Chapter
5

Object Oriented Programming with Python

5.1 Classes and Objects

Topics Covered: Introduction to OOP, Programmer-defined types, Attributes, Rectangles, Instances as return values, Objects are mutable, Copying.

5.1.1 Introduction

Object-oriented programming (OOP) is a programming paradigm that mirrors real-world entities as software objects, encompassing both data and methods. OOP aims to produce organized and reusable code, avoiding redundancy. The program is structured into self-contained objects or multiple mini programs. Each distinct object represents a specific part of the application, possessing its unique logic and data for seamless communication among themselves.

Table 5.1: Some Differences between OOP and POP

Object Oriented Programming (OOP)	Procedure Oriented Programming (POP)
It is a bottom-up approach	It is a top-down approach
Program is divided into objects	Program is divided into functions
Makes use of Access modifiers 'public', private', protected	Doesn't use Access modifiers
It is more secure	It is less secure
Object can move freely within member functions	Data can move freely from function to function within programs
It supports inheritance	It does not support inheritance

5.1.2 Object Oriented features supported in Python

Python supports several object-oriented programming (OOP) features, making it a versatile and powerful language for building complex applications. Some of the key OOP features in Python include:

Classes and Objects: Python allows you to define classes, which are blueprints for creating objects. Objects are instances of classes and represent real-world entities with attributes (data) and methods (functions).

Inheritance: Python supports inheritance, where a subclass can inherit attributes and methods from its superclass. This allows for code reuse and creating specialized classes based on existing ones.

Encapsulation: Python supports encapsulation, which means that the attributes and methods of a class can be hidden or protected from direct access outside the class. This is achieved using private and protected access modifiers.

Polymorphism: Python supports polymorphism, which allows different classes to have the same method name but different implementations. This is achieved through method overriding and method overloading.

Abstraction: Python supports abstraction, where you can create abstract classes that define the structure of a class but leave the implementation details to its subclasses. This is achieved using abstract methods and abstract classes.

Operator Overloading: Python allows you to define special methods (e.g., __add__(), __sub__(), etc.) that override the behavior of built-in operators when used with objects of your class.

Data Hiding: Python supports data hiding, allowing you to hide the implementation details of a class and expose only necessary attributes and methods. This is achieved using private variables and methods.

Class Methods and Static Methods: Python allows you to define class methods and static methods. Class methods are bound to the class and can access and modify class-level attributes. Static methods are not bound to the class or instance and are used for utility functions within the class.

With these OOP features, Python provides a robust and flexible framework for building object-oriented applications. Developers can use these features to create well-organized, modular, and maintainable code by leveraging the principles of abstraction, encapsulation, inheritance, and polymorphism.

5.1.3 Classes and objects

Programmer-defined types: In our journey through Python's built-in types, we now venture into defining a new type. To illustrate this, let's introduce a custom type named "**Point**," which will be used to represent points in a two-dimensional space.

In mathematical notation, points are conventionally denoted within parentheses, with the coordinates separated by a comma. For instance, the origin is represented as (0, 0), while a point (x, y) signifies an offset of x units to the right and y units upwards from the origin.

To represent points in Python, there are a few approaches we can consider:

- We could choose to store the coordinates separately using variables, such as x and y.
- Alternatively, we could opt to store the coordinates as elements within a list or tuple.
- The third option involves creating a new type, treating points as objects.

The third approach, although more involved, comes with distinct advantages that will become evident shortly. This type, which is defined by the programmer, is commonly referred to as a **"class."**

Classes and Objects: Class is an abstract data type which can be defined as a **template or blueprint** that describes the common attributes and behavior / state of the entity the programmer has to model in the program.

Examples

a. House plan containing the information like size, floors, doors, windows, locations of rooms, lawn, parking space and so on.
b. Blue print of the car containing the generic information of the car like name, size, shape, model, color and so on.

5.1.4 Defining a Class in Python

In Python, you can define a class using the class keyword. A class is like a blueprint that specifies the structure and behavior of objects. It contains attributes (variables) and methods (functions) that define the characteristics and actions of the objects created from that class. The general syntax to define a class is as follows:

```
class ClassName:
    def __init__(self, arg1, arg2, ...):
```

```
    # Constructor method (optional) - initializes the object's attributes
    self.attribute1 = arg1
    self.attribute2 = arg2
    # ...
def method1(self, ...):
    # Method 1 - defines actions the object can perform
    # ...
def method2(self, ...):
    # Method 2 - defines more actions
    # ...
```

Let's go through each part

class ClassName: This is the starting line of the class definition. Replace "ClassName" with the desired name of your class. By convention, class names use CamelCase (first letter of each word capitalized).

def __init__(self, arg1, arg2, ...): This is the constructor method (optional) and is called when an object is created from the class. It initializes the object's attributes with the values passed as arguments (arg1, arg2, ...). The first parameter of the constructor is typically named self, which refers to the instance of the object being created.

self.attribute1 = arg1: Within the constructor and other methods, self is used to refer to the object itself. You can assign values to object attributes using self.attribute_name.

def method1(self, ...): These are methods defined within the class. They represent the actions or behaviors that the objects can perform. Methods use self as the first parameter to access the object's attributes and perform operations on them.

To create an object (an instance of the class), you simply call the class as if it were a function:

object_name = ClassName(arg1_value, arg2_value, ...)
Now, object_name is an instance of the class ClassName, and you can use its attributes and methods as illustrated below:

object_name.method1(...)
object_name.method2(...)

This is the basic structure of defining a class in Python. You can expand the class by adding more attributes and methods as needed to represent the characteristics and behaviors of the objects you want to model.

Example1: Creating an object of Point Class. Let's consider a 2D Point as an example:

Class: Point

A class called "Point" can be defined to represent a 2D point in space. It would have two attributes (also called properties or data members): "x" and "y," representing the coordinates along the x and y axes, respectively. Additionally, the class may have methods (functions inside the class) that allow the objects to perform various operations.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
    def distance_from_origin(self):
        return (self.x ** 2 + self.y ** 2) ** 0.5
```

Object: 2D Point

Once the class "Point" is defined, we can create multiple objects (instances) of this class, each representing a distinct 2D point with its own set of coordinates.

```
# Creating Point objects
point1 = Point(3, 4)
point2 = Point(-2, 7)

# Accessing attributes of objects
print(point1.x, point1.y)  # Output: 3 4
print(point2.x, point2.y)  # Output: -2 7

# Using methods of objects
point1.move(2, 3)  # Move point1 by dx = 2, dy = 3
print(point1.x, point1.y)  # Output: 5 7

 # Calculate distance from the origin for point2
```

```
distance = point2.distance_from_origin()
print(distance)  # Output: 7.280109889280518
```

In this example, we defined a class "Point" with attributes "x" and "y" to represent 2D points. We then created two objects, point1 and point2, with different coordinates. Each object has its own set of "x" and "y" values, and they can use the methods defined in the class to perform specific operations.

By using classes and objects, we can create reusable and organized code, making it easier to model and manipulate real-world entities in our programs.

Example 2: Creating Employee class and its objects obj1 and obj2

```
class Employee:
    def __init__(self, emp_id, name, department, salary):
        self.emp_id = emp_id
        self.name = name
        self.department = department
        self.salary = salary

    def display_info(self):
        print(f"Employee ID: {self.emp_id}")
        print(f"Name: {self.name}")
        print(f"Department: {self.department}")
        print(f"Salary: ${self.salary:.2f}")

# Creating objects (instances) of the Employee class
obj1 = Employee(101, "John Doe", "IT", 55000)
obj2 = Employee(102, "Jane Smith", "HR", 48000)

# Displaying information of obj1 and obj2
print("Employee 1:")
obj1.display_info()

print("\nEmployee 2:")
obj2.display_info()
```

In the code above, we created an Employee class with attributes emp_id, name, department, and salary. The constructor method (__init__) initializes these attributes when we create objects of the class.

We then created two objects, obj1 and obj2, representing two employees with different information. Finally, we displayed the information of each employee using the display_info method defined in the Employee class.

5.1.5 Creating an Object in Python

Class only provides a blueprint. Class instances or objects implement the class. To use functions and data defined inside a class one should create instances or objects of the class. A class can have multiple objects. Each object will have their own copy of data and methods.

An Object is an instance of a Class. It is an entity that has state and behavior. In a nutshell, it is an instance of a class that can access the data.

Syntax: obj = class1()

Here obj is the "object " of class1.

A class is like a blueprint while an instance is a copy of the class with actual values. It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different instances, but without the class as a guide, you would be lost, not knowing what information is required.

An object consists of

- **State** : It is represented by **attributes** of an object. It also reflects the properties of an object.
- **Behavior** : It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity** : It gives a unique name to an object and enables one object to interact with other objects.

Example : Components of Student Object

Identity
Name: John Smith
Student ID: 123456
Date of Birth: January 15, 2000
Nationality: American
Contact Information: Email - john.smith@example.com, Phone - (555) 123-4567

State/Attributes
Age: 23
Gender: Male
Major: Computer Science
GPA: 3.75
Enrollment Status: Full-time
Residential Status: On-campus resident
Clubs and Organizations: Member of the Chess Club and Coding Society

Behaviors

Attending Classes: Regularly attends lectures and participates in class discussions.

Studying Habits: Spends several hours each day studying in the library or at home.

Participation: Actively participates in group projects and presentations.

Time Management: Balances academic commitments with extracurricular activities and personal time.

Communication: Communicates with professors to ask questions and seek clarification when needed.

Collaboration: Works well in team projects, contributing ideas and collaborating with classmates.

Problem-Solving: Engages in coding challenges and problem-solving competitions.

Leadership: Takes on the role of the coding society's treasurer and organizes fundraising events.

Example 1: Creating an object from a simple class

```
class MyClass:
    def __init__(self, value):
        self.value = value
# Creating an object named obj1 from the class MyClass
obj1 = MyClass(42)
# Accessing the object's attribute
print(obj1.value)  # Output: 42
```

Example 2: Creating multiple objects from the same class

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# Creating two objects, point1 and point2, from the class Point
point1 = Point(3, 5)
point2 = Point(-2, 7)

# Accessing attributes of the objects
print(point1.x, point1.y)  # Output: 3 5
print(point2.x, point2.y)  # Output: -2 7
```

Example 3: Using objects and their methods

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

# Creating an object named circle1 from the class Circle
circle1 = Circle(5)
# Calling the method area() on circle1
print("Area of circle1:", circle1.area())  # Output: 78.5
```

Example 4: Objects with custom methods

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

    def bark(self):
        print(f"{self.name} says: Woof!")

# Creating an object named dog1 from the class Dog
dog1 = Dog("Buddy", 3)

# Calling the method bark() on dog1
dog1.bark()  # Output: Buddy says: Woof!
```

In each of these examples, we defined a class and then created one or more objects (instances) of that class. The objects hold their own data (attributes) and can have their own unique behavior (methods). This way, we can model real-world entities and work with them in our Python programs.

5.1.6 Instances as return values

In Python, you can return instances of a class as return values from functions or methods. This allows you to create and customize objects outside the class itself. When a function returns an instance of a class, it is known as a factory function.

Let's illustrate this concept with an example using a Person class and a factory function called create_person:

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

def  create_person(name, age):
    return Person(name, age)

# Using the factory function to create instances of Person
person1 = create_person("Alice", 30)
person2 = create_person("Bob", 25)

# Accessing attributes of the objects
print(person1.name, person1.age)  # Output: Alice 30
print(person2.name, person2.age)  # Output: Bob 25
```

In this example, we defined a Person class with attributes name and age. Then, we created a factory function create_person, which takes name and age as arguments and returns an instance of the Person class with the given attributes.
By using the factory function create_person, we can create instances of the Person class with custom attributes without directly calling the class constructor. This approach can be useful when you need to create objects in different parts of your code or when object creation involves some additional logic or processing.

5.1.7 Objects are mutable

In Python, objects are generally mutable, which means their internal state can be modified after they are created. When an object is mutable, you can change its attributes, update its values, or even add new attributes.

Let's see an example to illustrate the mutability of objects:

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

# Creating an instance of the Person class
person1 = Person("Alice", 30)

# Modifying attributes of the object
person1.name = "Alicia"
person1.age = 31
# Adding a new attribute to the object
person1.city = "New York"
# Displaying the updated information
print(person1.name, person1.age, person1.city)

# Output: Alicia 31 New York
```

In the above example, we created a Person object named person1 with attributes name and age. After creating the object, we modified the name and age attributes with new values, and then we added a new attribute city to the object. These changes demonstrate the mutability of the Person object.

Note that not all objects in Python are mutable; some built-in objects like strings and tuples are immutable. Immutable objects cannot be changed after they are created. For mutable objects, any changes made to the object will be reflected in the same object's state throughout the program, unless explicitly overwritten or modified again.

5.1.8 Copying

In Python, the concept of copying objects is essential when you want to create a duplicate or a new independent instance of an object. Python supports two primary methods for copying objects: shallow copy and deep copy.

Shallow Copy: A shallow copy creates a new object but does not create new instances of the nested objects within the original object. Instead, it copies references to the nested objects. In other words, the copy points to the same nested objects as the original. If you modify the nested object in the copy, it will also affect the original object and vice versa.

In Python, you can create a shallow copy of an object using the copy() method or the built-in copy.copy() function from the copy module.

Example of a shallow copy

```
import copy
original_list = [1, 2, [3, 4]]
shallow_copy_list = copy.copy(original_list)

# Modify the nested list in the copy
shallow_copy_list[2][0] = 99

print(original_list)  # Output: [1, 2, [99, 4]]
print(shallow_copy_list)  # Output: [1, 2, [99, 4]]
```

Deep Copy: A deep copy creates a new object and recursively copies all the nested objects within the original object. In this way, the copy becomes fully independent of the original object, and any modifications to the nested objects in the copy will not affect the original object.

To perform a deep copy in Python, you can use the **copy.deepcopy()** function from the copy module.

Example of a deep copy

```
import copy

original_list = [1, 2, [3, 4]]
deep_copy_list = copy.deepcopy(original_list)

# Modify the nested list in the copy
deep_copy_list[2][0] = 99

print(original_list)  # Output: [1, 2, [3, 4]]
print(deep_copy_list)  # Output: [1, 2, [99, 4]]
```

As you can see, with the deep copy, the modification to the nested list in the copy does not affect the original list.

In summary, understanding the difference between shallow copy and deep copy is crucial when dealing with nested objects in Python. Shallow copies are faster and more memory-efficient but might lead to unexpected behavior if you're not aware of the shared references. On the other hand, deep copies provide independent duplicates of objects but can be slower and consume more memory, especially for large and complex objects. Choose the appropriate method based on your specific use case and requirements.

Let's consider a simple **Point** class representing a 2D point with x and y coordinates. We'll demonstrate the concepts of shallow copy and deep copy using instances of this class.

First, let's define the Point class:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x}, {self.y})"
```

Now, we'll create two instances of the Point class:

```
# Creating the original Point object
point1 = Point(2, 3)
print("Original Point:", point1)

# Creating another Point object using the first one
point2 = point1
print("Copied Point using assignment (shallow copy):", point2)
```

In this case, we haven't explicitly made a copy of point1. Instead, point2 is just another reference to the same object. Therefore, it is a shallow copy. Any changes to point2 will also affect point1:

```
point2.x = 10
print("Modified Point2:", point2)
print("Original Point after shallow copy change:", point1)
# Output:
Original Point: (2, 3)
Copied Point using assignment (shallow copy): (2, 3)
Modified Point2: (10, 3)
Original Point after shallow copy change: (10, 3)
```

Now, let's use the copy() method to create a shallow copy of the Point object:

```
import copy
# Creating the original Point object
point1 = Point(2, 3)
print("Original Point:", point1)
# Creating a shallow copy using the copy() method
point2 = copy.copy(point1)
print("Shallow Copy using copy() method:", point2)
```

Again, since this is a shallow copy, any changes to point2 will affect point1, and vice versa:

```
point2.x = 10
print("Modified Point2:", point2)
print("Original Point after shallow copy change:", point1)
# Output:
Original Point: (2, 3)
Shallow Copy using copy() method: (2, 3)
Modified Point2: (10, 3)
Original Point after shallow copy change: (10, 3)
```

Next, let's demonstrate a deep copy using the deepcopy() function:

```
# Creating the original Point object
point1 = Point(2, 3)
print("Original Point:", point1)
# Creating a deep copy using deepcopy() function
point2 = copy.deepcopy(point1)
print("Deep Copy using deepcopy() function:", point2)
```

Now, with the deep copy, **changes to point2 won't affect point1**, and vice versa:

```
point2.x = 10
print("Modified Point2:", point2)
print("Original Point after deep copy change:", point1)

# Output
Original Point: (2, 3)
Deep Copy using deepcopy() function: (2, 3)
Modified Point2: (10, 3)
Original Point after deep copy change: (2, 3)
```

As you can see, with the deep copy, the modification to point2 does not affect the point1 object. This behavior makes deep copy suitable when you want to create independent copies of objects with nested attributes.

5.2 Classes and Functions

Topics Covered: Classes and Functions, Time class and Objects, Pure functions, Modifiers, Prototyping versus planning.

5.2.1 Classes and Functions

After grasping the concept of creating new data types known as classes, the next step involves formulating functions that take custom objects created by the programmer as inputs and generate them as results. Here functions mean **User Defined Functions** and they are defined outside the class. Functions defined **outside of a class**, often referred to as standalone functions or external functions or functions, are not bound to any specific class and can be used globally within your code. They provide a way to encapsulate reusable code that can be used across different parts of your program without being tied to a particular class's attributes or methods.

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

    def calculate_area(self):
        return self.width * self.height

# Function to compare the areas of two rectangles
def compare_rectangle_areas(rect1, rect2):
    area1 = rect1.calculate_area()
    area2 = rect2.calculate_area()

    if area1 > area2:
        return "Rectangle 1 has a larger area."
    elif area2 > area1:
        return "Rectangle 2 has a larger area."
    else:
        return "Both rectangles have the same area."

# Main program
if __name__ == "__main__":
    width1 = float(input("Enter the width of rectangle 1: "))
    height1 = float(input("Enter the height of rectangle 1: "))
```

```
rect1 = Rectangle(width1, height1)

width2 = float(input("Enter the width of rectangle 2: "))
height2 = float(input("Enter the height of rectangle 2: "))
rect2 = Rectangle(width2, height2)

comparison_result = compare_rectangle_areas(rect1, rect2)
print(comparison_result)
```

In this example

- We define a Rectangle class with an **__init__** constructor to initialize width and height attributes and a **calculate_area** method to calculate the area of the rectangle.
- We then define a function **compare_rectangle_areas** outside the class that takes two rectangle objects as parameters and compares their areas.
- In the main part of the program, we create **two instances** of the Rectangle class based on user input.
- We call the **compare_rectangle_areas** function to compare the areas of the two rectangles and print the comparison result.

This example demonstrates how functions defined outside of a class can interact with class instances, using the data and methods provided by the class. External functions can be particularly useful for tasks that are not tied to a specific class's behavior or attributes. They promote code modularity, reusability, and better organization.

5.2.2 Time Class and Objects

A Time class is a blueprint that defines the structure and behavior of objects representing points in time. This class can have attributes to store hours, minutes, and seconds, as well as methods to manipulate and interact with these attributes.

Here's a basic example of a Time class:

```
class Time:
    def __init__(self, hours=0, minutes=0, seconds=0):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds
```

In this class:

The **__init__** method acts as a constructor. It initializes the Time object's attributes (hours, minutes, and seconds) with default values of 0 if no values are provided.

The **self** parameter refers to the instance being created and is used to access and modify the instance's attributes.

Creating Time Objects (Instances): Objects are instances of a class that embody the attributes and behavior defined in the class. To create a Time object, you use the class as a template and provide the necessary values for its attributes.

```
# Creating Time objects
t1 = Time(hours=10, minutes=30, seconds=45)
t2 = Time(hours=1, minutes=15, seconds=30)
```

In this example, we've created two Time objects, t1 and t2, by calling the class constructor **(__init__)** and passing in values for **hours, minutes, and seconds**.

Accessing Attributes and Methods: Once objects are created, you can access their attributes and methods using dot notation.

```
print(t1.hours, t1.minutes, t1.seconds)  # Output: 10 30 45
print(t2.hours, t2.minutes, t2.seconds)  # Output: 1 15 30
```

Adding Methods: Methods define the behavior of objects and allow them to perform actions or calculations. For example, you can define a method to display the time in a specific format:

```
class Time:
    def __init__(self, hours=0, minutes=0, seconds=0):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds

    def display_time(self):
        return
    f"{self.hours:02d}:{self.minutes:02d}:{self.seconds:02d}"
```

```
# Creating a Time object
t = Time(hours=8, minutes=45, seconds=15)
# Using the display_time method
print(t.display_time())  # Output: 08:45:15
```

In this example, the display_time method is defined **within the Time class** to format the time components.

You can define a **separate function display_time outside the class to display** the formatted time and then print the result. Here's how you can do it:

```
class Time
   def __init__(self, hours=0, minutes=0, seconds=0):
      self.hours = hours
       self.minutes = minutes
       self.seconds = seconds

def display_time(time_obj):
   return   f"{time_obj.hours:02d}:{time_obj.minutes:02d}:
               {time_obj.seconds:02d}"

# Creating a Time object
t = Time(hours=8, minutes=45, seconds=15)

# Using the display_time function and printing the result
formatted_time = display_time(t)
print("Formatted Time:", formatted_time)  # Output: Formatted Time:
08:45:15
```

In this example, the display_time function is defined outside the class Time. It takes a Time object as an argument and returns the formatted time string. After calling the function with the t object, we print the formatted time string using print(). This demonstrates how you can use external functions to operate on objects and present the results.

Consider another function named **"is_after"** function defined as follows:

```
def  is_after(t1, t2):
   return (t1.hour, t1.minute, t1.second) > (t2.hour, t2.minute, t2.second)
```

The **is_after** function takes two Time objects, t1 and t2, as input and returns **True** if t1 follows t2 chronologically, and **False** otherwise. The function uses tuple comparison to achieve this without using an explicit if statement.

Example usage:

```
time1 = Time(11, 59, 30)
time2 = Time(10, 30, 15)
```

```
print(is_after(time1, time2))  # Output: True (time1 follows time2 chronologically)
print(is_after(time2, time1))  # Output: False (time2 does not follow time1 chronologically)
```

In the **is_after** function, the tuples (t1.hour, t1.minute, t1.second) and (t2.hour, t2.minute, t2.second) are compared elementwise, allowing us to determine the chronological order of the two-time objects without using an explicit if statement.

5.2.3 Pure functions

A pure function is a type of function in object-oriented programming that does not change/modify the state of objects. **It means that a pure function does not depend on or modify any external state or data of the object outside of its scope.**

The add_time function discussed below is an example of a pure function.

```
def add_time(t1, t2):
    total_seconds_t1 = t1.hours * 3600 + t1.minutes * 60 + t1.seconds
    total_seconds_t2 = t2.hours * 3600 + t2.minutes * 60 + t2.seconds
    total_seconds_sum = total_seconds_t1 + total_seconds_t2

    sum_hour = total_seconds_sum // 3600
    sum_minute = (total_seconds_sum % 3600) // 60
    sum_second = total_seconds_sum % 60
    return Time(sum_hour, sum_minute, sum_second)
```

Example Usage

```
t1 = Time()
t1.hours = 10
t1.minutes = 30
t1.seconds = 45
t2 = Time()
t2.hours = 10
t2.minutes = 30
t2.seconds = 45
display_time(add_time(t1, t2)) #Output: 21:01:30
```

In this **add_time function**, two Time objects, t1 and t2, are passed as arguments. Inside the function, a new Time object named **sum** is created. The attributes of this new object (hour, minute, and second) are assigned the sum of the

corresponding attributes from t1 and t2. The new Time object sum is then returned as the result without altering the state/values of t1 and t2. This function is considered pure because it meets the following criteria:

Same Inputs, Same Outputs: For the same input **t1** and **t2**, the function will always return the same output, resulting in a predictable behavior.

No Side Effects: The function does not modify **the t1 and t2 objects**, nor does it alter any external data. It solely focuses on returning a new Time object as the result.

Independence from External State: The function does not rely on any global variables or external state, making it self-contained and isolated in its functionality.

5.2.4 Modifiers

Modifiers are functions in programming that **modify the attributes or state of objects directly**, and the changes made by these functions are visible to the caller. They alter the data they receive as arguments, and the modifications persist outside the function's scope.

In the provided concept, the example of a modifier function is given as increment. Let's examine it:

```
def increment(time, tseconds):
    time.seconds += tseconds
    while time.seconds >= 60:
        time.seconds -= 60
        time.minutes += 1
        while time.minutes >= 60:
            time.minutes -= 60
            time.hours += 1
            while time.hours >= 24:
                time.hours -= 24
```

The increment function takes a Time object time and a number of seconds as inputs. It modifies the time object by incrementing its second attribute by the given number of seconds. If the value of **time.seconds** exceeds sixty, it performs the necessary "carrying" of the extra seconds into the minute column and extra minutes into the hour column, ensuring that the time representation remains valid.

Example usage of the increment function

```
time = Time()
time.hours = 10
time.minutes = 30
time.seconds = 45
increment(time, 600)
display_time(time)  # Output: 10:40:45
```

In this example, the time object represents 10 hours, 30 minutes, and 45 seconds. The increment function is called with seconds = 30, and the time object is modified to 10 hours, 31 minutes, and 15 seconds.

The key characteristic of modifiers is that they directly alter the input objects and do not create new objects like pure functions. Any changes made to the object inside a modifier function are visible to the caller of the function.

5.2.5 Prototyping versus planning

Prototyping and planning are two different approaches to developing software solutions for a problem. Let's explain each approach:

Prototyping: Prototyping is an iterative and incremental development approach where developers create a preliminary or early version of a solution, known as a prototype, to explore the problem and gather feedback. The prototype typically contains essential features and functionalities to demonstrate the core aspects of the solution.

Prototyping involves quickly creating a working model of the software to demonstrate its functionality and gather feedback before developing the final version. It's more about experimenting and learning what works and what doesn't.

Prototype development involves:

- Writing a Rough draft of programs (or prototype)
- Performing a basic calculation
- Testing on a few cases , correcting flaws as we found them.

Prototype development leads code complicated. Let's create a prototype of a basic calculator using a Python script:

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

```
    def subtract(self, a, b):
        return a - b

# Create an instance of the CalculatorPrototype
prototype_calculator = CalculatorPrototype()

# Test the prototype
print(prototype_calculator.add(5, 3))        # Output: 8
print(prototype_calculator.subtract(10, 2)) # Output: 8
```

In this prototype, we've created a basic calculator class with add and subtract methods. The focus here is on quickly getting a functional version up and running to demonstrate the basic concept.

Advantages of Prototyping

Rapid Exploration: Prototyping allows quick exploration of ideas and potential solutions to a problem, enabling stakeholders to visualize and interact with an early version of the software.

Early Feedback: Stakeholders can provide feedback on the prototype, helping to identify issues, improvements, and changes early in the development process.

Iterative Refinement: Developers can refine and enhance the prototype through multiple iterations, gradually improving its quality and addressing concerns.

Disadvantages of Prototyping

Complexity Accumulation: Incremental corrections and patching errors in each iteration may lead to unnecessary complexities and difficulties in maintaining the codebase.

Uncertainty: Prototyping might not provide a deep understanding of the problem, leading to uncertainty about whether all errors have been identified and addressed.

Planning: Planning is a systematic and well-structured approach to software development, where developers analyze the problem, design the solution, and create a comprehensive plan before starting the implementation.

Planning involves careful design and consideration of requirements, architecture, and the overall structure of the software before implementation. This approach emphasizes creating detailed documentation, understanding user needs, and creating a well-thought-out design before writing code.

Let's create a planned version of the **calculator application:**

```
class Calculator:
def __init__(self):
self.value = 0

def add(self, num):
self.value += num

def subtract(self, num):
self.value -= num

def get_value(self):
return self.value

# Create an instance of the Calculator
calculator = Calculator()

# Perform calculations using the planned Calculator class
calculator.add(5)
calculator.subtract(2)

print("Result:", calculator.get_value()) # Output: Result: 3
```

In this planned version, we've considered the class structure more thoroughly, created an __init__ method for initialization, and organized methods logically. The planning process allows for a more organized and maintainable codebase.

Advantages of Planning

Clear Direction: Planning provides a clear roadmap for software development, ensuring that developers have a well-defined understanding of the problem and the intended solution.

Efficiency: A well-thought-out plan minimizes the chances of errors and allows for more efficient development and testing.

Easier Verification: A planned approach makes it easier to verify the correctness and completeness of the code since it is designed with a deep understanding of the problem.

Disadvantages of Planning

Time-Consuming: Planning can be time-consuming, especially for complex projects, as it involves detailed analysis and design before actual coding begins.

Less Flexibility: In some cases, planning may limit flexibility in adapting to unforeseen changes or new requirements that emerge during development.

In summary, prototyping focuses on creating early versions to explore and gather feedback, while planning emphasizes structured analysis, design, and a comprehensive approach to software development. Both approaches have their merits and drawbacks, and the choice between them depends on the nature of the problem, the complexity of the project, and the development team's preferences and requirements.

5.3 Classes and Methods

Topics Covered: Object-oriented features, Printing objects, Another example, A more complicated example, The __init__ method, The __str__ method, Operator overloading, Type-based dispatch, Polymorphism, Interface and implementation.

5.3.1 Object-oriented features

Python is an object-oriented programming (OOP) language, and it provides several features and concepts to support object-oriented programming. Here are some of the key object-oriented features in Python:

Classes and Objects: Python allows you to define classes, which are blueprints for creating objects. An object is an instance of a class, and it encapsulates both data (attributes) and behavior (methods).

Encapsulation: Python supports encapsulation, which means you can hide the internal details of a class from outside access. You can use private and protected access modifiers using underscores (e.g., **_private, __protected**) to indicate the level of visibility.

Inheritance: Inheritance allows you to create a new class that inherits properties and behaviors from an existing class. The new class is called a **subclass or derived class**, and the existing class is called the **superclass or base class**.

Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common base class. Python achieves **polymorphism** through method overriding and duck typing (accepting any object with a required set of methods).

Method Overriding: Subclasses can provide their own implementation of methods that are defined in the superclass. This is known as method overriding and allows customization of behavior.

Abstraction: Abstraction is the process of simplifying complex reality by modeling classes based on the essential features an object should have. In Python, you can use abstract base classes to define abstract methods that must be implemented by concrete subclasses.

Composition and Aggregation: Python supports creating complex objects by combining smaller objects. Composition represents a "has-a" relationship, while aggregation represents a "contains-a" relationship between objects.

Constructor and Destructor: Python classes have a special method called __init__ (constructor) that is automatically called when an object is created. The __del__ method (destructor) is called when an object is about to be destroyed.

Instance and Class Variables: Python supports both instance variables (belonging to objects) and class variables (shared among all instances of a class).

Getter and Setter Methods: Although not required, Python allows you to define methods for getting and setting the values of attributes, providing controlled access to class attributes.

Namespace and Scope: Classes provide their own namespace, allowing for better organization and encapsulation of variables and methods. Python also has the concept of global, local, and nonlocal scopes.

Operator Overloading: Python allows you to define custom behavior for standard operators (e.g., +, -, *) by implementing special methods like __add__, __sub__, and so on.
These are some of the key object-oriented features in Python that enable you to design and implement programs using the principles of object-oriented programming.

5.3.2 Real-world Representation

Objects in Python often represent entities or concepts from the real world. These objects embody the characteristics and behaviors of real-world entities. Methods within classes correspond to the ways these entities interact or perform actions in the real world.

By adhering to these principles, Python provides an effective and organized approach to programming, allowing developers to model real-world scenarios and create reusable and maintainable code using classes and objects.

5.3.3 Methods and Functions

In Python programming, both methods and functions are blocks of code that can be executed to perform specific tasks. However, there are some key differences between methods and functions

Definition and Scope of Functions and Methods

Functions: Functions are standalone blocks of code that can be defined outside of any class. They are created using the def keyword and can be called from anywhere in the code.

Methods: Methods are functions that are defined within a class and are associated with objects of that class. They are created using the def keyword within the class definition and can only be called on instances of the class.

Parameters in Functions and Methods

Functions: Functions can have any number of parameters, and they are specified in the function's definition.

Methods: Methods have an additional first parameter, usually named self, which represents the instance of the class calling the method. When calling methods on objects, you don't need to explicitly pass this parameter; Python automatically passes the instance (object) as the first argument.

Calling Syntax for Functions and Methods

Functions: Functions are called using their name followed by parentheses and any required arguments (if defined).

Methods: Methods are called on objects using the dot notation. The instance (object) on which the method is called becomes the self-parameter within the method.

Example of a Function

```
def square(num):
    return num ** 2
result = square(5)
print(result)  # Output: 25
```

Example of a Method

```
class MathOperations:
    def add(self, a, b):
        return a + b
obj = MathOperations()
sum_result = obj.add(3, 7)
print(sum_result)  # Output: 10
```

In the **function example**, we have a simple function **square()** that calculates the square of a given number. It takes one argument num and returns the square of that number.

In the **method example**, we have a class MathOperations with a method **add()**. The **add()** method takes two arguments a and b, and it adds them together. Notice that we use the dot notation (**obj.add()**) to call the method on the obj instance of the MathOperations class.

In summary, methods are functions that are associated with objects and are called on instances of classes. They have an additional self parameter to represent the calling instance. Functions, on the other hand, are standalone blocks of code that can be called from anywhere in the program. They don't have the self parameter and are not associated with objects or classes.

Methods vs Functions: Methods and functions are both blocks of code in Python that can be executed to perform specific tasks, but they have some key differences:

Table 5.2:

	Methods	**Functions**
Definition and Scope	Methods are functions that are defined within a class and are associated with objects (instances) of that class. They are created using the def keyword inside the class definition and can only be called on instances of the class.	Functions are standalone blocks of code that can be defined outside of any class. They are created using the def keyword and can be called from anywhere in the code.
First Parameter (self/ No Self)	Methods have an additional first parameter, usually named self, which represents the instance of the class calling the method. When calling methods on objects, you don't need to explicitly pass this parameter; Python automatically passes the instance (object) as the first argument.	Functions don't have the self-parameter like methods do. They are not associated with objects or classes and can't access object attributes.
Scope/ Access to Object	Methods have access to the attributes (data) of the object on which they are called. They	Functions can access global variables and other functions within their scope, but they

Attributes	can read and modify the object's attributes as needed.	don't have access to attributes specific to a class instance.
Calling Syntax	Methods are called using the dot notation, with the object instance followed by the method name and any required arguments.	Functions are called using their name followed by parentheses and any required arguments

Example	class Circle:
	def __init__(self, radius):
	self.radius = radius
	def calculate_area(self):
	return 3.14 * self.radius ** 2
	
	# Creating an instance of the Circle class
	circle1 = Circle(5)
	
	# Calling the calculate_area() method on the circle1 object
	
	area = circle1.calculate_area()
	
	print(area) # Output: 78.5
		class Circle:
		def __init__(self, radius):
		self.radius = radius
		
		# External function to calculate the area of a circle
		
		def calculate_circle_area(circle):
		return 3.14 * circle.radius ** 2
		
		# Creating an instance of the Circle class
		
		circle1 = Circle(5)
		
		# Calling the external calculate_circle_area() function
		
		area = calculate_circle_area(circle1)
		print(area) **# Output**: 78.5

In summary, the main differences between methods and functions are their association with objects and classes, the presence of the self-parameter in methods, and the access to object attributes. Methods are used to define behavior specific to class instances, while functions are standalone blocks of code that can be called from anywhere.

Transforming functions into methods: Transforming functions into methods involves incorporating the functions within a class and defining them as methods. This allows the methods to access and interact with the attributes and

behavior of objects (instances) of that class. Here's a step-by-step guide on how to transform a function into a method:

Step 1: Define the Class

Start by defining the class that will contain the method. Use the class keyword followed by the class name and a colon.

Step 2: Convert the Function into a Method

Take the function you want to transform into a method and indent its entire code under the class definition. Add the self parameter as the first parameter of the method. This allows the method to access the attributes and methods of the class instance (object).

Step 3: Modify Function References

Inside the method, replace any references to variables or functions with self. followed by the variable or function name. This ensures that the method accesses the class attributes and methods correctly.

Example of Transforming a Function into a Method: Suppose we have a simple function add() that adds two numbers:

```
def  add(a, b):
    return a + b
result = add(3, 5)
print(result)  # Output: 8
```

Now, let's transform this function into a method within a class:

```
class  Calculator:
   def  add(self, a, b):
        return a + b
# Creating an instance of the Calculator class
calculator = Calculator()

# Calling the add() method on the calculator object
result = calculator.add(3, 5)
print(result)  # Output: 8
```

In the transformed version, we defined the Calculator class and added the **add()** method within it. We added the self parameter to the method to make it a

method associated with class instances. We also changed the function reference **add()** to **self.add()** inside the method to access the method correctly.

Now, the **add()** function is successfully transformed into a method and can be called on instances of the Calculator class.

Remember that transforming a function into a method allows you to access and manipulate class attributes and provides a more object-oriented approach to organizing your code.

5.3.4 Optional Arguments

Optional arguments, also known as default arguments, are parameters in a Python function that have default values assigned to them. When calling a function with optional arguments, if you don't provide a value for the optional parameter, the function will use its default value. However, you can override the default value by explicitly passing a different value for the optional parameter when calling the function.

Here's an example of using optional arguments:

```
def   calculate_average(numbers, rounding=True):
    average = sum(numbers) / len(numbers)
    if rounding:
       return round(average)
    else:
       return average

# Using the function with a list of numbers and the default rounding
(True)
numbers_list1 = [10, 15, 20, 25, 30]
result1 = calculate_average(numbers_list1)
print(result1)  # Output: 20

# Using the function with a list of numbers and turning off rounding
(False)
numbers_list2 = [5, 7, 9]
result2 = calculate_average(numbers_list2, rounding=False)
print(result2)  # Output: 7.0
```

In this example, the calculate_average() function now takes a list of numbers as the required argument numbers and an optional argument rounding, which is set to True by default.

When calling the function with a list of numbers (e.g., numbers_list1 = [10, 15, 20, 25, 30]), the function will calculate the average of the numbers and, by default, round the result to the nearest integer using the round() function.

In the first call calculate_average(numbers_list1), we use the default value True for the rounding argument, so the function rounds the average to the nearest integer, resulting in an output of 20.

In the second call calculate_average(numbers_list2, rounding=False), we explicitly pass False for the rounding argument, which turns off the rounding behavior, and the function returns the average as a floating-point number without rounding, resulting in an output of 7.0.

Using a list of numbers as an optional argument allows the function to operate on different lists with different rounding preferences without requiring the caller to always provide the rounding argument.

5.3.5 Operator overloading

Operator overloading in Python allows you to define special methods in your classes that define the behavior of built-in operators (e.g., +, -, *, /) when used with objects of your class. By defining these special methods, you can customize how the operators work with your objects and make your classes behave like built-in data types.

In Python, operator overloading is achieved by implementing special methods with double underscores (dunder_methods). For example, to overload the addition operator +, you need to define the **__add__()** method in your class.
Let's illustrate operator overloading with an example of a custom Vector class representing 2D vectors:

```
class  Vector:
   def   __init__(self, x, y):
        self.x = x
        self.y = y
   def   __add__(self, other):
      # Overloading the addition operator (+)
       return Vector(self.x + other.x, self.y + other.y)
   def   __sub__(self, other):
      # Overloading the subtraction operator (-)
        return Vector(self.x - other.x, self.y - other.y)
   def __mul__(self, scalar):
      # Overloading the multiplication operator (*) with a scalar
```

```
        return Vector(self.x * scalar, self.y * scalar)
    def __str__(self):
        return f"Vector({self.x}, {self.y})"

# Creating two Vector objects
v1 = Vector(3, 4)
v2 = Vector(1, 2)

# Using the overloaded addition operator
v3 = v1 + v2
print(v3)  # Output: Vector(4, 6)

# Using the overloaded subtraction operator
v4 = v1 - v2
print(v4)  # Output: Vector(2, 2)

# Using the overloaded multiplication operator with a scalar
v5 = v1 * 2
print(v5)  # Output: Vector(6, 8)
```

In this example, we defined a Vector class with x and y attributes to represent 2D vectors. We overloaded the addition (+), subtraction (-), and multiplication (*) operators by implementing the **__add__()**, **__sub__()**, and **__mul__()** methods, respectively.

When we use these operators on Vector objects, Python calls the corresponding special methods to perform the custom operations. For example, v1 + v2 calls the **__add__()** method, which returns a new Vector with the component-wise sum of v1 and v2.

Operator overloading allows you to provide meaningful and intuitive behavior for your custom classes when using built-in operators, making your classes more flexible and easier to work with in Python.

5.3.6 Method Overriding

Method overriding is a concept in object-oriented programming where a subclass provides ***a specific implementation for a method that is already defined in its superclass***. When a method is overridden, the subclass's version of the method takes precedence over the superclass's version when called on objects of the subclass.

In Python, method overriding is achieved by redefining a method in the subclass with the same name as a method in the superclass. This allows the subclass to provide its own implementation for the method, which may differ from the behavior defined in the superclass.

Here's an example to illustrate **method overriding** in Python:

```
class  Animal:
   def  make_sound(self):
      return "Generic animal sound"

class  Dog(Animal):
   def  make_sound(self):
      return "Woof!"

class  Cat(Animal):
   def  make_sound(self):
      return "Meow!"

# Creating instances of the classes
animal = Animal()
dog = Dog()
cat = Cat()

# Calling the make_sound method on each object
print(animal.make_sound())  # Output: Generic animal sound
print(dog.make_sound())     # Output: Woof!
print(cat.make_sound())     # Output: Meow!
```

In this example, we have a base class Animal with a method make_sound(), which provides a generic implementation for the sound of an animal. We then have two subclasses, Dog and Cat, which are inherited from the Animal class.
Both Dog and Cat classes override the make_sound() method by providing their own implementation of the method. When we call the make_sound() method on objects of the Dog and Cat classes, Python uses the overridden version of the method in the respective subclass, resulting in different outputs based on the specific implementation provided in each subclass.

Method overriding allows subclasses to customize the behavior of methods defined in their superclass, providing a way to specialize and extend the functionality of the inherited methods. It is a fundamental feature of inheritance in object-oriented programming.

5.3.7 __init__()

In Python, the **__init__()** method is a special method used for initializing the attributes of an object when it is created. It is also known as a constructor method. The **__init__()** method is automatically called when a new instance of a class is created, and it allows you to set the initial state of the object by defining its attributes and their initial values.

Syntax of __init__() method:

```
class  ClassName:
   def   __init__(self, param1, param2, ...):
            # Attribute initialization and other setup code here
             self.param1 = param1
            self.param2 = param2
            # ...
```

The **__init__()** method takes at least one parameter, conventionally named self, which refers to the instance of the class being created. It allows you to access and modify the attributes of the object within the method.

Let's see an example to illustrate the use of **__init__()** method:

```
class  Dog:
   def    __init__(self, name, age, breed):
            # Attribute initialization
             self.name = name
             self.age = age
             self.breed = breed
   def   bark(self):
             print(f"{self.name} says: Woof!")
# Creating an instance of the Dog class
dog1 = Dog("Buddy", 3, "Labrador")

# Accessing the attributes of the object
print(dog1.name)   # Output: Buddy
print(dog1.age)    # Output: 3
print(dog1.breed)  # Output: Labrador

# Calling the bark method of the object
dog1.bark()  # Output: Buddy says: Woof!
```

In this example, we define a Dog class with an **__init__()** method. When a new Dog object is created, the **__init__()** method is automatically called with the specified arguments (name, age, and breed) to initialize the attributes of the object.

We create an instance of the Dog class called dog1, passing the values "Buddy", 3, and "Labrador" for the name, age, and breed parameters, respectively. The **__init__()** method sets the name, age, and breed attributes of the object dog1.
After creating the object, we can access its attributes using dot notation (e.g., dog1.name, dog1.age) to retrieve the values we set during initialization.
The **__init__()** method is a fundamental part of object-oriented programming in Python, allowing you to ensure that your objects start with the correct initial state.

5.3.8 __str__ ( )method

The __str__() method is a special method in Python that allows you to define a string representation of an object. When you use the str() function or the print() function on an object, Python will automatically call the __str__() method to get the string representation of that object.

Syntax of __str__() method:

```
class ClassName:
    def __str__(self):
        # Return the string representation of the object
        return "String representation of the object"
```

The __str__() method takes only one parameter, conventionally named self, which refers to the instance of the class for which the string representation is being generated. The method should return a string that represents the object in a human-readable format.

Let's see an example of using the **__str__()** method:

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

    def __str__(self):
        return f"Name: {self.name}, Age: {self.age},
                Major:{self.major}"
```

```
# Creating an instance of the Student class
student1 = Student("Alice", 20, "Computer Science")

# Using the print function on the object
print(student1)
```

In this example, we define a Student class with an __init__() method to initialize the attributes (name, age, and major) of the object. Additionally, we define the __str__() method to provide a custom string representation for the Student objects.

When we use the print() function on the student1 object, Python automatically calls the __str__() method of the Student class. The __str__() method returns a formatted string containing the information about the student's name, age, and major.

The output will be:

Name: Alice, Age: 20, Major: Computer Science

The **__str__()** method is useful for providing a meaningful and descriptive representation of your objects. It allows you to customize how objects of your class are displayed as strings, making it easier for debugging and readability.

5.3.9 Type-based dispatch

Type-based dispatch, also known as "Type-Based Polymorphism" or "Method Overloading," is a technique used to define different implementations of a method based on the types of arguments passed to the method. This allows you to have different behaviors for the same method name depending on the types of arguments.

In Python, type-based dispatch can be achieved using **function overloading**, where you define multiple versions of a function with the same name but different argument types. However, it's important to note that Python does not support traditional method overloading like some other languages do (e.g., Java), where you can define multiple methods with the same name but different parameter lists.

Here's a simple example of type-based dispatch using function overloading in Python:

```
class Calculator:
  def add(self, a, b):
```

```
        if isinstance(a, int) and isinstance(b, int):
            return self._add_int(a, b)
        elif isinstance(a, float) and isinstance(b, float):
            return self._add_float(a, b)
        elif isinstance(a, str) and isinstance(b, str):
            return self._add_str(a, b)
        else:
            raise TypeError("Unsupported types for addition")
    def  _add_int(self, a, b):
        return a + b

    def  _add_float(self, a, b):
        return a + b

    def  _add_str(self, a, b):
        return a + b

calculator = Calculator()

print(calculator.add(5, 3))       # Output: 8
print(calculator.add(2.5, 1.5))   # Output: 4.0
print(calculator.add("Hello, ", "world!"))  # Output: Hello, world!
```

In this example, the Calculator class has an add method that checks the **types of its arguments and then delegates the actual addition operation to specific private methods based on the argument types**. This demonstrates a form of type-based dispatch by considering different argument types and performing different operations accordingly.

It's important to note that this example showcases a simulated form of method overloading since Python itself doesn't support true method overloading with multiple versions of the same method name.

5.3.10 Encapsulation

Programming (OOP). It is the process of bundling the data (attributes) and methods (functions) that operate on the data within a single unit, called a class. Encapsulation allows you to control the access to the internal state of an object, preventing direct access to its attributes from outside the class.

In Python, encapsulation is achieved using private and protected access modifiers. Conventionally, private attributes and methods are prefixed with a

double underscore __, and protected attributes and methods are prefixed with a single **underscore _**. Although Python does not enforce true access control like some other languages, developers are encouraged to follow the naming convention to indicate the intended visibility of attributes and methods.

Let's see an example of encapsulation in Python

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

   def  display_student_info(self):
        print(f"Name: {self.__name}, Age: {self._age}, Roll Number:
                {self.roll_number}")

   def   __secret_method(self):
      print("This is a secret method!")

# Creating an instance of the Student class
student1 = Student("Alice", 20, "A123")

# Accessing public attributes directly
print(student1.roll_number)  # Output: A123

# Accessing protected attribute (conventionally, should not be accessed
directly)
print(student1._age)        # Output: 20
# Attempting to access private attribute directly (will raise an
AttributeError)
# print(student1.__name)     # Uncommenting this line will raise an
AttributeError

# Accessing private attribute using name mangling (not recommended)
print(student1._Student__name)  # Output: Alice

# Calling public method
student1.display_student_info()  # Output: Name: Alice, Age: 20, Roll
Number: A123

# Attempting to call private method directly (will raise an AttributeError)
```

```
# student1.__secret_method()   # Uncommenting this line will raise an AttributeError
```

In this example, we define a Student class with three attributes: __name (private), _age (protected), and roll_number (public). We also have two methods: display_student_info() (public) and __secret_method() (private).
Outside the class, we can directly access the public attributes like roll_number and the protected attribute _age. However, attempting to access the private attribute __name directly will raise an AttributeError.

To access private attributes, we can use name mangling by prefixing the attribute with the class name, as shown in student1._Student__name. However, name mangling is not recommended as it goes against the principle of encapsulation.

Similarly, attempting to call the private method __secret_method() directly will raise an AttributeError. Private methods should not be accessed or called directly from outside the class.

Encapsulation in Python allows you to protect the internal state of an object and provides a clear interface for interacting with the object. It helps in maintaining the integrity of the class and makes the code more maintainable and robust.

5.3.11 Polymorphism

Polymorphism is one of the four fundamental principles of object-oriented programming (OOP). It refers to the ability of a class to take on multiple forms or the ability of different classes to be treated as objects of a common superclass. In Python, polymorphism allows you to use a single interface to represent different data types or objects, providing a more flexible and dynamic approach to programming. There are two main types of polymorphism in Python:

Compile-time Polymorphism (Static Polymorphism)
Achieved through method overloading and operator overloading.

Method overloading allows a class to define multiple methods with the same name but different parameters. The appropriate method is called based on the number or type of arguments passed during the function call.

Operator overloading allows a class to define special methods (e.g., __add__(), __sub__(), etc.) that override the behavior of built-in operators when used with objects of the class.

Runtime Polymorphism (Dynamic Polymorphism)

Achieved through method overriding.
Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. When the method is called on an object of the subclass, the subclass's version of the method is executed instead of the superclass's version.

Let's see an example of method overriding to demonstrate runtime polymorphism in Python:

```
class Animal:
    def make_sound(self):
        return "Generic animal sound"

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"

# Function to demonstrate polymorphism
def animal_sound(animal):
    return animal.make_sound()

# Creating instances of the classes
animal = Animal()
dog = Dog()
cat = Cat()

# Calling the animal_sound function with different objects
print(animal_sound(animal))  # Output: Generic animal sound
print(animal_sound(dog))     # Output: Woof!
print(animal_sound(cat))     # Output: Meow!
```

In this example, we have a base class Animal with a method make_sound(), which provides a generic implementation for the sound of an animal. We then have two subclasses, Dog and Cat, which are inherited from the Animal class.
The Dog and Cat classes override the make_sound() method with their specific implementations. When we call the animal_sound() function with objects of the

Animal, Dog, and Cat classes, Python automatically calls the appropriate version of the make_sound() method based on the actual object passed as an argument. This is an example of runtime polymorphism.

Polymorphism in Python allows us to write more flexible and reusable code by treating objects of different classes in a uniform way. It promotes code modularity and simplifies the process of adding new subclasses without modifying the existing code that uses the superclass interface.

5.3.12 Inheritance

Inheritance is one of the four fundamental principles of object-oriented programming (OOP). It allows a class (subclass) to inherit attributes and methods from another class (superclass). Inheritance promotes code reusability, as the subclass can reuse the functionalities of the superclass without having to redefine them.

The class that is being inherited from is called the superclass (or parent class), and the class that inherits from the superclass is called the subclass (or child class).

Syntax of inheritance in Python

```
class Superclass:
    # Superclass attributes and methods
class Subclass(Superclass):
    # Subclass attributes and methods, including those inherited from the superclass
```

Let's see an example of inheritance in Python

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

    def make_sound(self):
        return "Generic animal sound"

class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
```

```
        return "Meow!"

# Creating instances of the subclasses
dog = Dog("Buddy")
cat = Cat("Whiskers")

# Accessing attributes and methods of the subclasses
print(dog.name)                 # Output: Buddy
print(cat.name)                 # Output: Whiskers
print(dog.make_sound())         # Output: Woof!
print(cat.make_sound())                 # Output: Meow!
```

In this example, we have a base class Animal, which has an **__init__()** method to initialize the name attribute and a **make_sound()** method that provides a generic implementation for the sound of an animal.

We then have two subclasses, Dog and Cat, which inherit from the Animal class using the syntax class Subclass(Superclass). The Dog and Cat subclasses override the make_sound() method with their specific implementations for the sound of a dog and a cat.

When we create instances of the Dog and Cat classes (dog and cat objects), they inherit the attributes and methods from the Animal superclass. We can access the name attribute of each object, and when we call the **make_sound()** method on each object, Python automatically calls the version of the method defined in the subclass, providing the specific sound of the animal.

Inheritance allows us to create specialized classes based on existing classes, making the code more organized and easier to maintain. It promotes code reuse and fosters a hierarchical relationship between classes, allowing for a more natural representation of real-world entities and behaviors.

5.3.13 Interface and implementation

In the context of object-oriented programming (OOP), "interface" and "implementation" are two fundamental concepts that play a crucial role in designing and developing maintainable software.

Interface: An interface in OOP refers to the set of methods (functions) and properties (attributes) that a class provides to the outside world. It defines the contract or the "interface" through which other code can interact with the class.

The interface acts as a bridge between different parts of a program, enabling communication and interaction between classes without exposing their internal details or implementation.

In some programming languages like Java or C#, an interface can be a formal construct, explicitly defined using the interface keyword. However, in Python, there is no formal interface construct. Instead, an interface is represented implicitly through the set of methods that a class provides to the outside world.

Implementation: Implementation refers to the internal details and logic of a class that are not directly exposed to its users or other classes. It includes private attributes, helper methods, and any other internal functionality required to fulfill the public interface of the class.

The implementation of a class is encapsulated within the class, hidden from the outside world, and should not be directly accessible or modifiable by external code. This helps to maintain data integrity, protect the class from unwanted modification, and promote code modularity.

Example: Let's consider an example of a simple interface and implementation for a geometric shape class:

```
class Shape:
    def area(self):
        pass
    def perimeter(self):
        pass

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

class Rectangle(Shape):
    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)
```

In this example, we define an **interface Shape**, which is an abstract class with two abstract methods, area() and perimeter(). The Shape class serves as a blueprint for creating different types of shapes.

Then, we have two classes, Circle and Rectangle, which inherit from the Shape class. These subclasses provide concrete implementations for the area() and perimeter() methods, specific to circles and rectangles, respectively.

The Shape interface defines the contract that any shape class should adhere to. It allows the client code to interact with any shape without knowing its specific implementation details.

For instance, the following client code can work with any shape (circle, rectangle, or any other shape class that adheres to the Shape interface):

```
def  print_shape_details(Shape):
    print(f"Area: {shape.area()}")
    print(f"Perimeter: {shape.perimeter()}")

circle = Circle(5)
rectangle = Rectangle(3, 4)

print_shape_details(circle)
print_shape_details(rectangle)
```

This example demonstrates how using an interface allows us to create a flexible and maintainable design. The client code can work with different shapes without needing to know their specific implementation details, making it easier to add new shapes or modify existing ones without affecting the rest of the progr