Object-Oriented Programming in Python - Lecture Material

Authors: Refat Othman and Diaeddin Rimawi

Lecture 1: Introduction to Object-Oriented Programming in Python

1. What is Object-Oriented Programming (OOP)?

- **Definition**: A programming paradigm based on the concept of "objects" which contain data and methods.
- **Benefits**:
  - Modularity: Code is easier to manage.
  - Reusability: Classes can be reused in different programs.
  - Encapsulation: Hides internal state and requires all interaction to be performed through methods.
  - Scalability and Maintenance: Easier to manage larger projects.

2. Class Definition & `main` Function

```
class Dog:
    def bark(self):
        print("Woof!")

if __name__ == "__main__":
    my_dog = Dog()
    my_dog.bark()
```

- **Explanation**: A class defines a blueprint for creating objects—an object is an instance of a class that holds specific data and can perform actions using methods. In the example, `my_dog` is an object (instance) of the `Dog` class, meaning it has access to the behavior defined by the class, such as `bark()`. `__main__` is used to execute the script directly when it is run, allowing object creation and method invocation to happen as part of the program flow.

3. Data Members, Methods, and the `self` Keyword

Object-oriented programming in Python relies on defining classes that include both *data members* (variables) and *methods* (functions). To associate them with specific objects, Python uses the `self` keyword. Below, we break down these components:

3.1 Data Members

- **Definition**: Data members are variables that store the state of an object. They can be:
  - **Instance variables**: Unique to each object.

```
class Dog:
    def __init__(self, name):
        self.name = name  # instance variable
```

In this example, `name` is a data member that holds the state specific to each instance of `Dog`.

3.2 Methods

- **Definition**: Methods are functions defined inside a class to describe behaviors that objects of the class can perform.
- **Instance methods** operate on data that belongs to the object and require the use of `self`.

```
class Dog:
    def bark(self):
        print("Woof!")
```

`bark` is a method that uses the data (if any) associated with the instance that calls it.

3.3 The `self` Keyword

- **Purpose**: `self` is a reference to the current instance of the class. It is used to access variables and methods associated with that specific object.
- **Why it's needed**:
    - It lets the method know which instance's data to use.
    - It allows each object to maintain its own copy of the data.
    - It distinguishes between class data members and local variables inside methods.
- **Usage**: Always include `self` as the first parameter in instance method definitions. When the method is called using an object, Python automatically passes the object as the `self` argument.

Example 1: Human Class with Distinct Objects

```
class Human:
    def __init__(self, name):
        self.name = name  # Instance variable

    def say_hello(self):
        print(f"Hi, I am {self.name}.")

person1 = Human("Alice")
person2 = Human("Bob")
```

```
person1.say_hello()  # Output: Hi, I am Alice.
person2.say_hello()  # Output: Hi, I am Bob.
```

Although both `person1` and `person2` are instances of the same class, they store different values for the `name` attribute. The `self` keyword ensures that each object accesses its own `name`.

Example 2: Car Class Demonstrating Data Members and Methods

```
class Car:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

    def description(self):
        return f"{self.brand} ({self.year})"

    def is_new(self):
        return self.year >= 2022

car1 = Car("Toyota", 2023)
car2 = Car("Ford", 2018)

print(car1.description())  # Toyota (2023)
print(car2.description())  # Ford (2018)

print(car1.is_new())  # True
print(car2.is_new())  # False
```

This example shows how methods (`description`, `is_new`) can work with data members using `self`. Each object (`car1`, `car2`) has its own values, and the methods operate accordingly.

4. Constructors

- Special method: `__init__`
- **Definition**: A constructor is a special method in Python classes, named `__init__`, which is automatically called when a new object is created from a class.
- **When it is called**: At runtime, immediately after the memory is allocated for the new object, the `__init__` method runs to initialize the object's state by assigning values to data members.
- **Purpose**: It sets up the object with initial values or configuration. You can define required (mandatory) parameters or provide default values for optional ones.
    - **Mandatory values**: Required during object instantiation.
    - **Optional values**: Provide default values.

```
class Dog:
    def __init__(self, name, age=1):
        self.name = name
        self.age = age
```

6. Immutable vs Mutable Variables

- **Immutable Variables**: These are variables whose values cannot be changed after they are created. Common immutable types in Python include `int`, `float`, `str`, and `tuple`.
- **Mutable Variables**: These are variables whose contents can be changed in place without changing their identity. Common mutable types include `list`, `dict`, and `set`.

Example:

```
# Immutable examples
int_value = 42
# int_value[0] = 5  # TypeError: 'int' object does not support item assignment

float_value = 3.14
# float_value[0] = 1 # TypeError: 'float' object does not support item assignment

name = "Fido"
print(name[0])    # Output: 'F'
# name[0] = "R"    # TypeError: 'str' object does not support item assignment

tuple_value = (1, 2, 3)
# tuple_value[0] = 100  # TypeError: 'tuple' object does not support item
assignment

# Mutable examples
tricks = ["sit", "roll"]
tricks.append("stay")
print(tricks)     # Output: ['sit', 'roll', 'stay']

dog_info = {"name": "Buddy", "age": 5}
dog_info["breed"] = "Labrador"
print(dog_info)   # Output: {'name': 'Buddy', 'age': 5, 'breed': 'Labrador'}

commands = {"sit", "stay"}
commands.add("roll")
print(commands)   # Output: {'stay', 'sit', 'roll'} (order may vary)
```

This example shows how immutable types like `int`, `float`, `str`, and `tuple` cannot be modified in place—any such attempt results in an error or creates a new object. In contrast, mutable types like `list`, `dict`, and `set` can be modified using their respective methods.

8. Method Overloading (Simulated)

- **Definition**: Method overloading refers to defining multiple methods with the same name but different signatures (number or types of parameters) to perform different tasks.
- **Python Limitation**: Unlike some other object-oriented languages like Java or C++, Python does not support method overloading based solely on different types or the same number of arguments. If you define multiple methods with the same name, the last definition will overwrite the previous ones.
- **How to simulate overloading in Python**:
    - Use **default parameter values** to handle variable input cases.
    - Use `*args` to accept a variable number of arguments.

Example 1: Using default arguments

```
class Calculator:
    def add(self, a, b=0):
        return a + b

calc = Calculator()
print(calc.add(5))      # Output: 5
print(calc.add(5, 10))  # Output: 15
```

Example 2: Using `*args`

```
class Calculator:
    def add(self, *args):
        result = 0
        for number in args:
            result += number
        return result

calc = Calculator()
print(calc.add(1, 2))            # Output: 3
print(calc.add(1, 2, 3, 4, 5))   # Output: 15
```

These techniques allow flexible method calls, imitating overloading behavior while respecting Python's dynamic typing model.

8.5. Importing Classes from Other Files

In Python, classes are often organized into separate files (modules) to keep code modular and maintainable. One common scenario is having a class defined in one file and importing it into another file that contains the main execution logic.

Example: We have a class `Car` defined in a file named `car.py`, and we want to use this class in a separate file named `main.py`.

File: `car.py`

```
class Car:
    def __init__(self, brand, year):
        self.brand = brand
        self.year = year

    def description(self):
        return f"{self.brand} ({self.year})"
```

File: `main.py`

```
from car import Car

if __name__ == "__main__":
    my_car = Car("Tesla", 2022)
    print(my_car.description())
```

- `from car import Car` tells Python to load the `Car` class from the file `car.py`.
- This approach separates class definitions from program execution logic, improving readability and code reuse.

9. Encapsulation: Public, Private, Protected

- **Encapsulation** is the concept of restricting direct access to some components of an object and only exposing a controlled interface. It promotes data hiding and safeguards the internal state of an object.
- **Public**: Members (variables or methods) that are accessible from anywhere in the program. In Python, any attribute without a leading underscore is considered public.
- **Protected**: Members prefixed with a single underscore `_`. By convention, they are intended for internal use only and should not be accessed directly outside the class or its subclasses. However, Python does not enforce this restriction.
- **Private**: Members prefixed with a double underscore `__`. These are name-mangled by Python to prevent direct access from outside the class. This provides a stronger level of hiding than protected members.

```
class Account:
    def __init__(self):
        self.balance = 0          # Public
        self._limit = 1000        # Protected
        self.__pin = "1234"       # Private
```

To access private variables in a controlled way, use **getter** and **setter** methods:

```python
class Account:
    def __init__(self):
        self.__pin = "1234"

    def get_pin(self):
        return self.__pin

    def set_pin(self, new_pin):
        if isinstance(new_pin, str) and len(new_pin) == 4:
            self.__pin = new_pin
        else:
            print("Invalid PIN format.")
```

Lecture 2: Advanced Concepts in OOP

1. Static Variables and Static Methods

- **Static Variables**: These are class-level variables that are shared across all instances of the class. They are defined outside any instance method using the class name or directly inside the class body.
- **Static Methods**: These are methods that do not operate on instance-specific data. They do not receive the `self` parameter and are defined using the `@staticmethod` decorator. They are often used for utility functions that relate to the class but do not need to access instance or class-level data.
- **Method Decorator**: In Python, a decorator is a special symbol (usually starting with `@`) placed above a method to modify its behavior. For example, `@staticmethod` is used to define a static method. Decorators can be used to define class methods, static methods, or to wrap additional functionality around functions.

```python
class Dog:
    species = "Canis familiaris"  # Static variable shared by all Dog instances

    @staticmethod
    def bark():  # Static method does not use 'self'
        print("Static Woof!")

# Usage
Dog.bark()              # Output: Static Woof!
d1 = Dog()
d2 = Dog()
print(d1.species)       # Output: Canis familiaris
print(d2.species)       # Output: Canis familiaris
```

In the example above, the static method `bark` can be called on the class without creating an instance. The `species` variable is shared among all instances of the class.

3. Exercises: UML Practice

Below are three small design problems to be solved using UML sketches that reflect the OOP principles learned in Lectures 1 and 2.

Exercise 1: Student and Course UML

- Design a `Student` class with attributes: name, id
- Design a `Course` class with: title, enrolled_students

UML Sketch:

```
+--------------+      +--------------+
|   Student    |      |    Course    |
+--------------+      +--------------+
| - name       |      | - title      |
| - id         |      | - students[] |
+--------------+      +--------------+
| +get_info()  |      | +enroll()    |
+--------------+      +--------------+
```

Exercise 2: Bank Account UML

- Design a class `Account` with private balance, and methods for deposit, withdraw, and check_balance.

UML Sketch:

```
+----------------+
|    Account     |
+----------------+
| - __balance    |
+----------------+
| +deposit()     |
| +withdraw()    |
| +get_balance() |
+----------------+
```

Exercise 3: Library and Book UML

- Design a class `Book` with title, author
- Design a class `Library` with a collection of books and methods to add and list them

UML Sketch:

```
+-----------+        +-------------+
|   Book    |        |  Library    |
+-----------+        +-------------+
| - title   |        | - books[]   |
| - author  |        +-------------+
+-----------+        | +add_book() |
```

```
| +info()   |        | +list_all() |
+-----------+        +-------------+
```

Assignment: Design and Implement a Restaurant Management System

You are required to design an object-oriented system for managing a restaurant. Your system should include at least the following classes:

- `MenuItem`: representing an individual menu item with a name, price, and category.
- `Order`: representing a customer's order with a list of items and a method to calculate the total.
- `Restaurant`: representing the restaurant with a menu and a record of placed orders.

UML Sketch:

Task: Based on the UML, implement the classes in Python using the principles of class definition, constructors, data members, instance methods, encapsulation, and optionally static methods.

Additionally, create a separate `main` class or script that will serve as the entry point of your program. This script should:

1. **Instantiate** a `Restaurant` object.
2. **Interact with the user** through the command line to perform the following:
    - Add new menu items (prompt the user for name, price, and category).
    - Display the current menu.
    - Create a new order (allow the user to select items by name).
    - Place the order into the restaurant's order list.
    - List all orders with total amounts.

Structure your code cleanly with comments explaining the logic and how it connects back to the concepts of object-oriented programming.

Example Run:

```
Welcome to the Restaurant Management System!
Choose an option:
1. Add menu item
2. View menu
3. Create new order
4. List all orders
5. Exit

> 1
Enter item name: Margherita Pizza
Enter item price: 8.5
Enter item category: Pizza
Menu item added successfully.

Welcome to the Restaurant Management System!
Choose an option:
1. Add menu item
2. View menu
3. Create new order
4. List all orders
5. Exit

> 1
Enter item name: Caesar Salad
Enter item price: 6.0
Enter item category: Salad
Menu item added successfully.

Welcome to the Restaurant Management System!
Choose an option:
1. Add menu item
2. View menu
3. Create new order
4. List all orders
5. Exit

> 2
Menu:
1. Margherita Pizza ($8.5) [Pizza]
2. Caesar Salad ($6.0) [Salad]

Welcome to the Restaurant Management System!
Choose an option:
1. Add menu item
2. View menu
3. Create new order
4. List all orders
5. Exit

> 3
Enter item numbers for the order separated by commas (e.g., 1,2): 1,2
Order created and added successfully.
```

```
Welcome to the Restaurant Management System!
Choose an option:
1. Add menu item
2. View menu
3. Create new order
4. List all orders
5. Exit

> 4
Orders:
Order 1:
- Margherita Pizza ($8.5)
- Caesar Salad ($6.0)
Total: $14.5

Welcome to the Restaurant Management System!
Choose an option:
1. Add menu item
2. View menu
3. Create new order
4. List all orders
5. Exit

> 5
Thank you for using the Restaurant Management System!
```