Week 6: Object-Oriented Programming in Python

Authors: Refat Othman and Diaeddin Rimawi

Lecture 1: Abstraction, Interfaces, and Exception Handling

1. Abstraction

Concept:

- Abstraction means hiding the internal implementation and showing only the necessary features of an object. This helps reduce complexity and isolate the impact of changes.
- For example, consider a `CoffeeMachine` class: users should only need to call `brew_coffee()` without knowing how water is heated or coffee is filtered internally.
- Abstraction allows developers to build reusable components that expose only what is essential for the client code, enhancing maintainability and flexibility.
- In Python:
    - Abstract classes cannot be instantiated directly.
    - Any subclass inheriting from an abstract class must implement all its abstract methods.
    - If a subclass does not implement all abstract methods, it remains abstract.
    - This enforces a clear structure in class design and ensures that subclasses provide concrete behaviors.

Python Implementation: Python uses the `abc` module to create abstract classes and methods.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

    @abstractmethod
    def move(self):
        pass

    def describe(self):
        return "Animals are living organisms that move and make sounds."

class Dog(Animal):
    def make_sound(self):
        return "Bark"

    def move(self):
        return "Runs on four legs"

class Cat(Animal):
```

```
    def make_sound(self):
        return "Meow"

    def move(self):
        return "Jumps gracefully"

# Usage
animals = [Dog(), Cat()]
for animal in animals:
    print(f"{animal.__class__.__name__} says {animal.make_sound()} and
{animal.move()}")
    print(animal.describe())
```

This example shows:

- `Animal` is an abstract class with two abstract methods: `make_sound` and `move`.
- It also includes a concrete method `describe` shared by all subclasses.
- `Dog` and `Cat` are concrete subclasses that implement all abstract methods.
- We cannot create an instance of `Animal` directly.
- The example demonstrates how abstraction enables polymorphism and enforces method implementation in derived classes, while allowing shared behavior through non-abstract methods.

Python vs Java: In Python, abstract methods are created using decorators. Unlike Java, Python does not have strict access modifiers (like `public`, `private`) and does not enforce method visibility.

---

2. Interface

Concept:

- An interface defines a contract (a set of methods) that implementing classes must follow.
- Interfaces are used to specify what a class must do, but not how it does it.
- In Java:
    - Interfaces are declared using the `interface` keyword.
    - A class can implement multiple interfaces, allowing for multiple behavior inheritances.
    - Unlike classes, interfaces cannot have state (fields/attributes) and typically only include method signatures. However, in Java, interfaces can declare constants, which are implicitly public, static, and final. These constants can be accessed directly from implementing classes but cannot be modified.
    - A class must implement all methods declared in the interface, or else it must be declared abstract.
    - Interfaces cannot be instantiated directly.
- In Python:
    - Python does not have a built-in `interface` keyword.
    - Abstract base classes (`ABC`) can be used to simulate interfaces.

    - Python supports duck typing: if an object implements the required methods, it is considered to fulfill the interface.
    - Multiple inheritance is allowed, so a class can inherit from multiple ABCs.
    - Just like Java, abstract base classes or pseudo-interfaces cannot be instantiated directly.

- Interfaces help in designing loosely coupled systems and promote the separation of concerns.
- They are especially useful when multiple classes need to share the same set of method definitions but implement them differently.

Python Implementation: Python does not have a built-in `interface` keyword. Instead, it relies on abstract base classes or duck typing.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    PI = 3.14159  # Constant to simulate Java-style interface constant

    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

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

    def area(self):
        return Shape.PI * self.radius ** 2

    def perimeter(self):
        return 2 * Shape.PI * self.radius

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

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

    def perimeter(self):
        return 2 * (self.width + self.height)

class Triangle(Shape):
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c
```

```
    def area(self):
        s = (self.a + self.b + self.c) / 2
        return (s * (s - self.a) * (s - self.b) * (s - self.c)) ** 0.5

    def perimeter(self):
        return self.a + self.b + self.c

# Usage
shapes = [Circle(5), Rectangle(4, 6), Triangle(3, 4, 5)]
for shape in shapes:
    print(f"{shape.__class__.__name__}: Area = {shape.area():.2f}, Perimeter =
{shape.perimeter():.2f}")
```

This extended example shows:

- `Shape` is used as an interface using an abstract base class.
- It declares a constant `PI`, similar to Java interface constants.
- It defines two abstract methods: `area()` and `perimeter()`.
- `Circle`, `Rectangle`, and `Triangle` provide concrete implementations.
- The list `shapes` shows how we can work with different implementations of the same interface using polymorphism.

---

3. Exception Handling

Concept:

- Exception handling allows you to catch and handle runtime errors, preventing program crashes.
- Python uses `try-except` blocks to capture specific or general exceptions.
- The order of `except` clauses matters; Python matches the first applicable exception handler.
- You can raise exceptions manually using the `raise` keyword.
- Custom exceptions can be defined by extending the `Exception` class.

Examples and Explanations:

1. **Catching Specific Exceptions**

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("You must enter a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

- This handles both input conversion errors and division by zero.

2. **Catching Type Errors and NoneType Issues**

```
def print_upper(text):
    try:
        print(text.upper())
    except AttributeError:
        print("Invalid input: expected a string, got NoneType or other.")

print_upper(None)
```

- Shows how Python catches runtime type issues.

3. **Catching File Errors**

```
try:
    with open("nonexistent.txt") as file:
        content = file.read()
except FileNotFoundError:
    print("The file does not exist.")
```

- Demonstrates file handling with exception awareness.

4. **Raising Exceptions Manually**

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")

try:
    set_age(-5)
except ValueError as e:
    print("Error:", e)
```

- The `raise` keyword is used to trigger exceptions based on custom logic.

5. **Custom Exceptions**

```
class InvalidAgeError(Exception):
    pass

def validate_age(age):
    if age < 0 or age > 120:
        raise InvalidAgeError("Age must be between 0 and 120.")

try:
    validate_age(150)
except InvalidAgeError as e:
    print("Caught custom exception:", e)
```

- Shows how to define and use a custom exception class.

6. **General Exception Handling**

```
try:
    risky_code()
except Exception as e:
    print("An unexpected error occurred:", e)
```

- Catches any unhandled exception type. Should be used sparingly.

7. File Handling with Multiple Exceptions and `finally`

```
try:
    file_name = "data.txt"
    f = open(file_name, "r")
    content = f.read()
    if not content:
        raise ValueError("File is empty.")
except FileNotFoundError:
    print("Error: File does not exist.")
except ValueError as ve:
    print("Error:", ve)
except TypeError:
    print("Error: Invalid operation on file.")
except Exception as e:
    print("General error:", e)
finally:
    try:
        f.close()
        print("File closed successfully.")
    except NameError:
        print("File was never opened, so nothing to close.")
```

Explanation:

- `FileNotFoundError` is caught if the file doesn't exist.
- `ValueError` is raised manually if the file is empty.
- `TypeError` is included for completeness (e.g., misuse of file object).
- A generic `Exception` captures any unexpected issues.
- The `finally` block ensures that the file is closed whether an exception occurred or not.

Definition of `finally`:

- The `finally` block is executed no matter what — whether or not an exception is raised or caught.
- It is typically used for cleanup code like closing files, releasing resources, or restoring system states.
- Ensures resources are freed and the system is left in a consistent state, even if an error occurs.

-

8. File Handling with Exceptions and `else`

```
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("That's not a valid integer.")
else:
    print("Good job! You entered:", number)
```

Definition and Usage:

- The `else` clause in a `try` block runs if no exceptions are raised.
- It is useful for code that should only run when everything in the `try` block succeeds.
- Keeps error handling (`except`) separate from successful execution logic.

Python vs Java:

- Python exceptions are dynamically typed and more flexible.
- Java requires you to declare checked exceptions using `throws` and handle them explicitly.
- Python does not distinguish between checked and unchecked exceptions.
- Python allows raising and defining exceptions with minimal syntax.

Lecture 2: Exercises

Abstraction Exercises

1. **Appliance Abstraction**
    - Define an abstract class `Appliance` with the abstract method `turn_on()`.
    - Subclasses: `WashingMachine` and `Oven` implement their own `turn_on()` method.
    - Each subclass can have an attribute `power_rating` (in watts) and `brand`.
    - UML Structure:
        - Appliance (abstract)
            - power_rating: int
            - brand: str
            - +turn_on(): void
        - WashingMachine
            - +turn_on(): void
        - Oven

        - +turn_on(): void

2. **Vehicle System**
    - Define an abstract class `Vehicle` with abstract methods `start_engine()` and `stop_engine()`.
    - Subclasses: `Car` and `Bike` implement both methods.
    - Add attributes: `make`, `model`, `year`.
    - UML Structure:
        - Vehicle (abstract)
            - make: str
            - model: str
            - year: int
            - +start_engine(): void
            - +stop_engine(): void
        - Car
            - +start_engine(): void
            - +stop_engine(): void
        - Bike
            - +start_engine(): void
            - +stop_engine(): void

3. **Employee Hierarchy**
    - Abstract class `Employee` with abstract methods `calculate_salary()` and `get_benefits()`.
    - Subclasses: `FullTimeEmployee`, `PartTimeEmployee` implement methods differently.
    - Attributes include: `name`, `id`, `base_salary`.
    - UML Structure:
        - Employee (abstract)
            - name: str
            - id: int
            - base_salary: float
            - +calculate_salary(): float
            - +get_benefits(): str
        - FullTimeEmployee
            - +calculate_salary(): float
            - +get_benefits(): str
        - PartTimeEmployee

- +calculate_salary(): float
- +get_benefits(): str

Exception Handling Exercises

1. **User Input Division Program**
   - Task: Prompt the user for two numbers and divide them.
   - Handle exceptions:
     - `ValueError`: if the input is not a number.
     - `ZeroDivisionError`: if the second number is zero.
     - `TypeError`: if incorrect data types are used.
     - `KeyboardInterrupt`: if the user cancels input.
   - Suggested Steps:
     - Use `input()` for both numbers.
     - Convert to `int` inside `try`.
     - Perform the division.
     - Use `except` blocks for all listed exceptions.
     - Use `finally` to print a message at the end.
2. **File Reader Function**
   - Task: Read the content of a file from a user-specified path.
   - Handle exceptions:
     - `FileNotFoundError`: if the file doesn't exist.
     - `PermissionError`: if the program can't access the file.
     - `IsADirectoryError`: if a directory is passed instead of a file.
     - `UnicodeDecodeError`: if the file contains non-text data.
   - Suggested Steps:
     - Use `input()` to get a file path.
     - Use `open()` inside a `try` block.
     - Read and print the file content.
     - Use `finally` to close the file or log the result.
3. **Custom Age Validator**
   - Task: Define a custom exception `InvalidAgeError`.
   - Raise it if the user inputs an invalid age (< 0 or > 120).
   - Handle additional exceptions:
     - `ValueError`: if the input is not a number.
     - `TypeError`: if non-integer type is used.

- Suggested Steps:
  - Create class `InvalidAgeError(Exception)`.
  - Define a function `validate_age(age)`.
  - Prompt user input and convert to integer.
  - Raise `InvalidAgeError` in invalid cases.
  - Wrap the call in a try-except block to catch both standard and custom exceptions.
  - Add a final message using `finally`.

Final Project: OOP Python Application

Objective: Create a fully functional Object-Oriented Python application that incorporates abstraction, interfaces, exception handling, and custom class design.

Project Theme: Build a simplified "Library Management System."

Requirements:

- All `LibraryItem` and `User` instances should initially be read from respective data files (e.g., `items.json`, `users.json`).
- When the administrator adds a new item or user, the updated information should be saved back to these files upon exiting the system.

1. **Abstract Classes and Interfaces**:
   - Create an abstract class `LibraryItem` with attributes like `title`, `author`, and methods like `display_info()` and `check_availability()`.
   - Subclasses: `Book`, `Magazine`, `DVD` each with specific attributes and behaviors.
   - Create an interface-like abstract class `Reservable` with method `reserve(user)` and implement it in `Book` and `DVD`.
2. **Class Composition and Relationships**:
   - Create a `User` class with attributes like `user_id`, `name`, `borrowed_items`.
   - Create a `Library` class to manage items and users. It should allow:
     - Adding/removing items
     - Adding/removing users
     - Borrowing and returning items
     - Reserving items using the `Reservable` interface
3. **Exception Handling**:
   - Handle the following scenarios with try-except:
     - Borrowing an unavailable item
     - Reserving an item that is already reserved
     - Invalid user inputs (nonexistent users or items)
     - File errors when saving/loading data (optional bonus)

- Use `finally` to ensure consistent application flow.

4. **Custom Exceptions**:
   - Create and raise custom exceptions like `ItemNotAvailableError`, `UserNotFoundError`, `ItemNotFoundError`.
5. **File Structure**:
   - Organize your project using multiple files:
     - `main.py`: contains the CLI interface or main loop
     - `models/`: directory with `library_item.py`, `book.py`, `user.py`, etc.
     - `exceptions/`: directory with custom exceptions
6. **System Interaction Scenario**:
   - Upon launching `main.py`, the system should:
     - Load existing items from `items.json` and users from `users.json`.
     - Present a CLI menu with options:
       1. View all available items
       2. Search item by title or type
       3. Register a new user
       4. Borrow an item
       5. Reserve an item
       6. Return an item
       7. Exit and Save
   - Full Interaction Example:

Upon launching `main.py`, the CLI displays the following menu:

```
Welcome to the Library System
1. View all available items
2. Search item by title or type
3. Register as a new user
4. Borrow an item
5. Reserve an item
6. Return an item
7. Exit and Save
```

Example session:

- **User selects option 1**: The system displays all `Book`, `Magazine`, and `DVD` items with their availability status.
- **User selects option 2**: They are prompted to input a search keyword. The system returns matching items.

- **User selects option 3**: They input name and email, and are assigned a unique `user_id`. The new user is added to `users.json`.
- **User selects option 4**: They provide their `user_id` and item ID. If the item is available, it is marked as borrowed and added to the user's `borrowed_items` list.
- **User selects option 5**: They provide their `user_id` and item ID. If the item supports reservation (i.e., implements `Reservable`) and is not already reserved, the reservation is recorded.
- **User selects option 6**: They input their `user_id` and item ID. If matched, the item is marked as available again.
- **User selects option 7**: The system saves all changes back to `items.json` and `users.json` using exception-safe file handling, and exits.

Throughout:

- All invalid inputs (e.g., non-existent `user_id`, unavailable items, wrong menu choices) raise appropriate custom or built-in exceptions.
- File errors during loading or saving are caught and reported.
- The use of abstraction, interfaces (`Reservable`), and structured exception handling is integral to system operation.

Submission Instructions:

- Submit your full project as a `.zip` file.
- The archive should include all Python files and a `README.md` explaining how to run your project and the design decisions you made.
- Make sure your code is well-documented and commented.