**Module 11. Classes and Objects**

**Learning Objectives**

* Differentiate between procedural and object-oriented programming paradigms.
* Define and create classes and objects in Python.
* Implement attributes and methods in Python classes.
* Utilize the ***str()*** function for custom object string representation.
* Interpret and create basic UML class diagrams.

1. **Procedural and Object-Oriented Programming**

Procedural programming and object-oriented programming (OOP) represent two distinct paradigms in software development. Procedural programming focuses on a linear sequence of instructions and the use of functions to organize code. Data is often stored in global variables, accessible throughout the program, which can lead to potential side effects. This approach is straightforward and efficient for small, simple tasks but can become complex and difficult to manage as the program grows. In procedural programming, the emphasis is on the actions or procedures that operate on data, following a top-down design approach.

In contrast, OOP organizes code into classes and objects, promoting a modular structure where data and methods are encapsulated within objects. This encapsulation enhances code security and integrity by restricting direct access to some components of the object. OOP supports inheritance and polymorphism, which facilitate code reuse and flexibility. This paradigm is particularly beneficial for large and complex applications, as it allows for better scalability and maintainability. The focus in OOP is on the objects and their interactions, following a bottom-up design approach, making it easier to extend and modify applications over time.

**Benefits of Object-Oriented Programming**

Object-oriented programming (OOP) offers several benefits that make it a powerful paradigm for software development, especially for complex and large-scale applications. Here are some key benefits:

**Modularity**

OOP allows you to break down a complex program into smaller, manageable, and reusable pieces called objects. Each object can represent a specific part of the program, making it easier to understand, develop, and test.

**Reusability**

Through inheritance, classes can be derived from existing classes, allowing you to reuse code and create a hierarchy of classes. This reduces redundancy and promotes code reuse, making development more efficient.

**Scalability**

OOP supports the design of systems that are easily scalable. You can add new classes and objects to an existing system without affecting its overall functionality. This makes it easier to expand and maintain applications as requirements evolve.

**Maintainability**

Encapsulation, one of the core principles of OOP, ensures that an object's internal state is hidden from the outside world. This means that changes to an object's internal implementation do not affect other parts of the program, making it easier to maintain and update.

**Abstraction**

OOP allows you to create abstract classes and interfaces that define a contract for what a class should do, without specifying how it should do it. This abstraction simplifies complex reality by modeling classes appropriate to the problem.

**Flexibility and Extensibility**

Polymorphism enables objects to be treated as instances of their parent class rather than their actual class. This allows for flexibility and the ability to extend the code with new functionality with minimal changes to the existing codebase.

**Improved Collaboration**

OOP makes it easier for multiple developers to work on the same project. Different developers can work on different classes and objects simultaneously without interfering with each other's work, thanks to the modular nature of OOP.

**Real-World Modeling**

OOP provides a clear structure for modeling real-world entities and relationships, making it easier to translate business requirements into technical solutions. This alignment with real-world concepts enhances the clarity and intuitiveness of the code.

OOP provides a robust framework for building complex and scalable applications. Its principles of modularity, reusability, scalability, maintainability, abstraction, flexibility, collaboration, and real-world modeling make it a preferred approach for modern software development.

1. **Understanding Classes and Objects**

A **class** in Python is a blueprint for creating objects. It defines attributes (variables) and methods (functions) that the created objects will have. Think of a class as a blueprint for building a house. It defines the structure and features of the house but isn't an actual house by itself.

An **object** is an instance of a class. When you create an object from a class, you are essentially making a concrete realization of that class. Each object can have its own unique set of attributes and can use the methods defined in the class. An object is a house built from that blueprint. Each house (object) can have its own unique features, like paint color or number of rooms, but all houses built from the same blueprint share the same basic structure.

**Step-by-Step Example**

**Step 1: Define a Class**

First, we define a class called Dog. This is our blueprint for creating dog objects.

class Dog:

    def \_\_init\_\_(self, name, age):

        self.name = name  # Attribute

        self.age = age    # Attribute

    def bark(self):       # Method

        print("Woof!")

In this example:

* **\_\_init\_\_**: This is a special method called a constructor. It initializes the attributes of the dog.
* **self**: A reference to the current instance of the class. It is used to access attributes and methods of the object.
* **name** and **age**: Attributes of the dog.
* **bark**: A method that makes the dog bark.

|  |
| --- |
| **The *\_\_init\_\_()* Function**  The ***\_\_init\_\_()*** function is a special function in Python classes. It's called automatically when you create a new object from a class. This function is used to initialize the object's attributes and to perform any setup operations necessary for the object.  Think of ***\_\_init\_\_()*** as a way to set up your object with initial values and configurations as soon as it's created. |

|  |
| --- |
| **The *self* Parameter**  The ***self*** parameter is a reference to the current instance of the class. It allows you to access the attributes and methods of the class within its methods. While it is traditionally named self, you can call it anything you like. However, it must be the first parameter of any method in the class.  **Why Use *self*?**  Using self helps to differentiate between instance variables (those belonging to the object) and local variables or parameters. It ensures that the attributes and methods are associated with the particular instance of the class. |

**Step 2: Create an Object**

Now, we create an object (an instance of the Dog class) called ***my\_dog***.

my\_dog = Dog("Buddy", 3)

Here, ***my\_dog*** is an instance of the ***Dog*** class with the name "Buddy" and age 3.

**Step 3: Use the Object**

We can access the attributes and methods of ***my\_dog*** using dot notation.

print(my\_dog.name)  # Output: Buddy

print(my\_dog.age)   # Output: 3

my\_dog.bark()       # Output: Woof!

**Putting It All Together**

Here's the complete code for our simple example:

class Dog:

    def \_\_init\_\_(self, name, age):

        self.name = name

        self.age = age

    def bark(self):

        print("Woof!")

# Create an object of the Dog class

my\_dog = Dog("Buddy", 3)

# Access attributes and methods

print(my\_dog.name)  # Output: Buddy

print(my\_dog.age)   # Output: 3

my\_dog.bark()       # Output: Woof!

To summarize:

* **Class**: Defines a blueprint for creating objects (e.g., ***Dog***).
* **Object**: An instance of a class (e.g., ***my\_dog***).
* **Attributes**: Characteristics of the object (e.g., ***name***, ***age***).
* **Methods**: Functions that the object can perform (e.g., ***bark***).

1. **Attributes and Methods**

When we use classes and objects in Python to represent a software process, we can create a program that mimics or handles tasks related to software.

* **Attributes (or variables)**: These are like characteristics of the process. They hold information about the process.
* **Methods (or functions)**: These are like actions the process can perform.

When we create an object of the ***Process*** class, it represents one specific software process in action. This object has its own set of values for each attribute (characteristic) and can perform actions defined by the methods in the class.

Here's an example to make it clear:

**Define a Class**: Think of it as creating a blueprint for a software process.

|  |
| --- |
| class Process:  def \_\_init\_\_(self, name, id):  self.name = name # Attribute: name of the process  self.id = id # Attribute: id of the process  def start(self):  print(f'Process {self.name} with ID {self.id} is starting.')  def stop(self):  print(f'Process {self.name} with ID {self.id} is stopping.') |

**Create an Object**: This is like making a specific process using the blueprint.

|  |
| --- |
| my\_process = Process("Data Analysis", 101) |

**Use the Object**: We can now use this object to perform actions.

|  |
| --- |
| print(my\_process.name) # Output: Data Analysis  print(my\_process.id) # Output: 101  my\_process.start() # Output: Process Data Analysis with ID 101 is starting.  my\_process.stop() # Output: Process Data Analysis with ID 101 is stopping. |

**Putting It All Together**

Here's the complete code for our ***Process*** class example:

class Process:

    def \_\_init\_\_(self, name, id):

        self.name = name  # Attribute: name of the process

        self.id = id      # Attribute: id of the process

    def start(self):

        print(f'Process {self.name} with ID {self.id} is starting.')

    def stop(self):

        print(f'Process {self.name} with ID {self.id} is stopping.')

def main():

    my\_process = Process("Data Analysis", 101)

    print(my\_process.name)  # Output: Data Analysis

    print(my\_process.id)    # Output: 101

    my\_process.start()      # Output: Process Data Analysis with ID 101 is starting.

    my\_process.stop()       # Output: Process Data Analysis with ID 101 is stopping.

if \_\_name\_\_=="\_\_main\_\_":

    main()

1. **The *\_\_str\_\_()* Function**

The ***\_\_str\_\_()*** function in a class is used to define what should be returned when the class object is converted to a string (for example, when using ***print()***). If you don't define the ***\_\_str\_\_()*** function, the default string representation of the object is used, which is not very informative.

**Example Without *\_\_str\_\_()***

Here’s what happens when you don’t define the ***\_\_str\_\_()*** function in the ***Car*** class:

class Car:

    def \_\_init\_\_(self, make, model, year):

        self.make = make

        self.model = model

        self.year = year

my\_car = Car("Toyota", "Corolla", 2020)

print(my\_car)       # Output: <\_\_main\_\_.Car object at 0x7f8e3c29f700>

Output: The default string representation, which is not very useful.

**Example With *\_\_str\_\_()***

Here’s how to define the ***\_\_str\_\_()*** function to provide a more useful string representation for the ***Car*** class:

class Car:

    def \_\_init\_\_(self, make, model, year):

        self.make = make

        self.model = model

        self.year = year

    def \_\_str\_\_(self):

        return f"{self.year} {self.make} {self.model}"

# Creating an object of the Car class

my\_car = Car("Toyota", "Corolla", 2020)

# Printing the object

print(my\_car)  # Output: 2020 Toyota Corolla

Output:

* **Custom String Representation**: The ***\_\_str\_\_()*** function returns a formatted string that includes the ***year***, ***make***, and ***model*** of the car.

1. **The Unified Modeling Language**

Unified Modeling Language (UML) is a standardized visual language used in software engineering to describe, specify, design, and document the architecture of software systems. It provides various diagram types, including class diagrams (to show system structure), use case diagrams (to illustrate user interactions), sequence diagrams (to detail object interactions over time), and activity diagrams (to represent workflows).

UML helps developers and stakeholders visualize system components, their relationships, and behaviors, facilitating better understanding, communication, and collaboration in software development projects. It enhances the clarity of system design and supports the creation of scalable, maintainable software.

When designing a class, creating a UML diagram can be very useful. UML offers a standardized way to visually represent object-oriented systems. As shown in the following figure, a UML class diagram is a box divided into three parts. The top part contains the class name, the middle part lists the class's data attributes, and the bottom part lists the class's methods.

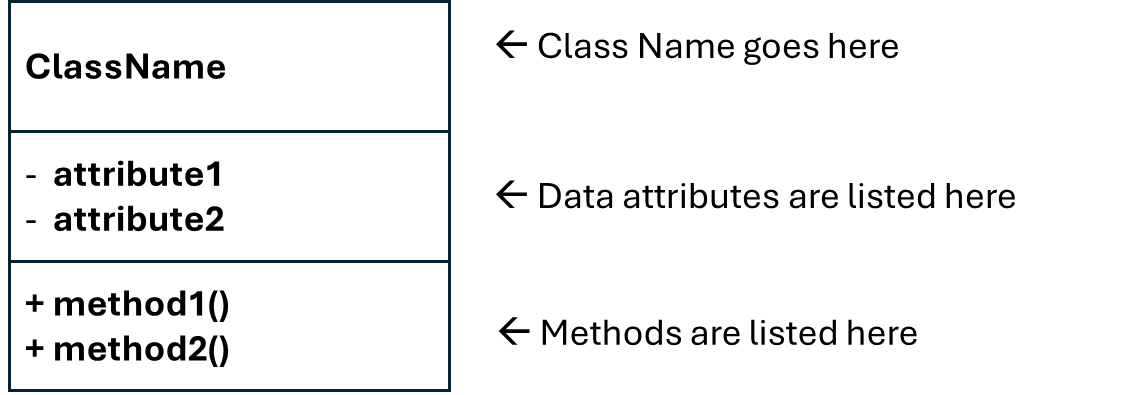


Figure 11-1. General Layout of a Class Diagram

* **Class Name**: The top section is for the name of the class.
* **Attributes**: The middle section lists the class's data attributes.
* **Methods**: The bottom section lists the class's methods.

**Example Class Diagrams**

***Car* Class**

* **Class Name**: Car
* **Attributes**: make, model, year
* **Methods**: get\_details(), start(), stop()

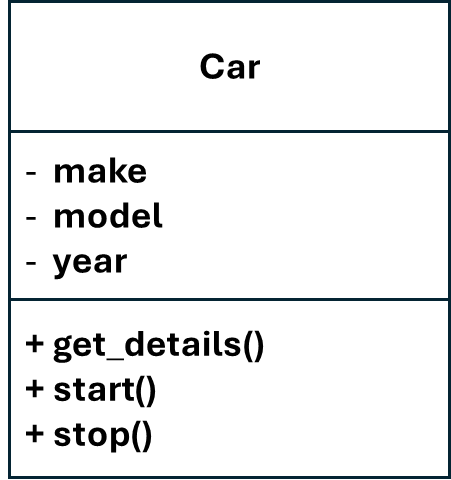


Figure 11-2. Class Diagram for ***Car*** Class

|  |
| --- |
| class Car:  def \_\_init\_\_(self, make, model, year):  self.make = make  self.model = model  self.year = year  def get\_details(self):  return f"{self.year} {self.make} {self.model}"  def start(self):  return f"The {self.make} {self.model} is starting."  def stop(self):  return f"The {self.make} {self.model} is stopping."  # Example usage:  my\_car = Car("Toyota", "Camry", 2022)  print(my\_car.get\_details()) # Output: 2022 Toyota Camry  print(my\_car.start()) # Output: The Toyota Camry is starting.  print(my\_car.stop()) # Output: The Toyota Camry is stopping. |

***Owner*** Class

* **Class Name**: Owner
* **Attributes**: name, address
* **Methods**: owns\_car(), move(), change\_address()

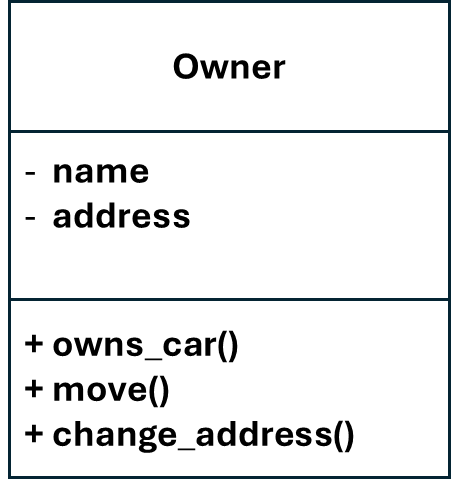


Figure 11-3. Class Diagram for ***Owner*** Class

|  |
| --- |
| class Owner:  def \_\_init\_\_(self, name, address):  self.name = name  self.address = address  def owns\_car(self, car):  return f"{self.name} owns a {car.get\_details()}."  def move(self, new\_address):  self.address = new\_address  return f"{self.name} has moved to {self.address}."  def change\_address(self, new\_address):  self.address = new\_address  return f"{self.name}'s new address is {self.address}."  # Example usage:  owner = Owner("John Doe", "123 Elm Street")  print(owner.owns\_car(my\_car)) # Output: John Doe owns a 2022 Toyota Camry.  print(owner.move("456 Oak Avenue")) # Output: John Doe has moved to 456 Oak Avenue.  print(owner.change\_address("789 Pine Road")) # Output: John Doe's new address is 789 Pine Road. |

To summarize, a UML class diagram visually represents a class by dividing it into three sections: the class name, its data attributes, and its methods. This structure helps in clearly organizing and presenting the class's components, making it easier to understand and design object-oriented systems.

**Summary**

1. Procedural programming focuses on a linear sequence of instructions and the use of functions to organize code.
2. Object-oriented programming (OOP) organizes code into classes and objects, promoting a modular structure.
3. OOP encapsulates data and methods within objects, enhancing code security and integrity.
4. Inheritance and polymorphism in OOP facilitate code reuse and flexibility.
5. OOP is beneficial for large and complex applications due to its scalability and maintainability.
6. Classes in Python serve as blueprints for creating objects, defining attributes and methods.
7. Objects are instances of classes, each with its own unique set of attributes and methods.
8. The ***\_\_init()\_\_*** function in Python classes initializes object attributes and performs setup operations.
9. The ***self*** parameter in Python classes references the current instance of the class.
10. Attributes are characteristics of an object, while methods are actions the object can perform.
11. The ***str()*** function defines the string representation of a class object for more informative output.
12. UML is a standardized visual language for describing, specifying, and documenting software system architecture.
13. UML class diagrams visually represent a class by dividing it into three sections: name, attributes, and methods.
14. The class name is placed in the top section of a UML class diagram.
15. Attributes and methods of a class are listed in the middle and bottom sections of a UML class diagram, respectively.

**Programming Exercises**

1. ***LibraryBook* Class**

Write a class named ***LibraryBook*** which should have the following data attributes:

* ***\_\_title*** (for the title of the book)
* ***\_\_author*** (for the author of the book)
* ***\_\_isbn*** (for the ISBN number of the book)

The class should have an ***\_\_init\_\_*** method to initialize these attributes and the following methods:

* Methods to assign values to each attribute.
* Methods to return the values of each attribute.

Write a program that creates an object of the ***LibraryBook*** class and prompts the user to enter the title, author, and ISBN number. Use the object’s accessor methods to retrieve and display this data.

1. ***Course* Class**

Design a class named ***Course*** that holds the following data: course name, instructor name, number of students, and classroom number. Write appropriate accessor and mutator methods. Also, write a program that creates three instances of the class. One instance should hold your current course information, and the other two should hold information about other courses.

1. ***Car* Class**

Write a class named ***Car*** that holds data about a car in a dealership. The class should store the following data: make, model, year, and price. Once you have written the class, write a program that creates three ***Car*** objects and stores different data in them.

1. ***GroceryItem* Class**

Create a class named ***GroceryItem*** that holds data about an item in a grocery store. The class should store the following data: item name, quantity in stock, and price per unit.

Write a program that creates three ***GroceryItem*** objects and stores different data in them.

1. ***Student* Class**

Design a class that holds the following student data: student ID, name, major, and GPA. Write appropriate accessor and mutator methods. Then, write a program that creates three instances of the class and stores information about three different students.

1. ***Movie* Class**

Write a class named ***Movie*** that holds data about a movie in a rental store. The class should store the following data: title, genre, director, and year released. Once you have written the class, write a program that creates three ***Movie*** objects and stores different data in them.

1. ***Employee* Class**

Create a class named ***Employee*** that stores the following data: employee ID, name, position, and salary. Write a program that creates three ***Employee*** objects and stores data about three different employees. Use accessor methods to retrieve and display the data.

1. ***House* Class**

Design a class that holds the following data about a house: address, square footage, number of bedrooms, and price. Write appropriate accessor and mutator methods. Also, write a program that creates three instances of the class and stores information about three different houses.

1. ***BookStore* Class**

Write a class named ***BookStore*** that holds data about books available in a bookstore. The class should store the following data: book title, author, price, and number of copies available. Write a program that creates three ***BookStore*** objects and stores different data in them.

1. ***MusicAlbum* Class**

Create a class named ***MusicAlbum*** that stores the following data: album name, artist, genre, and release year. Write a program that creates three ***MusicAlbum*** objects and stores data about three different albums. Use accessor methods to retrieve and display the data.