classes
- The ‘@abstractmethod‘ decorator
- Abstract Properties
- Handling Incomplete Subclasses
- Abstract Classes in Data Structure Design
- Practical Example: Abstract Stack
- Summary & Q&A

Recap: What is an Abstract Class?

- An abstract class is a blueprint for other classes.
- It allows you to create a set of methods that must be created within any child classes built from the abstract class.
- A class which contains one or more abstract methods is called an abstract class.
- It cannot be instantiated directly.

The ‘abc‘ Module in Python

- Python comes with a built-in module called ‘abc‘.
- ‘abc‘ stands for Abstract Base Classes.
- To define an abstract class, we inherit from ‘ABC‘ (Abstract Base Class).
- We use the ‘@abstractmethod‘ decorator to declare a method as abstract.

Defining Abstract Methods

- from abc import ABC, abstractmethod
-
- class Shape(ABC):
- @abstractmethod
- def area(self):
- pass
-
- @abstractmethod
- def perimeter(self):
- pass

Subclassing an Abstract Class

- class Rectangle(Shape):
- def __init__(self, w, h):
- self.w = w
- self.h = h
-
- def area(self):
- return self.w * self.h
-
- def perimeter(self):
- return 2 * (self.w + self.h)

Incomplete Subclasses

- What happens if a child class does NOT implement all abstract methods?
- Python will raise a ‘TypeError‘ if you try to instantiate it.
- The subclass itself is considered an abstract class.
- This enforces a strict contract between the base class and its subclasses.

Abstract Properties

- Besides methods, you can also define abstract properties.
- Use the ‘@property‘ decorator along with ‘@abstractmethod‘.
- This forces child classes to provide a specific attribute or property.
- Order matters: ‘@property‘ must be the outer decorator.

Example: Abstract Property

- class Employee(ABC):
- @property
- @abstractmethod
- def role(self):
- pass
-
- class Manager(Employee):
- @property
- def role(self):
- return 'Manager'

Abstract Classes in Data Structures

- Why use them for Data Structures?
- 1. Define a clear interface (e.g., ‘push()‘, ‘pop()‘, ‘is_empty()‘).
- 2. Allow multiple underlying implementations (e.g., Stack using List, Stack using Linked List).
- 3. Ensure consistency across different data structures.

UML Diagram: Abstract Interface

- Abstract Base Class: ‘BaseStack‘
- Methods: ‘push(item)‘, ‘pop()‘, ‘top()‘, ‘is_empty()‘
- Inherited by: ‘ArrayStack‘ and ‘LinkedListStack‘

Defining an Abstract Stack

- class BaseStack(ABC):
- @abstractmethod
- def push(self, item): pass
-
- @abstractmethod
- def pop(self): pass
-
- @abstractmethod
- def is_empty(self): pass

Implementing the Abstract Stack

- class SimpleStack(BaseStack):
- def __init__(self):
- self._data = []
-
- def push(self, item):
- self._data.append(item)
-
- def pop(self):
- return self._data.pop() if not self.is_empty() else None
-
- def is_empty(self):
- return len(self._data) == 0

Key Takeaways

- Abstract classes enforce a strict design contract.
- The ‘@abstractmethod‘ decorator prevents instantiation without implementation.
- Abstract properties can enforce state variables.
- Using ABCs in data structures ensures consistent interfaces.

Next Lecture Preview

- Next Topic: Unit 3 - Stack and Queues
- We will formally dive into Stacks and Queues.
- Implementing Stacks using lists in depth.
- Applications of Stacks (Infix, Postfix expressions).