MODULE 7

OBJECT OREIENTED APPROACH 1

MODULE OVERVIEW

In this module introduces the Object-Oriented Approach in Python, covering the fundamental concepts of objects, classes, methods, and access modifiers. Students will learn how to design and create classes, create objects from those classes, define methods to encapsulate behavior, and control access to class members using access modifiers. The module will provide a solid foundation in object-oriented programming (OOP) principles, enabling students to write more organized, reusable, and maintainable code.

MODULE LEARNING OBJECTIVES

By the end of this module, you should be able to:

- Understand the Fundamentals of Object-Oriented Programming (OOP):
- Understand the concept of state and behavior associated with objects.
- Define classes as blueprints for creating objects with common attributes and behaviors.
- Understand methods as functions defined within classes to encapsulate behavior.
- Define access modifiers and their role in controlling the visibility of class members.

LEARNING CONTENTS

7.1 Classes

In Python, a class is a blueprint or template for creating objects with specific attributes and behaviors. It defines the structure and characteristics of the objects that will be created based on that class. A class acts as a user-defined data type, allowing you to model real-world entities and create custom data structures.

7.1.1 Creating a Class

To create a class in Python, you use the class keyword followed by the class name. Inside the class, you can define attributes (data) and methods (functions) that belong to the class and its objects. Here's the basic syntax for creating a class:

Example

```
class ClassName:
    def __init__(self, parameter1, parameter2, ...):
        # Initialize instance attributes here
        self.attribute1 = parameter1
        self.attribute2 = parameter2
        # ...

    def method1(self, parameter1, parameter2, ...):
        # Method code here
    def method2(self, parameter1, parameter2, ...):
        # Method code here
```

Let's create a simple class called Person to represent a person with attributes like name and age, along with a method to introduce themselves:

Example

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

    def introduce(self):
        return f"My name is {self.name} and I am {self.age} years old."

person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

print(person1.name)
print(person2.age)
print(person1.introduce())
print(person2.introduce())
```

Create objects (instances) of the 'Person' class

Access object attributes

Call object methods

Output

```
Alice
25
My name is Alice and I am 30 years old.
My name is Bob and I am 25 years old.
```

In the example above, we defined a class named Person with an __init__() method to initialize instance attributes name and age. We also defined a method called introduce() to return a string introducing the person. To create objects of the Person class, we called the class as if it were a function, passing the required arguments to the __init__() method. Each object (person) created from the class has its own set of attributes and can call the introduce() method.

7.1.2 The __init__ function

In Python, the __init__() function is a special method that is automatically called when you create an object (instance) of a class. It is known as the constructor method and is used to initialize the attributes (data) of the object. This is defined within a class and always takes the `self` parameter as its first argument. The `self` parameter refers to the instance of the class (the object itself) and is used to access and manipulate the object's attributes.

Here's the general syntax of the __init__() method:

Example

```
class ClassName:
    def __init__(self, parameter1, parameter2, ...):
        # Initialize instance attributes here
        self.attribute1 = parameter1
        self.attribute2 = parameter2
        # ...
```

When you create an object of the class, Python automatically calls the __init__() method and passes the object as the `self` parameter. Any additional arguments you provide during object creation will be

passed to the __init__() method and used to initialize the instance attributes. Using the __init__() method allows you to set up the initial state of objects and define the necessary attributes that each object should have. It is a crucial part of defining classes in Python and is commonly used to set the initial values for object attributes.

7.2 Objects

In Python, an object is a fundamental concept in object-oriented programming (OOP). An object is an instance of a class, which is a blueprint that defines the attributes and methods that the object will have. In simple terms, an object is a specific entity created from a class, and it encapsulates both data (attributes) and behavior (methods). In addition, simple data types (integers, floats, strings, etc.) and more complex data structures (lists, dictionaries, sets, etc.) are all objects. Even functions and classes are objects in Python.

7.2.1 Creating an object

To create objects in Python, you need to define a class first. A class is a blueprint that specifies the attributes and behaviors (methods) that the objects of that class will have. Once the class is defined, you can create objects (instances) of that class.

Here's the general process for creating an object in Python:

1. Define the class using the class keyword, followed by the class name (by convention, class names start with an uppercase letter).
2. Inside the class, define an __init__() method. This special method is called when an object is created and is used to initialize its attributes.
3. Within the __init__() method, you can define the instance attributes that will hold data specific to each object. These attributes are created using the `self` keyword.
4. To create an object of the class, call the class as if it were a function, passing any required arguments to the __init__() method.

Let's create a simple class called Person to represent a person with attributes like name and age:

Example

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

    def introduce(self):
        return f"My name is {self.name} and I am {self.age} years old."

person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

print(person1.name)
print(person2.age)
```

Output

```
Alice
25
```

In the example above, we defined a class named Person with an __init__() method to initialize instance attributes name and age. We then created two objects (person1 and person2) of the Person class by

calling the class as if it were a function and passing the required arguments. Each object has its own set of attributes that hold different data.
Creating objects allows you to create instances of a class with specific data, and each object can have its own unique properties and behaviors defined in the class. This is the basis of object-oriented programming (OOP) in Python, which helps you model real-world entities and create modular, reusable code.

7.2.2 Modifying object properties

In Python, you can modify the properties (attributes) of an object after it has been created. Since attributes are variables bound to the object, you can assign new values to them at any time. You don't need to define any additional methods or special syntax to modify object properties.

Here's an example of how to modify object properties:

Example

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

person = Person("Alice", 30)

print("Original name:", person.name)
print("Original age:", person.age)

person.name = "Alicia"
person.age = 31

print("Modified name:", person.name)
print("Modified age:", person.age)
```

Access and print the initial values of attributes

Modify object properties

Access and print the modified values of attributes

Output

```
Original name: Alice
Original age: 30
Modified name: Alicia
Modified age: 31
```

In the example above, we defined a class Person with attributes name and age in the __init__() method. After creating an object person, we accessed and printed the initial values of the attributes. To modify the object properties, we simply assign new values to the attributes using dot notation. We changed the name attribute from "Alice" to "Alicia" and the age attribute from 30 to 31.

7.2.2 Deleting object/object properties

In Python, you can delete an object using the del keyword. When you delete an object, it removes the reference to that object, making it eligible for garbage collection. Once an object is deleted, you can no longer access its attributes or call its methods. It's important to note that Python automatically handles memory management, and you don't typically need to manually delete objects. Python's garbage collector automatically frees up the memory occupied by objects that are no longer in use, and it takes care of object deletion for you.

Deleting an object is usually not necessary in most Python programs. Memory management is efficient, and objects are automatically deleted when they are no longer needed or have no active references to them.

Here's an example of how to delete an object:

Example

```
class MyClass:
    def __init__(self, value):
        self.value = value

obj = MyClass(42)
print(obj.value)
del obj
print(obj.value)
```

Create an object (instance) of the class

Output: 42

Delete the object

Trying to access the object after deletion will raise an

Output

```
NameError: name 'obj' is not defined
```

In the example above, we created an object `obj` based on the class MyClass. After accessing the attribute `value`, we used `del obj` to delete the object. If we try to access the `obj` after deletion, it raises a `NameError` because the object no longer exists in memory.

Here's an example of how to delete an object property:

Example

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
person = Person("Alice", 30)

print("Original name:", person.name)
print("Original age:", person.age)

del person.name
print("Deleted name:", person.name)
```

Output: Alice

Output: 30

Trying to access the deleted property will raise an

Output

```
AttributeError: 'Person' object has no attribute 'name'
```

In the example above, we defined a class Person with attributes name and age in the __init__() method. After creating an object person, we accessed and printed the initial values of the attributes. To delete the name property from the object, we used del person.name. After deletion, attempting to access the name property raises an AttributeError since the property no longer exists in the object.

7.3 Methods

In Python, methods are functions defined inside a class that operate on objects created from that class. They are an essential part of object-oriented programming (OOP) and are used to define the behaviors (actions) that objects of a class can perform. Methods are defined within a class using the `def` keyword,

followed by the method name and any parameters the method takes. The first parameter of a method is conventionally named `self`, which refers to the instance of the class (the object itself) and is used to access and modify object attributes.

7.3.1 Creating Methods

To create methods in Python, you define them within a class. A method is a function that operates on objects created from that class. It takes the `self` parameter as its first argument, which represents the instance of the class (the object itself) and is used to access and modify object attributes.

Here's the general syntax for creating methods in Python:

```
class ClassName:
    def __init__(self, parameter1, parameter2, ...):
        # Initialize instance attributes here
        self.attribute1 = parameter1
        self.attribute2 = parameter2
        # ...

    def method1(self, parameter1, parameter2, ...):
        # Method code here
        pass

    def method2(self, parameter1, parameter2, ...):
        # Method code here
        pass
```

Let's create a class called Rectangle with methods to calculate the area and perimeter of a rectangle:

Example

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

rect = Rectangle(5, 3)

print("Area:", rect.area())
print("Perimeter:", rect.perimeter())
```

Create an object (instance) of the 'Rectangle' class

Output: 15

Output: 16

In this example, we defined a class `Rectangle` with attributes `length` and `width` in the __init__() method. We also defined two methods, area() and perimeter(), to calculate the area and perimeter of the rectangle, respectively. After creating an object `rect `of the `Rectangle` class, we called the area() and perimeter() methods on the object using dot notation. The object itself (`rect`) is automatically passed as the `self` parameter to the methods.

7.3.2 Modifying Methods

Modifying methods often refers to altering the functionality or behavior of existing methods in a class or piece of software. This procedure is crucial to software development since it enables you to modify and expand current functionality without having to completely rebuild the codebase. Methods can be changed for several reasons, including bug fixes, performance enhancements, the addition of new features, and the customization of behavior to meet needs. When modifying methods, it's essential to understand the existing implementation and the impact of your changes. The modified methods must be thoroughly tested to make sure they function as intended and do not have any unforeseen side effects. To keep the codebase readable and consistent, think about adhering to best practices and coding conventions.

There are a few common ways to modify methods:

1. **Overriding Methods**: In object-oriented programming (OOP), you can override methods from a superclass in a subclass. By doing so, the subclass provides a new implementation for the method that replaces the behavior defined in the superclass. This is a fundamental concept in inheritance, and it allows you to specialize or extend the behavior of the original method.
   Here's a step-by-step explanation of overriding methods:
   a. **Inheritance**: A subclass inherits properties and methods from its superclass. This includes both the attributes (variables) and methods (functions) defined in the superclass.
   b. **Method Overriding:** If a subclass wants to modify or extend the behavior of a method inherited from the superclass, it can do so by defining a method with the same name in the subclass. This new method will override the behavior of the inherited method.
   c. **Method Signature:** To successfully override a method, the method name, and its signature (number and types of parameters) in the subclass must exactly match that of the method in the superclass.
   d. **Access to Superclass Method:** Inside the overridden method in the subclass, you can use the super() function to call the superclass's version of the method. This allows you to reuse the original functionality and then customize it further, if needed.
   Here's an example:

Example

```
class Animal:
    def speak(self):
        print("Animal speaks")
class Dog(Animal):
    def speak(self):
        print("Dog barks")
class Cat(Animal):
    def speak(self):
        print("Cat meows")

dog = Dog()
dog.speak()

cat = Cat()
cat.speak()
```

Output

```
Dog barks
Cat meows
```

In this example, the `Animal` class has a method called speak(). Both the `Dog` and `Cat` classes inherit this method. However, they override it by providing their own versions of the method. When you call speak() on a `Dog` object, it prints "Dog barks," and when you call it on a `Cat` object, it prints "Cat meows."

2. **Decorators**: Decorators are a way to modify the behavior of functions or methods in Python. They allow you to wrap the original method with additional functionality before or after its execution.

   Here's a step-by-step explanation of method decorators:

   a. **Create the Decorator:** First, you define a function that serves as the decorator. This function takes another function (the method to be decorated) as an argument and typically returns a new function that extends or modifies the behavior of the original function.
   b. **Decorate the Method:** To use the decorator, you apply it to the method you want to modify by placing the @decorator_name syntax above the method definition in the class.
   c. **Use the Decorated Method:** When you call the decorated method on an instance of the class, the decorator's behavior is executed before and/or after the original method's execution.

   Here's an example of a method decorator:

Example

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Something is happening before the method is called.")
        result = func(*args, **kwargs)
        print("Something is happening after the method is called.")
        return result
    return wrapper

class MyClass:
    @my_decorator
    def some_method(self, x, y):
        return x + y

obj = MyClass()
result = obj.some_method(3, 5)
print(result)
```

Output

```
Something is happening before the method is called.
Something is happening after the method is called.
8
```

In this example, we have defined a decorator called `my_decorator`, which prints a message before and after the execution of the decorated method. We then apply this decorator to the `some_method` within the MyClass class. When we create an instance of `MyClass` and call

some_method(3, 5), the decorator's behavior is executed before the method's logic. The decorator provides additional functionality without modifying the original method's implementation.

3. **Method Extension:** Sometimes, you may want to keep the original functionality of a method but still add some additional actions or checks. It refers to the process of adding extra functionality to an existing method without modifying its original implementation. This is achieved by calling the original method and then performing additional actions before or after its execution. Method extension is particularly useful when you want to add extra features to a method without altering its core behavior, allowing for code reusability and maintainability. In this case, you can extend the method by calling the original method and then performing your extra steps. The concept of method extension can be applied in different programming paradigms, including object-oriented programming (OOP) and functional programming.
   Here's a step-by-step explanation of method extension:
   a. **Identify the Method to Extend:** First, you identify the method that you want to extend. It can be a method within a class or a standalone function.
   b. **Create the Extension:** Next, you define a new function or method that will extend the functionality of the original method. This extension function typically takes the same parameters as the original method.
   c. **Call the Original Method:** Inside the extension function, you call the original method with the same arguments it expects. This ensures that the core functionality is executed.
   d. **Add Extra Actions:** After calling the original method, you can include additional code to perform extra actions or modifications to the result returned by the original method.

   Here's an example demonstrating method extension:

Example

```
class MathOperations:
    def add(self, a, b):
        return a + b

def extended_add(obj, a, b):
    result = obj.add(a, b)
    print(f"The sum of {a} and {b} is: {result}")
    return result

math_ops = MathOperations()
math_ops.add = lambda a, b: extended_add(math_ops, a, b)

math_ops.add(3, 5)
```

Call the original 'add' method

Extend the 'add' method of MathOperations using the extended_add function

Now, calling the 'add' method will include the extra actions defined in extended_add

Output

```
Something is happening before the method is called.
Something is happening after the method is called.
8
```

In this example, we have a `MathOperations` class with an add method that performs basic addition. We want to extend this method to print the result along with the input numbers. To achieve this, we define the `extended_add` function, which calls the original add method and

then prints the result with the input values. Then extend the add method of the `MathOperations instance` by assigning a lambda function to it. The lambda function calls our `extended_add function` with the necessary arguments. When calling math_ops.add(3, 5), the `extended_add function` is invoked, which, in turn, calls the original add method with the same input values and prints the result with the extra information.

7.4 Access Modifiers

Unlike in languages like Java or C++, access modifiers are not enforced through keywords in Python. Instead, Python operates under the tenet that "we are all consenting adults here," which mandates that programmers adhere to standards and best practices that protect users' privacy and encapsulation. This method encourages a more adaptable and dynamic style of programming and gives developers access to class members without stringent visibility constraints. However, programmers may use naming conventions to denote a class member's intended visibility.

The common conventions used in Python are as follows:

1. **Public:** Publicly accessible class members are defined without any leading underscores. By tradition, both inside and outside the class, access to these members is deemed public and is available from anyone.

   Example

```
class MyClass:
    def public_method(self):
        # Public method

    public_var = 10
```

   Public member variable

2. **Private**: Private class members are identified by a single leading underscore ('_') before their names. These members are often regarded as private, and their use should be restricted to the class definition.

   Example

```
class MyClass:
    def _private_method(self):
        # Private method

    _private_var = 20
```

   Private member variable

3. **Protected**: Unlike several other languages, Python does not strictly enforce protected access. For members that are meant to be protected, it is customary to use a single leading underscore `_`. Although they are regarded as belonging to the class and its subclasses, these members are nonetheless accessible from outside the class.

   Example

```
class MyClass:
    def _protected_method(self):
        # Protected method

    _protected_var = 30
```

   Protected member variable

It's crucial to realize that these conventions are only suggestions for developers and not requirements imposed by the Python interpreter. Because Python is intended to be a language with a high level of readability and expressiveness, developers may rely on one another to use the class members properly. While it is possible to access "private" members in Python, it is recommended to follow conventions to preserve code readability and abide by sound program design principles, such as encapsulation. These conventions assist other developers to understand the intended use of class members and make the code easier to maintain.