**Module 12. Inheritance**

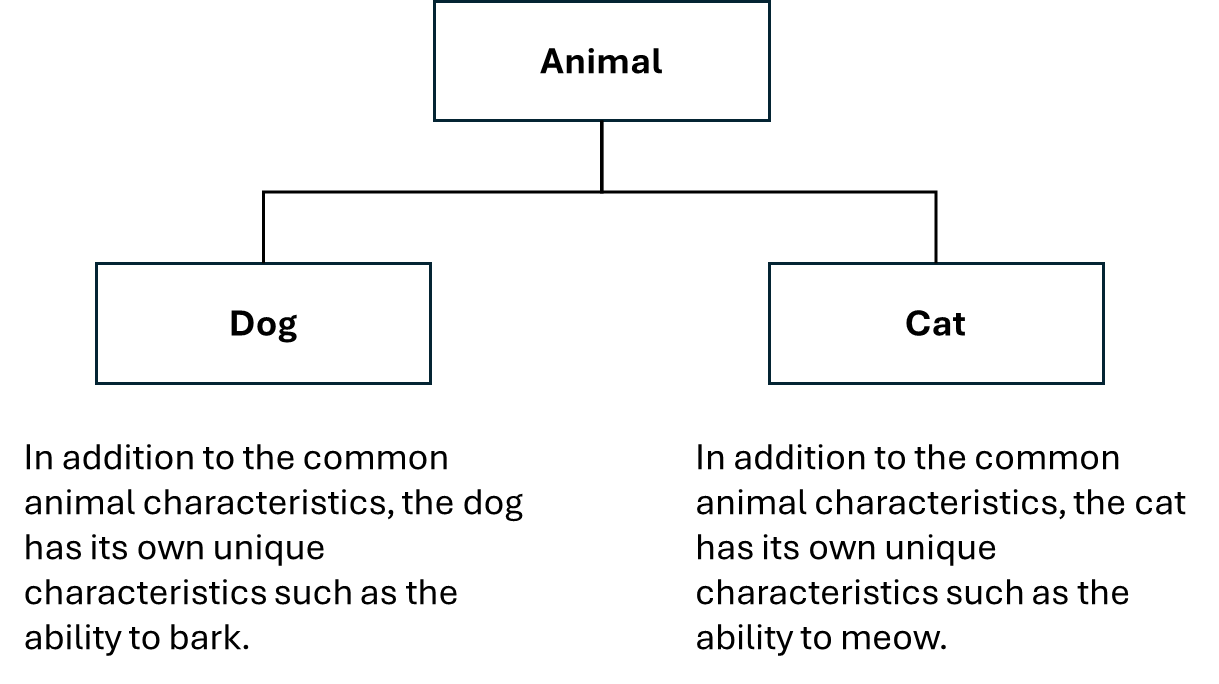
**Learning Objectives**

* Understand the concept and benefits of inheritance in object-oriented programming.
* Implement inheritance in Python to create subclasses from a superclass.
* Differentiate between method overloading and method overriding in Python.
* Use inheritance to represent "is a" relationships in class hierarchies.
* Interpret and utilize UML diagrams to visualize class relationships and inheritance.

1. **Understanding Inheritance in Object-Oriented Programming**

In Python, and many other programming languages, there is a concept called inheritance. This helps us write cleaner and more organized code by allowing new classes to use features from existing ones.

Imagine you have a class called ***Animal*** that has common features like ***eat()*** and ***sleep()***. Now, you want to create classes called ***Dog*** and ***Cat***. Since dogs and cats are an animal, they should have the same basic features, but they might also have some unique features like ***bark()*** for dogs and ***meow()*** for cats. Instead of writing the common features again, you can use inheritance.



Here's how it works:

* **Superclass (or Base or Parent Class)**: This is the class with common features. In our case, ***Animal*** is the base class.
* **Subclass (or Derived or Child Class)**: These classes inherit features from the base class and can have additional features. ***Dog*** and ***Cat*** are the derived classes.

**Benefits of Inheritance**

* **Avoids Code Duplication**: You don't have to write the same code again and again. By inheriting from ***Animal***, the ***Dog*** and ***Cat*** classes automatically get ***eat()*** and ***sleep()***.
* **Makes Code More Intuitive**: It’s easier to understand the relationship between different classes. A dog and a cat are a type of animal, which makes sense in both real life and code.

**How to Implement Inheritance**

Let's look at some Python code to see this in action.

class Animal:

    def eat(self):

        print("This animal is eating")

    def sleep(self):

        print("This animal is sleeping")

class Dog(Animal):  # Dog inherits from Animal

    def bark(self):

        print("This dog is barking")

class Cat(Animal):  # Dog inherits from Animal

    def meow(self):

        print("This cat is meowing")

# Creating an object of the Dog class

my\_dog = Dog()

# Using inherited methods

my\_dog.eat()   # This animal is eating

my\_dog.sleep() # This animal is sleeping

# Using Dog's unique method

my\_dog.bark()  # This dog is barking

# Creating an object of the cat class

my\_cat = Cat()

# Using inherited methods

my\_cat.eat()   # This animal is eating

my\_cat.sleep() # This animal is sleeping

# Using Cat's unique method

my\_cat.meow()  # This cat is meowing

**What’s Happening Here?**

* **class Dog(Animal)**: This line tells Python that ***Dog*** is inheriting from ***Animal***.
* **class Cat(Animal)**: This line tells Python that ***Cat*** is inheriting from ***Animal***.
* **Inherited Methods**: ***my\_dog*** can use ***eat()*** and ***sleep()*** from ***Animal*** without them being defined again in ***Dog***. ***my\_cat*** can use ***eat()*** and ***sleep()*** from ***Animal*** without them being defined again in ***Cat***.
* **Unique Methods**: ***Dog*** has its own method ***bark()***, which is not available in ***Animal***. ***Cat*** has its own method ***meow()***, which is not available in ***Animal***.

1. **Understanding the "is a" Relationship in Inheritance**

In object-oriented programming (OOP), the concept of inheritance helps us represent the "is a" relationship between objects. This means that one object is a specialized version of another object. Let's look at some real-world examples:

* A **car** is a vehicle.
* A **rose** is a flower.
* A **circle** is a shape.
* A **tree** is a plant.

When an "is a" relationship exists, the specialized object (like a rose) has all the characteristics of the general object (like a flower) plus some additional characteristics that make it special.

Example:

Let’s create a ***Car*** superclass and two subclasses: ***RV*** and ***Truck***. The ***Car*** class will have common attributes, while RV and Truck will have additional unique attributes.

Superclass: ***Car***

class Car:

    def \_\_init\_\_(self, make="", model="", year=0):

        self.make = make

        self.model = model

        self.year = year

    def print\_info(self):

        print(f"A {self.year} {self.make} {self.model}. Impressed?")

* **Attributes**: ***make***, ***model***, and ***year*** are common to all cars.
* **Method**: ***print\_info()*** prints out information about the car.

Subclass 1: ***RV***

class RV(Car):  # RV inherits from Car

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

        super().\_\_init\_\_(make, model, year)  # Call the constructor of Car

        self.rv\_type = rv\_type

    def print\_info(self):

        print(f"A {self.year} {self.make} {self.model} RV of type {self.rv\_type}. Impressed?")

* **Inheritance**: By using ***class RV(Car):***, we declare that ***RV*** is a subclass of ***Car***.
* **Calling the Superclass Constructor**: ***super().\_\_init\_\_(make, model, year)*** initializes the common attributes.
* **New Attribute**: ***rv\_type*** is a new attribute unique to ***RV***.
* **Overriding a Method**: ***print\_info()*** is redefined in RV to include ***rv\_type***.

Subclass 2: ***Truck***

class Truck(Car):  # Truck inherits from Car

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

        super().\_\_init\_\_(make, model, year)  # Call the constructor of Car

        self.capacity = capacity

    def print\_info(self):

        print(f"A {self.year} {self.make} {self.model} Truck with a capacity of {self.capacity} tons. Impressed?")

* **Inheritance**: By using ***class Truck(Car):***, we declare that ***Truck*** is a subclass of ***Car***.
* **Calling the Superclass Constructor**: ***super().\_\_init\_\_(make, model, year)*** initializes the common attributes.
* **New Attribute**: ***capacity*** is a new attribute unique to ***Truck***.
* **Overriding a Method**: ***print\_info()*** is redefined in ***Truck*** to include ***capacity***.

**Creating Objects and Using Inherited Methods**

Let’s create objects of ***RV*** and ***Truck*** and see how inheritance works:

* **Creating and Using an RV Object**

my\_rV = RV("Winnebago", "Vista", 2021, "Class A")

my\_rV.print\_info()  # A 2021 Winnebago Vista RV of type Class A. Impressed?

* **Creating and Using a Truck Object**

my\_truck = Truck("Ford", "F-150", 2022, 3)

my\_truck.print\_info()  # A 2022 Ford F-150 Truck with a capacity of 3 tons. Impressed?

**Key Points**

* **Superclass and Subclass**: ***Car*** is the superclass, and ***RV*** and ***Truck*** are subclasses.
* **Inheritance**: Both ***RV*** and ***Truck*** inherit attributes and methods from ***Car***.
* **Constructor Call**: ***super().\_\_init\_\_(make, model, year)*** ensures that the ***Car*** constructor is called.
* **Method Overriding**: The ***print\_info()*** method in both RV and Truck overrides the method in ***Car***.

1. **Method Overloading and Overriding**

**Method Overloading**

Method overloading is a feature in some programming languages where multiple methods can have the same name but differ in the number or type of their parameters. However, Python does not support method overloading in the same way as languages like Java or C++. Instead, Python allows default parameter values and variable-length arguments to achieve similar behavior.

**Example of Method Overloading (Conceptually)**

Although Python does not support method overloading directly, you can achieve similar behavior using default parameters or variable-length arguments.

class MathOperations:

    def add(self, a, b, c=0):  # c is optional

        return a + b + c

# Creating an object

math\_op = MathOperations()

# Calling the method with two arguments

print(math\_op.add(2, 3))  # Output: 5

# Calling the method with three arguments

print(math\_op.add(2, 3, 4))  # Output: 9

In this example, the ***add*** method can be called with either two or three arguments.

**Method Overriding**

Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The subclass method replaces the superclass method.

**Example of Method Overriding**

Let’s consider the previous example with ***Car***, ***RV***, and ***Truck*** classes to demonstrate method overriding.

class Car:

    def \_\_init\_\_(self, make="", model="", year=0):

        self.make = make

        self.model = model

        self.year = year

    def print\_info(self):

        print(f"A {self.year} {self.make} {self.model}. Impressed?")

class RV(Car):

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

        super().\_\_init\_\_(make, model, year)

        self.rv\_type = rv\_type

    def print\_info(self):  # Overriding the print\_info method

        print(f"A {self.year} {self.make} {self.model} RV of type {self.rv\_type}. Impressed?")

class Truck(Car):

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

        super().\_\_init\_\_(make, model, year)

        self.capacity = capacity

    def print\_info(self):  # Overriding the print\_info method

        print(f"A {self.year} {self.make} {self.model} Truck with a capacity of {self.capacity} tons. Impressed?")

# Creating objects

my\_rv = RV("Winnebago", "Vista", 2021, "Class A")

my\_truck = Truck("Ford", "F-150", 2022, 3)

# Calling the overridden methods

my\_rv.print\_info()  # Output: A 2021 Winnebago Vista RV of type Class A. Impressed?

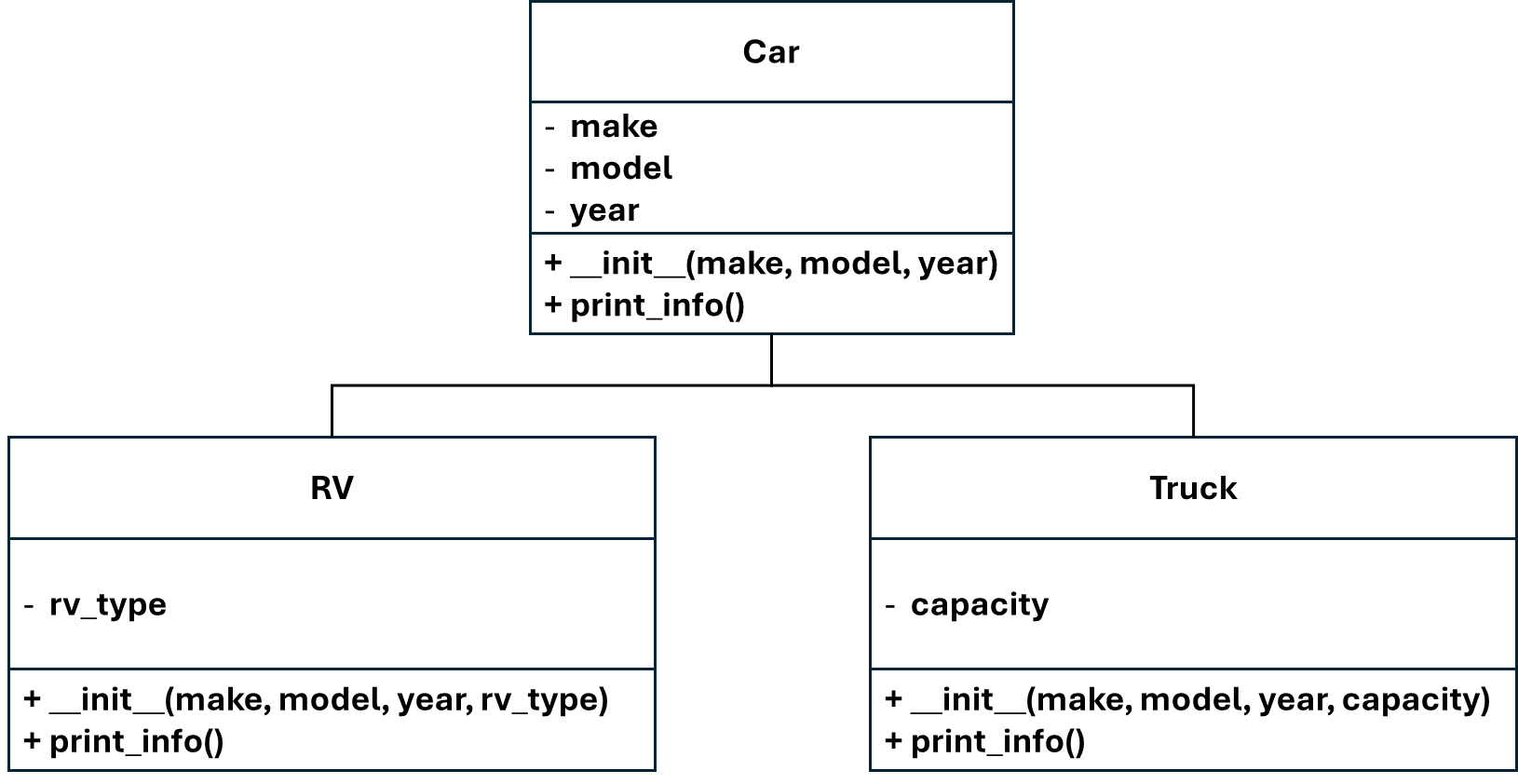
my\_truck.print\_info()  # Output: A 2022 Ford F-150 Truck with a capacity of 3 tons. Impressed?

**Key Differences**

* **Method Overloading**: Allows multiple methods with the same name but different parameters. Python handles this through default arguments and variable-length arguments since it doesn’t support traditional overloading.
* **Method Overriding**: Allows a subclass to provide a specific implementation for a method already defined in its superclass. The subclass method replaces the superclass method when called on an instance of the subclass.

1. **Inheritance in UML Diagrams**

The following figure is a UML diagram showing the relationship between the ***Car***, ***RV***, and ***Truck*** classes.



(Explanation)

* ***Car* Class**:
  + Attributes: ***make***, ***model***, ***year***
  + Methods: ***\_\_init\_\_(make, model, year)***, ***print\_info()***
* ***RV* Class** (inherits from ***Car***):
  + Additional Attribute: ***rv\_type***
  + Methods: ***\_\_init\_\_(make, model, year, rv\_type)***, ***print\_info()***
* ***Truck* Class** (inherits from ***Car***):
  + Additional Attribute: ***capacity***
  + Methods: ***\_\_init\_\_(make, model, year, capacity)***, ***print\_info()***

This class diagram illustrates the inheritance relationship and the attributes and methods of each class.

**Uses of Class Diagrams:**

* **Visualizing System Architecture**:
  + Provide a visual representation of the system’s structure, making it easier to understand and communicate the design.
* **Documentation**:
  + Serve as documentation for the system, helping new developers understand the codebase and the relationships between different classes.
* **Design and Development**:
  + Used in the design phase to plan out the structure of the system before coding begins.
  + Helps in identifying and defining relationships, dependencies, and responsibilities of classes.
* **Refactoring**:
  + Aid in refactoring efforts by providing a clear view of the system’s architecture, making it easier to identify areas that need improvement or optimization.
* **Collaboration**:
  + Facilitate better communication among team members, ensuring that everyone has a shared understanding of the system’s structure.

**Summary**

1. Inheritance allows new classes to use features from existing ones, promoting cleaner and more organized code.
2. The superclass, or base class, contains common features shared by multiple classes.
3. Subclasses, or derived classes, inherit features from the superclass and can have additional unique features.
4. Inheritance avoids code duplication by allowing subclasses to reuse methods from the superclass.
5. The relationship between classes becomes more intuitive with inheritance, reflecting real-life hierarchies.
6. In Python, a class can inherit from another class using the syntax class ***Subclass(Superclass):***.
7. Methods defined in the superclass can be used by instances of the subclass without redefining them.
8. Subclasses can have unique methods that are not present in the superclass.
9. The "is a" relationship in OOP shows that a subclass is a specialized version of the superclass.
10. Constructors of the superclass can be called within the subclass using ***super().\_\_init\_\_()***.
11. Method overriding allows a subclass to provide a specific implementation for a method already defined in its superclass.
12. Method overloading in Python is achieved through default parameter values and variable-length arguments.
13. UML diagrams help visualize the inheritance relationships and the structure of classes.
14. UML class diagrams serve as documentation and aid in understanding the system’s architecture.
15. Class diagrams facilitate better communication among team members, ensuring a shared understanding of the system's structure.

**Programming Exercises**

1. ***Manager* and *DepartmentManager* Classes**

Write a ***Manager*** class that keeps data attributes for the following pieces of information:

* Manager name
* Manager ID

Next, write a class named ***DepartmentManager*** that is a subclass of the ***Manager*** class. The ***DepartmentManager*** class should keep data attributes for the following information:

* Department name
* Number of employees in the department

Write the appropriate accessor and mutator methods for each class. Once you have written the classes, write a program that creates an object of the ***DepartmentManager*** class and prompts the user to enter data for each of the object’s data attributes. Store the data in the object, then use the object’s accessor methods to retrieve it and display it on the screen.

1. ***Vehicle* and *Truck* Classes**

Write a ***Vehicle*** class that keeps data attributes for the following pieces of information:

* Vehicle make
* Vehicle model

Next, write a class named ***Truck*** that is a subclass of the ***Vehicle*** class. The ***Truck*** class should keep data attributes for the following information:

* Cargo capacity (in tons)
* Number of axles

Write the appropriate accessor and mutator methods for each class. Once you have written the classes, write a program that creates an object of the ***Truck*** class and prompts the user to enter data for each of the object’s data attributes. Store the data in the object, then use the object’s accessor methods to retrieve it and display it on the screen.