Python Online Free Ramzan Course 2025
Taught by: Shaida Muhammad

Object-Oriented Programming (OOP)

Day 16 - Python Basics

Shaida Muhammad

ShaidaSherpao@gmail.com

Monday, March 24, 2025

Agenda

1 What is OOP?

2 Classes and objects

3 Attributes and methods

4 The __init__ method (constructor)

5 The self keyword

6 Encapsulation

7 Inheritance

8 Polymorphism

9 Abstraction

10 Hands-on practice

What is OOP?

- **Definition:** A programming paradigm that organizes code into reusable, self-contained structures called **objects**.
- **Why OOP?**
  - **Reusability:** Write once, use many times.
  - **Modularity:** Break down complex problems into smaller, manageable parts.
  - **Maintainability:** Easier to update and debug.
- **Key Concepts:**
  - **Class:** A blueprint for creating objects.
  - **Object:** An instance of a class.
  - **Attributes:** Variables that belong to an object.
  - **Methods:** Functions that belong to an object.

Classes and Objects

- **Class Syntax:**

```
class ClassName:
    # Attributes and methods
```

- **Object Creation:**

```
object_name = ClassName()
```

- **Example:**

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

my_dog = Dog()
my_dog.bark()  # Output: Woof!
```

The `__init__` Method (Constructor)

- **Purpose:** Initializes an object when it's created.
- **Syntax:**

```
def __init__(self, parameters):
    # Initialize attributes
```

- **Example:**

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

student = Student("Ali", 17)
print(student.name)  # Output: Ali
print(student.age)   # Output: 17
```

The `self` Keyword

- **Purpose:** Refers to the instance of the class.
- **Example:**

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

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

student = Student("Ali", 17)
student.display_info()  # Output: Name: Ali, Age: 17
```

- **Key Points:**
  - `self` is used to access attributes and methods within the class.
  - It must be the first parameter of any method in a class.

Encapsulation

- **Definition:** Bundling data (attributes) and methods that operate on the data into a single unit (class).
- **Purpose:** Protects data from unauthorized access and modification.
- **Example:**

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute
    def deposit(self, amount):
        self.__balance += amount
    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds!")
    def get_balance(self):
        return self.__balance
account = BankAccount(1000)
account.deposit(500)
account.withdraw(200)
print(account.get_balance())  # Output: 1300
```

- **Key Points:**
  - Use private attributes (e.g., `__balance`) to restrict direct access.
  - Provide public methods (e.g., `get_balance()`) to interact with private data.

Inheritance

- **Definition:** Creating a new class from an existing class.
- **Purpose:** Promotes code reuse and establishes a relationship between classes.
- **Example:**

```
class Animal:
    def speak(self):
        print("Animal speaks!")

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

my_dog = Dog()
my_dog.speak()  # Output: Animal speaks!
my_dog.bark()   # Output: Woof!
```

- **Key Points:**
  - The new class (child) inherits attributes and methods from the existing class (parent).
  - Use `super()` to call the parent class's methods.

Polymorphism

- **Definition:** Using a single interface to represent different types.
- **Purpose:** Allows objects of different classes to be treated as objects of a common superclass.
- **Example:**

```
class Animal:
    def speak(self):
        print("Animal speaks!")  # Default implementation

class Dog(Animal):
    def speak(self):  # Method overriding (polymorphism)
        print("Woof!")  # Dog-specific behavior

class Cat(Animal):
    def speak(self):  # Method overriding (polymorphism)
        print("Meow!")  # Cat-specific behavior

# Using polymorphism
animals = [Dog(), Cat()]  # List of different Animal subtypes
for animal in animals:
    animal.speak()  # Same method, different behaviors
```

- **Key Points:**
  - Polymorphism enables flexibility and extensibility in code.
  - Achieved through method overriding and interfaces.

Abstraction

- **Definition:** Hiding complex implementation details and enforcing patterns
- **Purpose:** Simplifies interaction with objects by focusing on what they do, not how they do it.
- **Example:**

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius ** 2

circle = Circle(5)
print(circle.area())  # Output: 78.5
```

- **Key Points:**
  - Use abstract classes and methods to define a blueprint for other classes.
  - Abstract classes cannot be instantiated directly.

Hands-On Practice

- **Task 1:** Create a `Car` class with attributes `brand`, `model`, and `year`.

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

    def display_info(self):
        print(f"{self.brand} {self.model} ({self.year})")

my_car = Car("Toyota", "Corolla", 2020)
my_car.display_info() # Output: Toyota Corolla (2020)
```

- **Task 2:** Create a `Student` class with attributes `name` and `age`.

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

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}")

student = Student("Ali", 17)
student.display_info()  # Output: Name: Ali, Age: 17
```

Task 3: Create a `BankAccount` class with methods `deposit()` and `withdraw()`.

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

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
        else:
            print("Insufficient funds!")

account = BankAccount(1000)
account.deposit(500)
account.withdraw(200)
print(account.balance)  # Output: 1300
```

Recap

- OOP organizes code into reusable objects.
- A **class** is a blueprint, and an **object** is an instance of a class.
- Use `__init__` to initialize attributes and `self` to access them.
- **Encapsulation**: Protects data by bundling it with methods.
- **Inheritance**: Promotes code reuse through parent-child relationships.
- **Polymorphism**: Allows objects of different types to be treated uniformly.
- **Abstraction**: Simplifies interaction by hiding implementation details.

Homework

1. Create a `Book` class with attributes `title`, `author`, and `year`. Add a method to display the book's details.
2. Create a `Vehicle` class and derive `Car` and `Bike` classes from it.
3. Create a base class `Shape` with an abstract method `area()`. Derive `Circle` and `Rectangle` classes from it.
4. Create a `Person` class with private attributes `name` and `age`. Use getter and setter methods to access them.

Q&A

- Do you have any questions?
- Share your thoughts.

Closing

Next class: Working with APIs