Introduction to Programming and Problem Solving

Week 11: Object-Oriented Programming

Lecturer: Lemi Agrey Oliver

Department of Information Technology

Kumi University

Content

- Introduction Object Oriented Programming
- Defining Classes,
- The Self-parameter and Adding Methods to a Class,
- Display Class Attributes and Methods,
- Special Class Attributes,
- Accessibility,
- The __init__ Method (Constructor),
- Passing an Object as Parameter to a Method __del__() (Destructor Method),
- Class Membership Tests,
- Method Overloading in Python, Operator Overloading, Inheritance,etc

Introduction to Object Oriented Programming

- Object-Oriented Programming (OOP) is a programming paradigm that is widely used in software development to model real-world entities and their interactions.
- Python fully supports OOP principles and treats everything as an object, permitting easy creation and manipulation of objects.

Key OOP Concepts

- **Classes and Objects**: A class is a blueprint or template for creating objects. It defines the structure and behavior of the objects it creates while An object is an instance of a class.
- **Abstraction**: Abstraction is the process of hiding complex implementation details and showing only the necessary features of an object.

Key OOP Concepts

- **Attributes and Methods**: **Attributes** are variables that store data about the object's state. They are defined within a class and are specific to the objects created from that class while **Methods** are functions defined within a class that define the object's behavior.
- **Encapsulation**: Encapsulation is the concept of bundling data (attributes) and the methods that operate on that data into a single unit (the class).
- **Inheritance**: Inheritance is a mechanism that allows one class (the child or derived class) to inherit the properties and methods of another class (the parent or base class). It promotes code reuse.
- **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common base class. This simplifies code and makes it more flexible

Defining a class

- A class is created using the **class** keyword. Classes are the blueprints for creating objects, and they encapsulate attributes (data) and methods (functions) that operate on that data

- Example 1

```
1.   class Dog:
2.       species = "Canis familiaris"
3.       def __init__(self, name, age):
4.           self.name = name
5.           self.age = age
6.       def description(self):
7.           return f"{self.name} is {self.age} years old."
8.       def speak(self, sound):
9.           return f"{self.name} says {sound}"
10.  mikey = Dog("Mikey", 6)
11.  print(mikey.description()) 00
12.  # Output: "Mikey is 6 years old."
13.  print(mikey.speak("Woof!"))
14.  # Output: "Mikey says Woof!"
```

Let's break down the components of a class definition:

- **Class Name**: The class name follows the **class** keyword and should be a valid identifier. By convention, class names are typically written in CamelCase, where the first letter of each word is capitalized (e.g., **MyClass**, **Person**, **Car**).
- **Class Attributes** (optional): These are variables that belong to the class and are shared among all instances (objects) created from the class. Class attributes are defined within the class block but outside of any methods.
- **Constructor (__init__ method)**: The **__init__** method is a special method (constructor) that gets called when an object is created from the class. It initializes the instance-specific attributes. You can pass parameters to the constructor to set the initial state of the object.
- **Instance Attributes**: These are specific to each object created from the class. They are created and initialized within the **__init__** method using the **self** keyword.
- **Instance Methods**: These are functions defined within the class that can operate on the instance attributes or perform other actions related to the class. Instance methods take **self** as their first parameter, which refers to the instance on which the method is called.

Example 2

```
1.  class Person:
2.  # Class attribute
3.      species = "Homo sapiens"
4.  # Constructor
5.      def __init__(self, name, age):
6.  # Instance attributes
7.          self.name = name
8.          self.age = age
9.  # Instance method
10.     def greet(self):
11.         return f"Hello, my name is {self.name} and I am {self.age} years old. And I am a {self.species}"
12. # Creating an object (instance) of the Person class
13. alice = Person("Alice", 30)
14. # Accessing instance attributes and calling an instance method
15. print(alice.name)
16. # Output: "Alice"
17. print(alice.greet())
18. # Output: "Hello, my name is Alice and I am 30 years old. And I am a Homo sapiens"
```

Explanation

- In this example, the **Person** class defines class attributes (**species**), a constructor (**__init__**), instance attributes (**name** and **age**), and an instance method (**greet**). You can create multiple **Person** objects with different names and ages, each with its own set of instance attributes.
- **The Self-parameter and Adding Methods to a Class**
- The **self** parameter is a reference to the instance of the class, and it is the first parameter in every instance method.
- By convention, you should name this parameter **self**, but it's not a keyword; you can technically choose any name for it, although using **self** is a widely-accepted convention.
- The purpose of **self** is to access and manipulate instance attributes and call other instance methods within the class.

Example three

```
class MyClass:
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
    def instance_method(self, arg1, arg2):
        result = self.attribute1 + self.attribute2 + arg1 + arg2
        return result
    def another_method(self):
        result = self.instance_method(10, 20)
        return result
my_object = MyClass(5, 7)
result1 = my_object.instance_method(2, 3)
print(result1) # Output: 17
result2 = my_object.another_method()
print(result2) # Output: 42
```

In the above example:

1. The **__init__** method is a constructor that initializes the instance attributes **attribute1** and **attribute2** when an object is created.
2. The **instance_method** takes the **self** parameter as its first argument and can access instance attributes (**self.attribute1** and **self.attribute2**) along with the arguments **arg1** and **arg2**.
3. The **another_method** calls **instance_method** by using **self.instance_method(10, 20)** to perform a computation.
4. When we create an object (**my_object**) from the class **MyClass**, we can call the instance methods on that object, which will automatically pass the object itself as the **self** parameter.
5. The **self** allows you to work with object-specific data and behavior. It distinguishes instance attributes and methods from class attributes and methods, making it possible to have multiple instances of the same class with different data.

Accessing Class Attributes and Methods

1. **Accessing Class Attributes**:

- Class attributes are accessed using the class name. You don't need to create an instance of the class to access them.

```
class MyClass:
    class_attribute = "I am a class attribute"
    # Accessing a class attribute p
print(MyClass.class_attribute)
# Output: "I am a class attribute"
```

Accessing Class Methods:

Class methods are accessed using the class name as well. You can call them without creating an instance.

```
class MyClass:
    @classmethod
    def class_method(cls):
        return "I am a class method"
# Accessing and calling a class method
result = MyClass.class_method()
print(result) # Output: "I am a class method
```

Accessing Instance Methods

➢ Instance methods are accessed through an instance of the class, and the **self** parameter is automatically passed to them

```
1.  class MyClass:
2.      def instance_method(self):
3.          return "I am an instance method"
4.  # Create an instance of MyClass
5.  my_instance = MyClass()
6.  # Accessing and calling an instance method
7.  result = my_instance.instance_method()
8.  print(result)
9.  # Output: "I am an instance method"
```

- **Displaying Information about a Class**

Special Class Attributes

- There are several special class attributes and methods in Python that have a specific meaning and purpose within classes. These special attributes and methods are denoted by double underscores at the beginning and end of their names.
- They provide hooks for customizing the behavior of your classes.
- Here are some of the commonly used special class attributes and methods:
- **__init__**: The constructor method is used to initialize instance attributes when an object is created from the class. It's automatically called when you create an object.
- **__str__**: This special method allows you to define a human-readable string representation of an object. It's called when you use the **str()** function or when you print an object.

```
1. def __str__(self):
2.     return f"MyClass instance with attribute1={self.attribute1} and
   attribute2={self.attribute2}"
```

Special Class Attributes

- **__repr__**: This method is used to define an unambiguous string representation of an object. It's called when you use the **repr()** function or when you inspect an object in an interactive environment

```
def __repr__(self):
    return f"MyClass({self.attribute1}, {self.attribute2})"
```

- ****__eq__** and **__ne__**: These methods are used to define the equality and inequality comparisons for objects. They are called when you use the == and **!=** operators, respectively.

    def __eq__(self, other):

- ****__lt__**, **__le__**, **__gt__**, **__ge__**: These methods are used to define comparison methods for objects. They are called when you use comparison operators like <, <=, >, and >=.

```
1.  def __lt__(self, other):
2.      return self.attribute1 < other.attribute1
3.  def __le__(self, other):
4.      return self.attribute1 <= other.attribute1
5.  def __gt__(self, other):
6.      return self.attribute1 > other.attribute1
7.  def __ge__(self, other):
8.      return self.attribute1 >= other.attribute1
```

Accessibility

- In Python, attributes and methods of a class can have different levels of accessibility, which determine where they can be accessed from. The three main levels of accessibility in Python are:
- **Public**: Attributes and methods with public accessibility are accessible from anywhere. They can be accessed from outside the class, as well as from within the class and its subclasses.

```
1. class MyClass:
2.     public_attribute = "I am a public attribute"
3.     def public_method(self):
4.         return "I am a public method"
5. obj = MyClass()
6. # Accessing a public attribute
7. print(obj.public_attribute)
8. # Calling a public method
9. print(obj.public_method())
```

Accessibility

- **Protected**: In Python, there is no strict enforcement of protected attributes or methods. However, a common convention is to prefix attribute and method names with a single underscore to indicate that they are intended for internal use within the class and its subclasses, but they can still be accessed from outside

```
1. class MyClass:
2.     _protected_attribute = "I am a protected attribute"
3.     def _protected_method(self):
4.         return "I am a protected method"
5. obj = MyClass()
6. # Accessing a protected attribute (conventional)
7. print(obj._protected_attribute)
8. # Calling a protected method (conventional)
9. print(obj._protected_method())
```

Accessibility

- **Private**: Attributes and methods with private accessibility are indicated by prefixing their names with a double underscore. This enforces a stronger form of name mangling, making it harder to accidentally access them from outside the class. However, they are still technically accessible.

```
1. class MyClass:
2.     __private_attribute = "I am a private attribute"
3.     def __private_method(self):
4.         return "I am a private method"
5. obj = MyClass()
6. # Accessing a private attribute (technically)
7. print(obj._MyClass__private_attribute)
8. # Calling a private method (technically)
9. print(obj._MyClass__private_method())
```

The __init__ Method (Constructor)

- The **__init__** method, often referred to as the constructor, is a special method in Python that is automatically called when an object is created from a class. It is used to initialize and set the initial state of object attributes.
- This method allows you to perform any necessary setup when an instance of the class is created.
- The **self** parameter, which is the first parameter of **__init__**, refers to the instance being created and is used to access and assign instance-specific attributes.
- Example:

```
1.  class MyClass:
2.      def __init__(self, attribute1, attribute2):
3.  # Initialize instance attributes
4.          self.attribute1 = attribute1
5.          self.attribute2 = attribute2
6.  # Create an object (instance) of MyClass
7.  obj = MyClass("value1", "value2")
8.  # Accessing instance attributes
9.  print(obj.attribute1)
10. # Output: "value1"
11. print(obj.attribute2)
12. # Output: "value2"
```

Explanation of the init constructor example

- The **__init__** method is defined within the **MyClass** class, and it takes three parameters: **self**, **attribute1**, and **attribute2**. The **self** parameter is a reference to the newly created object, while **attribute1** and **attribute2** are the values that you can provide when creating an instance.
- Inside the **__init__** method, instance attributes (**self.attribute1** and **self.attribute2**) are initialized with the values passed to the constructor.
- When you create an object (**obj**) from the class using **MyClass("value1", "value2")**, the **__init__** method is automatically called with the **self** parameter referring to the new object. This initializes the **attribute1** and **attribute2** attributes with the values "value1" and "value2," respectively.

Passing an Object as Parameter to a Method

- Objects can be pass as a parameter to a method just like you would pass any other argument. When you pass an object as a parameter, you can access and manipulate its attributes and call its methods within the method. This allows you to work with objects and perform operations on them.

- Example of passing an object as a parameter to a method:

```
1.  class Person:
2.      def __init__(self, name, age):
3.          self.name = name
4.          self.age = age
5.      def introduce(self):
6.          print(f"My name is {self.name} and I am {self.age} years old.")
7.  class School:
8.      def enroll_student(self, student):
9.          print(f"Enrolling {student.name} in the school.")
10.         student.introduce()
11. # Create two Person objects
12. alice = Person("Alice", 25)
13. bob = Person("Bob", 30)
14. # Create a School object
15. school = School()
16. # Pass Person objects to the School's enroll_student method
17. school.enroll_student(alice)
18. school.enroll_student(bob)
```

Explanation for the above example

- We have a **Person** class with an **introduce** method that prints information about a person's name and age.
- We also have a **School** class with an **enroll_student** method. This method takes a **student** parameter, which is expected to be a **Person** object.
- We create two **Person** objects, **alice** and **bob**, and a **School** object, **school**.
- We then call the **school.enroll_student()** method and pass **alice** and **bob** as arguments. Inside the **enroll_student** method, we access the **student** object's attributes (e.g., **name**) and call its **introduce** method.
- When you run the code, it will output:

- Enrolling Alice in the school.
- My name is Alice and I am 25 years old.
- Enrolling Bob in the school.
- My name is Bob and I am 30 years old.

__del__() (Destructor Method)

- The **__del__()** method, often referred to as the destructor method, is a special method that gets called when an object is about to be destroyed or garbage collected.
- It's not commonly used in Python because Python's memory management system handles object cleanup automatically using reference counting and a cyclic garbage collector.
- The **__del__()** method can be defined in a class to perform custom cleanup actions before an object is removed from memory.

```
1.  class MyClass:
2.      def __init__(self, name):
3.          self.name = name
4.      def display(self):
5.          print(f"Hello, my name is {self.name}")
6.      def __del__(self): print(f"{self.name} is being destroyed")
7.  # Create an object
8.  obj1 = MyClass("Alice")
9.  # Call the object's method
10. obj1.display()
11. # Explicitly delete the object (not recommended, shown for demonstration)
12. del obj1
```

Explanation of the destructor method example

- We define a **MyClass** class with an **__init__()** method for object initialization, a **display()** method to display a message, and a **__del__()** method.
- The **__del__()** method is automatically called when the object is deleted, either explicitly using the **del** statement or when the object goes out of scope and is no longer referenced.
- When we create an object **obj1** and call its **display()** method, it displays a message.
- After we explicitly delete **obj1** using **del**, the **__del__()** method is called, and it displays a message indicating that the object is being destroyed.

More on the destructor method

- It's important to note that relying on the **__del__()** method for resource cleanup is not recommended in most cases.
- Python's automatic garbage collection system is generally sufficient for managing object cleanup, and using **__del__()** can lead to unpredictable behavior or memory leaks in some situations.
- In practice, it's more common to use context managers and explicit cleanup methods e.g., **close()** for resource management.
- The **__del__()** method should only be used in specific cases where you need to perform custom cleanup actions, and you fully understand its implications and potential pitfalls.

Class Membership Tests

- To check if an object belongs to a particular class or its subclass, you can use the **isinstance()** function and the **type()** function to check class membership. Here's how to use both methods:

- **Using isinstance()**: The **isinstance()** function checks if an object is an instance of a specified class or a tuple of classes. It returns **True** if the object is an instance of any of the specified classes, and **False** otherwise.

```
class Person:
    pass
class Student(Person):
    pass
alice = Person()
bob = Student()
print(isinstance(alice, Person))
# True
print(isinstance(bob, Student))
# True
print(isinstance(alice, Student))
# False
print(isinstance(bob, Person))
# True (due to inheritance)
```

Class Membership Tests+

- In the above example, **alice** is an instance of the **Person** class, and **bob** is an instance of the **Student** class, which is a subclass of **Person**.

- **Using type()**: The **type()** function returns the type of an object. You can use it to check the class of an object by comparing it to the class itself.

```
1. class Person:
2.     pass
3. class Student(Person):
4.     pass
5. alice = Person()
6. bob = Student()
7. print(type(alice) is Person)
8. # True
9. print(type(bob) is Student)
10.# True
11.print(type(alice) is Student)
12.# False
13.print(type(bob) is Person)
14.# False (due to inheritance)
```

Method Overloading in Python

- Method overloading is a feature that allows a class to have multiple methods with the same name but different parameter lists.
- Unlike in other languages, in Python, the latest defined method with a particular name will overwrite any previously defined methods with the same name, irrespective of the number or type of parameters.
- However, you can achieve similar behavior in Python using default arguments and variable-length arguments.
- Let us take a quick example
- **Using Default Arguments**:
- You can define a single method with default argument values and then check the arguments within the method to implement different behaviors based on the provided arguments.

Method Overloading in Python code example

```
1. class MyClass:
2.     def my_method(self, arg1, arg2=None):
3.         if arg2 is None:
4.             # Method behavior for one argument
5.             print(f"Received one argument: {arg1}")
6.         else:
7.             # Method behavior for two arguments
8.             print(f"Received two arguments: {arg1} and {arg2}")
9. obj = MyClass()
10.obj.my_method(10)
11.obj.my_method(10, 20)
```

Method overloading Using Variable-Length Arguments:

➢ You can use variable-length arguments to create a more flexible method that can accept any number of arguments. You can then process the arguments inside the method.

```
1.  class MyClass:
2.      def my_method(self, *args):
3.          if len(args) == 1:
4.              # Method behavior for one argument
5.              print(f"Received one argument: {args[0]}")
6.          elif len(args) == 2:
7.              # Method behavior for two arguments
8.              print(f"Received two arguments: {args[0]} and {args[1]}")
9.  obj = MyClass()
10. obj.my_method(10)
11. obj.my_method(10, 20)
```

Operator overloading

- Operator overloading in Python allows you to define how operators such as +, -, *, /, ==, !=, and others behave when used with objects of user-defined classes.
- It gives you the ability to customize the behavior of operators for your custom classes.
- To overload an operator in Python, you need to define special methods in your class with double underscores (also known as "magic" or "dunder" methods) that correspond to the operator you want to overload.
- For example, to overload the + operator, you should define the __add__ method in your class.

Operator overloading example

```
1.  class Vector:
2.      def __init__(self, x, y):
3.          self.x = x
4.          self.y = y
5.          # Overload the addition operator (+)
6.          def __add__(self, other):
7.              return Vector(self.x + other.x, self.y + other.y)
8.          # Overload the equality operator (==)
9.          def __eq__(self, other):
10.             return self.x == other.x and self.y == other.y
11.         def __str__(self):
12.             return f"({self.x}, {self.y})"
13. # Create two Vector objects
14. v1 = Vector(1, 2)
15. v2 = Vector(3, 4)
16. # Use the overloaded addition operator
17. result = v1 + v2
18. print(result)
19. # Output: (4, 6) # Use the overloaded equality operator
20. print(v1 == v2)
21. # Output: False
```

Operator overloading

- In this example, the **__add__** method is used to customize the behavior of the **+** operator for instances of the **Vector** class, and the **__eq__** method is used to customize the behavior of the **==** operator. This allows you to perform vector addition and equality checks using the standard operators.
- **Operator overloading** is a powerful feature in Python and can make your custom classes more intuitive and user-friendly by providing natural syntax for your objects.

Inheritance

- Inheritance is a fundamental concept in object-oriented programming that allows you to create a new class (the child or subclass) based on an existing class (the parent or superclass).
- Inheritance promotes code reuse and the creation of hierarchies of related classes, with the child class inheriting attributes and methods from the parent class while allowing for further customization or extension.
- Inheritance in python can be implemented using the **class** statement and specifying the parent class in parentheses after the child class's name.
- Overview of inheritance in Python:
- **Parent Class (Superclass/Base Class)**: is the class from which attributes and methods are inherited.
- It serves as a template or blueprint for creating child classes.
- Parent classes are sometimes referred to as superclasses or base classes.

Inheritance

- **Child Class (Subclass/Derived Class)**:is the class that inherits attributes and methods from the parent class.
- It can also define additional attributes and methods or override the inherited ones.
- Child classes are sometimes referred to as subclasses or derived classes.
- **Syntax for Inheritance in Python:**
- class ParentClass:

    # Parent class attributes and methods

- class ChildClass(ParentClass):

    # Child class attributes and methods

Inheritance Example 1

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        pass
class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"
class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"
    # Creating objects of child classes
dog = Dog("Buddy")
cat = Cat("Whiskers")
# Calling the speak method
print(dog.speak())
# Output: "Buddy says Woof!"
print(cat.speak())
# Output: "Whiskers says Meow!"
```

Explanation of Inheritance Example 1

- In the example that we have just seen, **Animal** is the parent class, and **Dog** and **Cat** are child classes that inherit from **Animal**.
- The child classes override the **speak** method to provide their own implementations.
- When you create objects of the child classes, they can call the **speak** method to exhibit different behaviors.
- Inheritance allows you to create a hierarchy of classes, making it easier to model relationships between objects and reuse code.
- It promotes code organization, maintenance, and extensibility, which are essential principles in object-oriented programming.

Types of Inheritance

- In object-oriented programming, there are several types of inheritance that allow you to create relationships between classes in different ways. The main types of inheritance include:

- **Single Inheritance**: In single inheritance, a class inherits from one and only one parent class. This is the simplest form of inheritance and is widely used in many programming scenarios.

```
1.  class Parent:
2.      pass
3.  class Child(Parent):
4.      pass
```

- **Multiple Inheritance**: Multiple inheritance allows a class to inherit from more than one parent class. This means that a child class can inherit attributes and methods from multiple parent classes.

Types of Inheritance+

```
1. class Parent1:
2.     pass
3. class Parent2:
4.     pass
5. class Child(Parent1, Parent2):
6.     pass
```

- **Multiple inheritance** can lead to the "diamond problem," where there are multiple paths to a single inherited method, potentially causing conflicts.
- Python's Method Resolution Order (MRO) and the **super()** function help manage this issue

Types of Inheritance++

1. **Multilevel Inheritance**: In multilevel inheritance, a class inherits from a class that, in turn, inherits from another class. This creates a chain of inheritance.

```
class Grandparent:
    pass
class Parent(Grandparent):
    pass
class Child(Parent):
    pass
```

2. **Hierarchical Inheritance**: Hierarchical inheritance occurs when multiple classes inherit from a single base class. This creates a hierarchy of classes, each sharing a common parent class.

Examples of Hierarchical Inheritance

```
1.  class Parent:
2.      def display_parent(self):
3.          print("This is my parent's house.")
4.  class First_Born(Parent):
5.      def display_1st(self):
6.          print("Proud first born though lazy at times.")
7.  class Second_Born(Parent):
8.      def display_2nd(self):
9.          print("Seconds are always humble.")
10. class Third_Born(Parent):
11.     def display_3rd(self):
12.         print("Third born are troublesome")
13. obj1 = First_Born()
14. obj2 = Second_Born()
15. obj3 = Third_Born()
16. obj1.display_1st()
17. obj2.display_2nd()
18. obj3.display_3rd()
```

A figure illustrating Hierarchical Inheritance

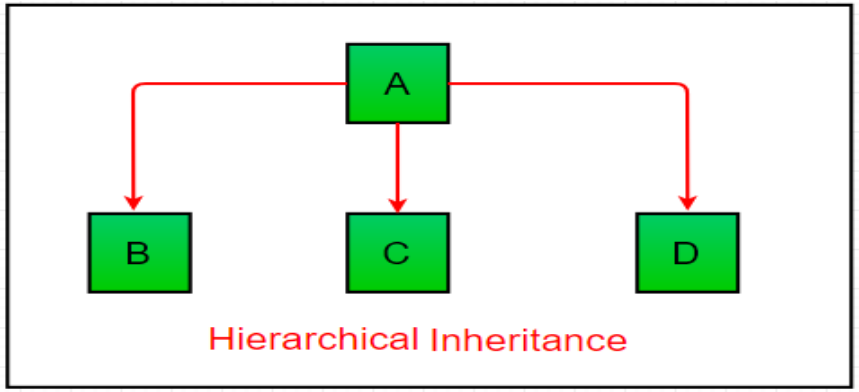

Fig 1: Hierarchical Inheritance[1]

- Hierarchical inheritance is often used to model real-world relationships between objects. For example, a base class called Animal could have common attributes like name, age, and type.
- Derived classes from Animal, such as Dog, Cat, and Bird, would inherit these attributes and could add their own specific ones, such as breed for Dog, furColor for Cat, and wingspan for Bird

Benefits of Hierarchical Inheritance

- Code Reusability: It promotes code reuse by allowing derived classes to inherit common attributes and methods from the base class, reducing redundant code.
- Maintainability: It improves code maintainability by organizing classes in a hierarchical structure, making it easier to understand and modify code.
- Flexibility: It provides flexibility to add new derived classes without affecting the base class, allowing for code extensions and modifications.

Examples of Hierarchical Inheritance

- **Transportation**: A base class Vehicle could have attributes like model, year, and speed. Derived classes from Vehicle, such as Car, Truck, and Motorcycle, would inherit these attributes and add their own specific ones, such as numberOfDoors for Car, cargoCapacity for Truck, and engineType for Motorcycle.
- **Food:** A base class Food could have attributes like name, calories, and ingredients. Derived classes from Food, such as Fruit, Vegetable, and Dessert, would inherit these attributes and add their own specific ones, such as vitaminC for Fruit, fiberContent for Vegetable, and sugarContent for Dessert.
- **Technology**: A base class ElectronicDevice could have attributes like brand, powerSource, and screenSize. Derived classes from ElectronicDevice, such as Smartphone, Laptop, and Tablet, would inherit these attributes and add their own specific ones, such as cameraResolution for Smartphone, processorType for Laptop, and batteryLife for Tablet.

Types of Inheritance+++

➢ **Hybrid (or Mixed) Inheritance**: Hybrid inheritance is a combination of different types of inheritance within a single program. For example, a class can use both single and multiple inheritance in the same codebase.

```
1.  class Parent1:
2.      pass
3.  class Parent2:
4.      pass
5.  class Child(Parent1, Parent2):
6.      pass
```

Types of Inheritance+++

- **Cyclic Inheritance**: Cyclic inheritance, also known as circular inheritance, occurs when classes form a cycle in their inheritance hierarchy. This is not recommended and can lead to various problems in the code.

```
1. class A(B):
2.     pass
3. class B(A):
4.     pass
```

The Object Class

- In Python, the object class serves as the base class for all other classes. It defines the fundamental methods and attributes that are common to all objects in the Python environment. These methods and attributes provide the essential building blocks for object-oriented programming (OOP) in Python.
- Key Features of the object Class in Python:
- Equality Checking: The __eq__() method determines whether two objects represent the same entity.
- Hash Code Generation: The __hash__() method produces a unique identifier for an object, enabling efficient storage and retrieval in hash-based data structures.
- String Representation: The __str__() method converts an object into a human-readable string representation.

The Object Class+

- Garbage Collection Support: The __del__() method allows the object to perform cleanup tasks before it is reclaimed by the garbage collector.
- Inheritance Hierarchy: The __class__() method returns the class to which the object belongs.
- Type Checking: The isinstance() function checks whether an object belongs to a particular class or its subclasses.
- Class Attribute Access: The __dict__ attribute provides access to an object's class attributes.
- Method Resolution Order (MRO): The __mro__ attribute defines the order in which methods are searched for when calling an object's method.
- Subclassing: The super() function allows subclasses to access and override methods from their parent classes.
- Object Inspection: The type() function returns the type of an object.

Inheritance in Detail

- Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to create new classes (subclasses or child classes) based on existing classes (superclasses or parent classes).
- Inheritance promotes code reuse, modularity, and the creation of class hierarchies.
- In this detailed explanation of inheritance, we will cover its various aspects.

Defining a Parent (Superclass) Class:

- A parent class (superclass or base class) is the class that you want to extend or derive from. It serves as a template or blueprint for the child classes.

Defining Child (Subclass) Classes:

Child classes (subclasses or derived classes) are classes that inherit attributes and methods from a parent class. They can also define their own attributes and methods or override the inherited ones

Example1

```
1.  class Animal:
2.      def __init__(self, name, species):
3.          self.name = name
4.          self.species = species
5.      def speak(self):
6.          pass
7.  class Dog(Animal):
8.      def speak(self):
9.          return f"{self.name} (a {self.species}) says Woof!"
10. class Cat(Animal):
11.     def speak(self):
12.         return f"{self.name} (a {self.species}) says Meow!“
13. #Creating Objects of Child Classes
14. dog = Dog("Buddy", "Dog")
15. cat = Cat("Whiskers", "Cat")
16. print(dog.name) # Output: "Buddy"
17. print(cat.species) # Output: "Cat"
18. print(dog.speak()) # Output: "Buddy (a Dog) says Woof!“
19. print(cat.speak()) # Output: "Whiskers (a Cat) says Meow!"
```

Inheritance in Detail+

- **Explanation of the above Example**
- This Python code defines a basic class hierarchy to represent animals and their sounds. It demonstrates the concept of inheritance, where child classes (Dog and Cat) inherit attributes and methods from a parent class (Animal). Here's an explanation of the code:
- class Animal:
- This is the parent class or base class that defines the common attributes and a placeholder method for all animals.
- The __init__ method is the class constructor. It initializes the name and species attributes for an animal when an object is created from this class.
- The speak method is a placeholder method, and it does nothing (contains the pass statement). It is meant to be overridden by child classes to provide specific behaviors for each animal.

Inheritance in Detail++

- **class Dog(Animal):**
- This is a child class that inherits from the Animal class. It specifies a specific sound for dogs.
- The speak method in the Dog class overrides the speak method inherited from the Animal class. It returns a string that includes the name and species of the dog and the sound it makes ("Woof!").
- **class Cat(Animal):**
- This is another child class that also inherits from the Animal class. It specifies a specific sound for cats.
- The speak method in the Cat class overrides the speak method inherited from the Animal class. It returns a string that includes the name and species of the cat and the sound it makes ("Meow!").

Inheritance in Detail+++

- **Creating Objects of Child Classes:**
- Two objects are created, one of the Dog class (named dog) and one of the Cat class (named cat), with specific names and species.
- These objects inherit the attributes and methods of the Animal class.
- Printing Object Attributes and Calling Methods:
- The code demonstrates how to access object attributes and call methods on the objects:

Inheritance in Detail+++

- print(dog.name) outputs "Buddy," which is the name of the dog object.
- print(cat.species) outputs "Cat," which is the species of the cat object.
- print(dog.speak()) calls the speak method of the Dog class, and it outputs "Buddy (a Dog) says Woof!"—the specific sound for dogs.
- print(cat.speak()) calls the speak method of the Cat class, and it outputs "Whiskers (a Cat) says Meow!"—the specific sound for cats.

Multilevel Inheritance in Detail

- **Multilevel inheritance** is a type of inheritance in object-oriented programming where a class inherits from another class, which in turn inherits from another class.
- This creates a chain of inheritance with multiple levels of parent-child relationships. Multilevel inheritance can help create a hierarchy of classes, each building on the functionality of the previous class.

Example

- **Defining Parent Classes:**
- Start by defining a base class (parent class) that provides some attributes and methods

```
1. class Animal:
2.     def __init__(self, name, species):
3.         self.name = name
4.         self.species = species
5.         def speak(self): pass
```

Multilevel Inheritance in Detail+

- Create another class (subclass) that inherits from the parent class. This class will have access to the attributes and methods of the parent class and can add its own.

```
1. class Mammal(Animal):
2.     def __init__(self, name, species, age):
3.         super().__init__(name, species)
4.         self.age = age
5.         def is_mammal(self):
6.             return True
```

- Define another class (child class) that inherits from the intermediate class (in this case, **Mammal**). This class can access the attributes and methods of both the parent class (**Animal**) and the intermediate class (**Mammal**) and can add its own attributes and methods.

Multilevel Inheritance in Detail+++

```
1. class Dog(Mammal):
2.     def __init__(self, name, breed, age):
3.         super().__init__(name, "Dog", age)
4.         self.breed = breed
5.         def speak(self):
6.             return "Woof!“
```

➢ Creating Objects and Accessing Attributes and Methods:

- Now, you can create objects of the most derived (child) class and access the attributes and methods from all levels of the inheritance hierarchy.

Multilevel Inheritance in Detail+++

```python
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species
    def speak(self):
        pass
class Mammal(Animal):
    def __init__(self, name, species, age):
        super().__init__(name, species)
        self.age = age
    def is_mammal(self):
        return True
class Dog(Mammal):
    def __init__(self, name, breed, age):
        super().__init__(name, "Dog", age)
        self.breed = breed
    def speak(self):
        return "Woof!"
buddy = Dog("Buddy", "Golden Retriever", 5)
print(buddy.name) # Output: "Buddy"
print(buddy.species) # Output: "Dog"
print(buddy.age) # Output: 5
print(buddy.breed) # Output: "Golden Retriever"
print(buddy.is_mammal()) # Output: True
print(buddy.speak()) # Output: "Woof!"
```

In the above example:

- **Dog** is the most derived class (the child class) in the hierarchy.
- **Mammal** is an intermediate class that inherits from **Animal**.
- **Dog** inherits from **Mammal**.
- The **super()** function is used to call the constructors of the immediate parent classes (**super().__init__()**).

Multiple Inheritance in Detail

- Multiple inheritance is a type of inheritance in object-oriented programming where a class inherits from more than one parent class.
- In Python, a class can inherit attributes and methods from multiple parent classes, allowing you to create complex class hierarchies and mix behaviors from different sources.
- Here's a detailed explanation of multiple inheritance in Python

Defining Parent Classes:

- Start by defining multiple parent classes, each with its own attributes and methods.

```
1. class Animal:
2.     def __init__(self, name):
3.         self.name = name
4.     def speak(self):
5.         pass
6. class Mammal:
7.     def give_birth(self):
8.         return "Live birth"
9. class Bird:
10.    def lay_eggs(self):
11.        return "Lay eggs"
```

Multiple Inheritance in Detail+

- Create a child class that inherits from multiple parent classes.
- The child class will have access to the attributes and methods of all the parent classes.

```
class Platypus(Animal, Mammal, Bird):
    def __init__(self, name):
        super().__init__(name)
    def speak(self):
        return "Quack!"
```

- Creating Objects and Accessing Attributes and Methods
- Now, you can create objects of the child class and access the attributes and methods from all the parent classes

```
perry = Platypus("Perry")
```

Multiple Inheritance in Detail++

```python
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        pass
class Mammal:
    def give_birth(self):
        return "Live birth"
class Bird:
    def lay_eggs(self):
        return "Lay eggs"
class Platypus(Animal, Mammal, Bird):
    def __init__(self, name):
        super().__init__(name)
    def speak(self):
        return "Quack!"
perry = Platypus("Perry")
print(perry.name) # Output: "Perry"
print(perry.speak()) # Output: "Quack!"
print(perry.give_birth()) # Output: "Live birth"
print(perry.lay_eggs()) # Output: "Lay eggs"
```

In the above example:

- **Platypus** is the child class that inherits from multiple parent classes (**Animal**, **Mammal**, and **Bird**).
- The **super()** function is used to call the constructors of the parent classes (**super().__init__()**).

Using Super()

- The **super()** function is used to call methods and constructors of a parent class (superclass) from a child class (subclass).
- It is particularly helpful when you have multiple inheritance or when you want to extend the behavior of a parent class in a child class. **super()** allows you to access and execute methods and constructors from the parent class, helping you avoid duplicating code and promoting code reusability.
- Here's how to use **super()** in various contexts:
- Calling the Parent Class Constructor:
- Use **super().__init__()** to call the constructor of the parent class from the child class.

Using Super()+

- Calling Parent Class Methods:

- Use **super().method_name()** to call a method of the parent class from the child class.

```
1. class Parent:
2.     def greet(self):
3.         return "Hello from the parent class"
4. class Child(Parent):
5.     def greet(self):
6.         parent_greeting = super().greet()
7.         return f"Child says: {parent_greeting}"
8. child = Child()
9. print(child.greet())
10.# Output: "Child says: Hello from the parent class"
```

Multiple Inheritance with **super()**:

- When using multiple inheritance, **super()** helps maintain the correct method resolution order (MRO) and call parent class methods accordingly.

```
1.  class A:
2.      def method(self):
3.          return "A method"
4.  class B(A):
5.      def method(self):
6.          return "B method"
7.  class C(A):
8.      def method(self):
9.          return "C method"
10. class D(B, C):
11.     def method(self):
12.         return super().method()
13. d = D()
14. print(d.method())
15. # Output: "B method" (calls B's method due to MRO)
```

Multiple Inheritance with **super()**:+

- In this example, **super()** in class **D** calls the method of class **B** because it follows the method resolution order based on the class hierarchy.
- **super()** is a powerful tool that helps maintain code integrity and enables the use of inheritance effectively.
- However, it's important to use it carefully and ensure that it aligns with the design and structure of your classes.
- Proper understanding of method resolution order is crucial, especially when working with multiple inheritance.

Method Overriding in details

- Method overriding is a concept in object-oriented programming that allows a subclass to provide a specific implementation for a method that is already defined in its parent class.
- This allows the child class to customize the behavior of the inherited method without modifying the original method in the parent class. Method overriding is achieved by defining a method with the same name in the child class as the one in the parent class.
- Here's how method overriding works in Python:
- Define a parent class with a method that you want to override:

```
1. class Parent:
2.     def speak(self):
3.         return "This is the parent speaking."
```

Method Overriding in details+

- Create a child class that inherits from the parent class and define a method with the same name as the one you want to override:

```
1. class Child(Parent):
2.     def speak(self):
3.         child_message = "This is the child speaking."
4.         parent_message = super().speak()
5.         return f"{child_message} {parent_message}"
```

- When an instance of the child class calls the overridden method, the child class's method will be executed instead of the parent class's method:

Method Overriding in details++

- Method overriding allows child classes to customize their behavior and extend the functionality of their parent classes.
- It is a fundamental concept in polymorphism, where different objects can respond to the same method call in a way that is appropriate for their specific class.

Precaution: Overriding Methods in Multiple Inheritance

- When you are dealing with multiple inheritance in Python and overriding methods in child classes, it's important to take precautions to ensure that your code is clear, and method resolution order (MRO) behaves as expected.

- Here are some precautions and guidelines for overriding methods in multiple inheritance scenarios

1. **Understand Method Resolution Order (MRO):** In multiple inheritance, Python uses a method resolution order to determine which method to call when there are conflicts in method names. Make sure you understand how MRO works and how the **super()** function follows the MRO to call methods from parent classes.

Precaution: Overriding Methods in Multiple Inheritance

Be Explicit and Document Your Intent: When overriding a method, be explicit about which method you want to call using **super()**. This makes your intent clear and helps others understand the code.

Consistent Method Signatures: Ensure that the overridden method in the child class has the same method signature (i.e., the same name and the same parameters) as the method in the parent class. Otherwise, it may not be considered an override.

Use Different Method Names: If possible, use different method names in parent classes to avoid conflicts. If you do have to use the same method name, make sure the methods have similar behavior to avoid unexpected results.

Precaution: Overriding Methods in Multiple Inheritance

Avoid Deep Method Hierarchies: Deep method hierarchies (many levels of inheritance) can lead to complex and difficult-to-maintain code. It's generally a good practice to keep the inheritance hierarchy as shallow as possible.

Write Tests: Write tests for your classes and methods, especially in the case of multiple inheritance. This helps ensure that your methods behave as expected and that the MRO works correctly.

Summary

- Our second-largest lecture has drawn to a close, and it has been a comprehensive exploration of the realm of Object-Oriented Programming (OOP) in Python, a pivotal programming paradigm that perceives everything as an object.
- Throughout this lecture, we delved deep into the foundational concepts of OOP. We started by understanding the essence of classes, the blueprints that encapsulate the attributes and behaviors of objects. Objects, the concrete instances of these classes, were also a focal point of our discussion. We learned how objects store data in the form of attributes and exhibit functionality through methods.

Summary+

- A significant part of our lecture was dedicated to the intricate world of inheritance, a key mechanism in OOP. We scrutinized how classes can inherit attributes and methods from parent classes, allowing for the creation of hierarchical structures and the reuse of code. This concept of building upon existing classes empowers programmers to craft elegant and efficient solutions.
- But our journey didn't stop at inheritance. We ventured further into the OOP landscape, unraveling the subtleties of method overloading, a technique that enables a single method to serve multiple purposes depending on the provided arguments. We also explored operator overloading, which empowers objects to respond to standard operators, making them more versatile and intuitive.

Summary++

- Method overriding was another vital concept we thoroughly examined. It allows child classes to provide their own implementation of a method inherited from a parent class, fostering customization and adaptability.
- As we reflect on the wealth of knowledge we've acquired in this lecture, it becomes evident that Object-Oriented Programming in Python is not just a programming paradigm but a powerful approach that promotes modularity, reusability, and clear, structured design. Our journey through classes, objects, attributes, methods, inheritance, and various forms of method and operator manipulation has equipped us with a comprehensive toolkit to tackle complex programming challenges and create elegant, efficient, and maintainable software solutions.

Reference

[1]Types of inheritance Python. (2022, July 7). GeeksforGeeks. Retrieved November 7, 2023, from https://www.geeksforgeeks.org/types-of-inheritance-python/

[2] Kamthane, A. N., & Kamthane , A. A. (2018). *PROGRAMMING AND PROBLEM SOLVING WITH PYTHON* McGraw Hill Education (India) Private Limited.

[3] (n.d.). *Basics of Object Oriented Programming*. Studytonight.com. Retrieved November 9, 2023, from https://www.studytonight.com/python/oops-in-python