Intermediate Python Programming – Lesson 3

Facilitated by Kent State University

Topic: Object-Oriented Programming (OOP) Basics
Duration: 1 Hour

Learning Objectives

By the end of this lesson, participants will be able to:

- Define and instantiate Python classes and objects.
- Use instance attributes and methods effectively.
- Differentiate between public, private, and protected attributes.
- Understand encapsulation and its benefits.

Lesson 3: Object-Oriented Programming (OOP) Basics

I. Introduction to Object-Oriented Programming (10 minutes)

Object-Oriented Programming (OOP) is a paradigm that organizes code using objects and classes. Each object is an instance of a class and encapsulates both data and behavior. OOP promotes modularity, reusability, and organization in your codebase.

Key concepts of OOP include:

- **Encapsulation:** Keeping data safe inside objects, accessible only through methods.
- **Abstraction:** Hiding internal details and exposing only essential features.
- **Inheritance:** Defining new classes from existing ones to reuse and extend behavior.
- **Polymorphism:** Allowing different classes to implement the same interface or method differently.

Multiple-Choice Question

What is the primary advantage of Object-Oriented Programming?

A. It makes code harder to debug
B. It allows structuring code using functions only
C. It enables modularity and code reuse
D. It eliminates the need for functions

Answer: C. OOP enables modularity and code reuse.

Short Answer Question

What is the difference between procedural programming and object-oriented programming?

Expected Answer: Procedural programming is function-based, whereas OOP organizes code into objects that encapsulate both data and behavior.

II. Defining Classes and Creating Objects (15 minutes)

In Python, a class is defined using the `class` keyword. Objects are created by calling the class like a function. The special method `__init__()` is called a constructor and runs automatically when a new object is instantiated. It is typically used to initialize attributes.

Example:

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

# Create an instance
my_car = Car("Toyota", "Camry", 2022)
print(my_car.brand, my_car.model, my_car.year)
```

Expected Output:

```
Toyota Camry 2022
```

Exercise 1

Define a `Car` class with attributes `brand`, `model`, and `year`. Create an object for a Toyota Camry (2022).

Multiple-Choice Question

What is the purpose of the `__init__()` method?

A. To delete an object
B. To initialize object attributes when an instance is created
C. To execute the class only once
D. To define a static method

Answer: B. The `__init__()` method initializes object attributes.

III. Instance Attributes and Methods (15 minutes)

Instance attributes belong to a specific object. You define them using `self.attribute` in the `__init__` method or other instance methods. Instance methods operate on these attributes and are defined with `self` as the first parameter.

Example:

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

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

# Test the method
my_car = Car("Toyota", "Camry", 2022)
print(my_car.describe())
```

Expected Output:

```
2022 Toyota Camry
```

Exercise 2

Add a method `describe()` to the Car class that prints the car details.

Multiple-Choice Question

How do you call an instance method inside a class?

A. method_name()
B. self.method_name()
C. class.method_name()
D. object_name.method_name()

Answer: B. `self.method_name()` is used inside the class.

Short Answer Question

What does `self` represent in a class method?

Expected Answer: `self` represents the instance of the class and allows access to instance attributes and methods.

IV. Public, Private, and Protected Attributes (10 minutes)

Python uses naming conventions to indicate the intended visibility of attributes.

- **Public attributes** (no underscore): Accessible from anywhere.
- **Protected attributes** (single underscore `_attribute`): Intended for internal use, but not enforced.

- **Private attributes** (double underscore `__attribute`): Name-mangled by Python to prevent accidental access from outside the class.

Example:

```python
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.__year = year  # Private attribute

    def get_year(self):
        return self.__year

# Test the method
my_car = Car("Toyota", "Camry", 2022)
print(my_car.get_year())  # Correct way to access private attribute
# print(my_car.__year)  # This would cause an AttributeError
```

Expected Output:

```
2022
```

Exercise 3

Modify the Car class to make `year` a private attribute and create a method to retrieve it.

Multiple-Choice Question

How can you access a private attribute in Python?

A. Directly using `object.__attribute`
B. By using a method inside the class
C. Using `super()`
D. By renaming the attribute

Answer: B. Private attributes should be accessed using a method inside the class.

Short Answer Question

What is the difference between a protected (`_attribute`) and private (`__attribute`) attribute?

Expected Answer: Protected attributes are conventionally internal but can still be accessed, while private attributes are name-mangled and cannot be accessed directly.

V. Encapsulation and Its Benefits (5 minutes)

Encapsulation is the principle of restricting direct access to some parts of an object. This protects the internal state of the object and enforces how it is accessed or modified.

Getter and setter methods allow safe, controlled access:

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if amount > 0 and amount <= self.__balance:
            self.__balance -= amount
```

Multiple-Choice Question

What is the main purpose of encapsulation?

A. To speed up program execution
B. To restrict direct access to certain attributes
C. To remove unwanted data from objects
D. To make code more complex

Answer: B. Encapsulation restricts direct access