MODULE 5 PYTHON

Introduction to python Programming

Syllabus: Classes and objects: Programmer-defined types, Attributes, Rectangles, Instances as return values, Objects are mutable, Copying, Classes and functions: Time, Pure functions, Modifiers, Prototyping versus planning, Classes and methods: 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,
Textbook 2: Chapters 15 – 17

1. CLASSES AND OBJECTS

In Python, **classes** are used to create **user-defined types**, which allow you to organize both data and functions. A **class** acts like a blueprint for creating objects, and an **object** is an instance of a class. Let's break this down simply:

Key Concepts

1. **Class**: A blueprint for creating objects.
    - Example: class Point:
2. **Object**: An instance of a class.
    - Example: p = Point()

Creating a Class

A class is defined using the class keyword, followed by the class name and a colon. Inside the class, you can define variables (attributes) and methods (functions). Here's a simple class:

```
class Point:
    """Represents a point in 2D space."""
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

- __init__ is a special method called the constructor, which initializes the object's attributes.
- self.x and self.y are attributes of the Point object representing its coordinates.

Creating an Object

To create an object from a class, you call the class like a function:

p = Point(3.0, 4.0)

- `p` is now a `Point` object with coordinates `x = 3.0` and `y = 4.0`.

Example Output

When you print the object:

print(p) # Output: Point(3.0, 4.0)

Python tells you the class (`Point`) and memory location (in hexadecimal).

Benefits of Using Classes and Objects

1. **Organization**: Classes allow you to group related data (attributes) and behavior (methods) together.
2. **Reusability**: Once defined, a class can be used to create multiple objects with the same structure.
3. **Extensibility**: New attributes or methods can be added to existing classes to extend functionality.

Example Scenario

Let's say you want to create a class for a **Car**:

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
    def __str__(self):
        return f"{self.make} {self.model}"
car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")
print(car1)  # Output: Toyota Corolla
print(car2)  # Output: Honda Civic
```

2. ATTRIBUTES

In Python, attributes are variables that are associated with an object (instance of a class). These attributes hold data that is specific to that object. Let's break it down simply:

Key Concepts

1. Dot Notation: You can assign and access attributes using dot notation.
    - Example: object.attribute
2. Assigning Values:
    - To assign values to attributes

object.attribute = value

Accessing Values:

- To access an attribute's value

value = object.attribute

Example: Point Class with Attributes

Let's create a simple Point class with two attributes: x and y.

Defining the Class

```
class Point:
    def __init__(self, x, y):
        self.x = x  # Attribute for the x-coordinate
        self.y = y  # Attribute for the y-coordinate
```

Creating an Object

```
p = Point(3.0, 4.0)  # p is a Point object with x = 3.0 and y = 4.0
```

Assigning and Accessing Attributes

- Assigning values to attributes:

```
p.x = 5.0  # Setting the x-coordinate to 5.0

p.y = 6.0  # Setting the y-coordinate to 6.0
```

Accessing values of attributes:

```
print(p.x)  # Output: 5.0

print(p.y)  # Output: 6.0
```

Passing Attributes as Arguments

Attributes can be passed to functions:

```
def display_point(point):
    print(f"Point({point.x}, {point.y})")

display_point(p)  # Output: Point(5.0, 6.0)
```

Inside the function, the alias point acts as an access point for the attributes of p.

Example Exercise: Distance Between Two Points

To calculate the distance between two points:

```
import math
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
def distance(p1, p2):
    return math.sqrt((p2.x - p1.x) ** 2 + (p2.y - p1.y) ** 2)
p1 = Point(3.0, 4.0)
p2 = Point(6.0, 8.0)
print(distance(p1, p2))  # Output: 5.0
```

This function calculates the distance between two points using their attributes.

3. RECTANGLES

What is a Rectangle?

A rectangle is a shape that has four sides, with opposite sides being the same length. To represent this shape in a computer program, we create a "Rectangle class."

What is a Class?

A class is like a blueprint or a recipe. It tells the program what a rectangle should look like and what information it needs (like width, height, and where it is located).

What Information Does a Rectangle Need?

To fully describe a rectangle, we need:

- Width: How wide it is.
- Height: How tall it is.
- Corner: A starting point that tells us where the rectangle is located (e.g., the bottom-left corner).

Creating a Rectangle Object

Think of a "Rectangle object" as a real rectangle you draw on paper. When you create one, you need to:

- Tell it how wide it is (width).
- Tell it how tall it is (height).
- Tell it where the bottom-left corner is (like an address for the rectangle).

What is the Corner?

The corner is another object (called a "Point") that stores two numbers:

- x: How far left or right the point is.
- y: How far up or down the point is.

How Do You Use This?

- You first create a rectangle using the class.
- Then, you set its width, height, and corner (location).
  For example:

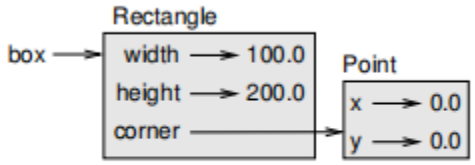

Figure 15.2: Object diagram.

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

This says:

- The rectangle is **100 units wide** and **200 units tall**.
- The bottom-left corner is at position (0, 0).

What Does Embedded Mean?
The corner (a Point object) is inside the rectangle object. It's like saying, "The rectangle owns this corner," or "The rectangle has a corner inside it."
In Simple Terms:

- A rectangle is like a picture frame.
- You describe it by saying how big it is (width and height) and where it starts (corner).
- The corner is like a little dot that tells you where the rectangle begins.
- All of this information is stored neatly in one place: the rectangle object.

4. INSTANCES AS RETURN VALUES

This says:

- The rectangle is 100 units wide and 200 units tall.
- The bottom-left corner is at position (0, 0).

What Does Embedded Mean?

The corner (a Point object) is inside the rectangle object. It's like saying, "The rectangle owns this corner," or "The rectangle has a corner inside it."

In Simple Terms:

- A rectangle is like a picture frame.
- You describe it by saying how big it is (width and height) and where it starts (corner).
- The corner is like a little dot that tells you where the rectangle begins.
- All of this information is stored neatly in one place: the rectangle object.

center_x = rectangle's left edge + (rectangle's width ÷ 2)

Next, it calculates the center's y-coordinate

center_y = rectangle's bottom edge + (rectangle's height ÷ 2)

- Finally, it puts these values into the new Point and **returns it**.

4. Example in Action

Imagine you already have a rectangle called a box. You call find_center(box) to figure out the center of the rectangle.

Here's what happens step by step:

1. The find_center function calculates the middle point of the rectangle.
    - For example, if the bottom-left corner of the rectangle is at (0, 0), the width is 100, and the height is 200:
        - Center's x = 0 + (100 ÷ 2) = 50
        - Center's y = 0 + (200 ÷ 2) = 100
2. The function creates a **Point object** with these coordinates (50, 100).
3. The function gives you the new **Point object** as the result.

5. What Do You Do with the Result?

When the function gives you the center point, you can save it in a variable, like this:

center = find_center(box)

Now, the center contains the Point object for (50, 100).

If you print it, it might look like this:

print_point(center) # Output: (50, 100)

In Simple Terms:

- find_center is a helper that figures out the middle of a rectangle.
- It takes in the rectangle, does some math, and gives you back a "Point" that tells you where the center is.
- You can then use this center point for anything you need, like drawing or checking positions.

5. OBJECTS ARE MUTABLE

1. Objects Can Be Changed

When we say "objects are mutable," it means you can change their attributes after you create them. For example:

- A rectangle has attributes like `width`, `height`, and `corner` (the position of its bottom-left corner).
- If you want to resize the rectangle, you can simply update its `width` or `height` by assigning new values.

2. Changing the Size of a Rectangle

Here's how you might resize a rectangle:

```
# Original size of the rectangle

box.width = 150.0

box.height = 200.0

# Add to the width and height

box.width += 50.0  # New width = 150.0 + 50.0 = 200.0

box.height += 100.0  # New height = 200.0 + 100.0 = 300.0
```

3. Writing a Function to Resize a Rectangle

Sometimes, instead of changing the rectangle directly, you might want to use a **function** to do it. Here's an example:

Function: grow_rectangle

This function takes:

- A rectangle (rect).
- Two numbers: dwidth (change in width) and dheight (change in height).

It adds these numbers to the rectangle's width and height.

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

Example:

```
# Example rectangle
box.width = 150.0
box.height = 200.0

# Call the function to grow the rectangle
grow_rectangle(box, 50.0, 100.0)

# Now the rectangle is bigger!
print(box.width, box.height)  # Output: 200.0, 300.0
```

4. Moving the Rectangle

Instead of changing the size, you might want to move the rectangle to a new location.
To do this, you change the coordinates of its bottom-left corner.

Function: move_rectangle

This function takes:

- A rectangle (rect).
- Two numbers: dx (change in x-coordinate) and dy (change in y-coordinate).

It adds these numbers to the rectangle's corner.x and corner.y

```
def move_rectangle(rect, dx, dy):
    rect.corner.x += dx
    rect.corner.y += dy
```

Example:

```
# Original position of the rectangle
box.corner.x = 0.0
box.corner.y = 0.0

# Move the rectangle by (50, 100)
move_rectangle(box, 50.0, 100.0)

# Now the rectangle has moved!
print(box.corner.x, box.corner.y)  # Output: 50.0, 100.0
```

5. What's Happening Behind the Scenes?

When you pass the rectangle object to a function, the function doesn't make a copy of it. Instead, it works with the original rectangle. So any changes inside the function will affect the original object.

Conclusion

- You can change parts of an object (like resizing or moving a rectangle) by updating its attributes.
- Use functions to make these changes easier and more reusable.
- Example functions:
  - grow_rectangle: Changes the size of the rectangle.
  - move_rectangle: Moves the rectangle to a new position.
- When the function modifies the rectangle, the changes are permanent because objects are mutable!

6. COPYING

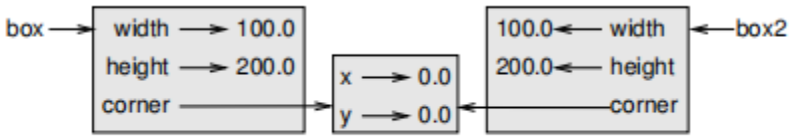

Figure 15.3: Object diagram.

What is Aliasing?Aliasing happens when **two or more variables point to the same object** in memory. If you change the object using one variable, the change is visible to the other variables too.

Example:

```
p1 = Point()
p1.x = 3.0
p1.y = 4.0

p2 = p1  # Aliasing: p2 points to the same object as p1
p2.x = 5.0  # Changing p2 also changes p1!

print(p1.x)  # Output: 5.0
```

Here, both `p1` and `p2` refer to the **same Point object**. Any change made through one variable affects the other.

2. What is Copying? Copying means creating a **new object** that has the same data as the original object. Now, if you change one object, it **won't affect** the other because they are completely separate.

Example with Copying:

```
import copy
```

```
p1 = Point()
p1.x = 3.0
p1.y = 4.0
p2 = copy.copy(p1)  # Copying: p2 is a new Point object
p2.x = 5.0  # Changing p2 does NOT affect p1
print(p1.x)  # Output: 3.0
print(p2.x)  # Output: 5.0
```

Key Points to Remember:

1. **Aliasing**: Two variables point to the same object (changes affect both).
2. **Copying**: Creates a new, independent object (changes affect only one).
3. Use the copy module to create copies and avoid aliasing problems.

This way, your program becomes easier to understand and debug because objects won't unexpectedly change due to aliasing.

1. Aliasing Problem

- **Aliasing** happens when two variables point to the **same object**.
- If one variable makes changes to the object, the other variable also "sees" those changes because they refer to the same thing.

```
r1 = Rectangle()
r2 = r1  # Both r1 and r2 point to the same rectangle
r2.width = 50  # Change the width using r2
print(r1.width)  # r1 also shows width as 50 because r1 and r2 refer to the same object
```

2. Shallow CopyA **shallow copy** creates a **new object**, but the embedded (nested) objects inside it are still shared.

Example of a shallow copy:

```
import copy

# Create a Rectangle with a nested Point for its corner
r1 = Rectangle()
r1.corner = Point()
r1.corner.x = 10
r1.corner.y = 20
# Create a shallow copy
r2 = copy.copy(r1)
# r1 and r2 are different rectangles
print(r1 is r2)  # False
# But their corners are the same object!
print(r1.corner is r2.corner)  # True
# If we change the corner of r2, it also affects r1
r2.corner.x = 30
print(r1.corner.x)  # Output: 30 (unexpected if you didn't know about shallow copying)
```

3. Deep Copy

A **deep copy** creates a completely new object, including copies of all the embedded (nested) objects inside it. The new object and the original are completely independent.

Example of a deep copy:

```
import copy

# Create a Rectangle with a nested Point for its corner
r1 = Rectangle()
r1.corner = Point()
r1.corner.x = 10
r1.corner.y = 20
# Create a deep copy
r2 = copy.deepcopy(r1)
# r1 and r2 are different rectangles
print(r1 is r2)  # False
# Their corners are also different objects
print(r1.corner is r2.corner)  # False
# If we change the corner of r2, it does NOT affect r1
r2.corner.x = 30
print(r1.corner.x)  # Output: 10 (r1 is unchanged)
```

4. Why Use Deep Copy?

Deep copying ensures that the original object and its copy are **completely independent**. This prevents bugs where changes to one object accidentally affect another.

Summary

- **Aliasing** makes objects share changes, which can cause unexpected effects.
- **Shallow copy** creates a new object but shares nested objects, leading to potential issues.
- **Deep copy** creates a completely independent copy of an object, including all its nested parts.
- A **modified** **move_rectangle** function uses deep copy to ensure the original object remains unchanged while creating a new rectangle with the desired changes.

7. DEBUGGING

1. Attribute Errors

If you try to access something (like a property or attribute) that doesn't exist in an object, Python will throw an error called AttributeError.

```
class Point:
    pass
# Create a Point object
p = Point()
# Add some attributes
p.x = 3
p.y = 5
```

```
# Try to access an attribute that doesn't exist
print(p.z)  # This will raise: AttributeError: 'Point' object has no attribute 'z'
```

What happened?

- Python says, "Hey, there's no z attribute in this Point object."

2. Checking the Type of an Object

If you're not sure what type of object you're working with, you can use the type() function.

Example:

```
print(type(p))  # Output: <class '__main__.Point'>
```

This tells you that p is an instance of the Point class.

3. Checking if an Object Belongs to a Class

You can check whether an object belongs to a specific class using the isinstance() function.

Example:

```
print(isinstance(p, Point))  # Output: True
```

This confirms that p is an instance of the Point class.

4. Checking if an Object Has a Specific Attribute

If you're not sure whether an object has a particular attribute, you can use the hasattr() function.

Example:

```
print(hasattr(p, 'x'))  # Output: True (because we added p.x = 3 earlier)

print(hasattr(p, 'z'))  # Output: False (because there's no attribute 'z')
```

- The first argument is the object (like `p`).
- The second argument is the name of the attribute you're checking, written as a string (like `'x'` or `'z'`).

5. Using try to Handle Missing Attributes

Instead of letting your program crash when an attribute is missing, you can use a try block to handle it gracefully.

Example:

```
try:
    value = p.z  # Try to access the 'z' attribute
except AttributeError:
    value = 0  # If it doesn't exist, assign a default value
print(value)  # Output: 0
```

What's happening here?

- Python tries to get p.z.
- If it doesn't exist, Python catches the error and runs the code inside the except block.

Why Use These Debugging Tools?

1. **Avoid Crashes:** If you try to access an attribute that doesn't exist, your program can crash. Using tools like hasattr() or try can prevent this.
2. **Make Your Code Flexible:** You can write functions that work with different types of objects, even if they don't always have the same attributes.
3. **Understand Objects:** Debugging tools like type() and isinstance() help you understand what kind of object you're working with.

Conclusion

- If you try to access a missing attribute, Python raises an AttributeError.
- Use type() to find out what type an object is.
- Use isinstance() to check if an object belongs to a specific class.
- Use hasattr() to check if an object has a specific attribute.
- Use a try block to handle missing attributes without crashing your program.

These tools make debugging easier and help you write more flexible and error-proof code!

CHAPTER 16

1. TIME

What is the Time Class? The Time class is just a way to represent a specific time of day (like 11:59:30 for 11 hours, 59 minutes, and 30 seconds). You can think of it as a container to store hours, minutes, and seconds as attributes.

Defining the Time Class Here's what the class looks like:

```
class Time:
    """Represents the time of day."""
    # Attributes: hour, minute, second
```

We can create a `Time` object and assign values to its attributes:

```
time = Time()  # Create a Time object
time.hour = 11
```

time.minute = 59
time.second = 30

At this point, the Time object contains:

- time.hour = 11
- time.minute = 59
- time.second = 30

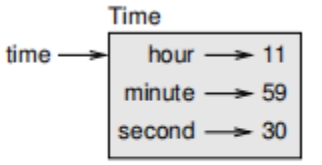

Figure 16.1: Object diagram.

-

Function 1: Printing the Time

We'll write a function called print_time to display the time in the format hour:minute:second. We'll also make sure the numbers always have two digits using the format "%02d".

Here's the function:

```
def print_time(time):
    """Prints the time in hour:minute:second format."""
    print(f"{time.hour:02d}:{time.minute:02d}:{time.second:02d}")
```

How it works:

`02d` ensures the numbers are always two digits (e.g., `01` instead of `1`).

Example:

```
time = Time()
time.hour = 9
time.minute = 5
time.second = 3
print_time(time)  # Output: 09:05:03
```

Function 2: Checking if One Time is After Another

We'll write a boolean function called is_after to check if one time comes after another time.

The Idea:

- Compare the hour, minute, and second of two Time objects.

- Instead of using an if statement, we use a **single return statement** to keep it simple.

Here's the function:

```
def is_after(t1, t2):
    """Returns True if t1 is after t2 chronologically."""
    return (t1.hour, t1.minute, t1.second) > (t2.hour, t2.minute, t2.second)
```

How it works:

- (t1.hour, t1.minute, t1.second) creates a tuple for t1.
- (t2.hour, t2.minute, t2.second) creates a tuple for t2.
- Python compares these tuples from left to right: first hour, then minute, and finally second.

Example:

```
t1 = Time()
t1.hour = 14
t1.minute = 30
t1.second = 0

t2 = Time()
t2.hour = 13
t2.minute = 45
t2.second = 15

print(is_after(t1, t2))  # Output: True (because 14:30:00 is after 13:45:15)
```

Summary

1. **Created a Time class** to represent a time of day.
2. **Wrote a print_time function** to display the time in hour:minute:second format.
3. **Wrote an is_after function** to check if one time is later than another.

2. PURE FUNCTIONS

A pure function is a function that:

- Always produces the same output if given the same input.
- Does not change or modify anything outside of the function (like variables, objects, etc.).

In the example you shared, the function takes two time values (like hours, minutes, and seconds) and combines them to return a new time object without changing the original times.

2. Prototype and Patch: This is a way of solving problems by starting with a simple version of a solution (the prototype) and then gradually improving it (patching) to handle more complex situations.

In your example:

- Prototype: The first version of the function just adds the times together but doesn't account for when minutes or seconds go over 60.
- Patch: The improved version fixes this by making sure that if the seconds go over 60, they get carried over into minutes, and if the minutes go over 60, they get carried over into hours.

3. Modifiers: Modifiers are functions that change something about an object, unlike pure functions which just create new things without changing anything.

In the example, the function that adds the time values is improved (patched) to handle cases where the total seconds or minutes exceed 60. This new version is a modifier because it adjusts the time values to make sure they are in a proper format.

So, in simple terms:

- A pure function just calculates and returns something, without changing anything else.
- You can start with a simple solution (the prototype) and make it better by adding fixes (patching).
- A modifier changes something that already exists, like fixing the time format.

3. MODIFIERS

A modifier is a function that changes something that it gets as input. When it changes an object (like a Time object), those changes are visible to the rest of the program. This is different from a pure function, which does not change anything outside itself.

2. How does the Modifier Work?:

This function works by:

- Taking a Time object and a number of seconds.
- Adding those seconds to the time.
- Modifying the original Time object directly (this is the "modifier" behavior).

For example, if you add 30 seconds to a time of 1 hour and 20 minutes, the function will change the time directly to reflect the new value (1 hour, 20 minutes + 30 seconds).

3. Problem with Seconds Over 60:

The issue comes up when the number of seconds to add exceeds 60. For example, if you add 70 seconds, the function needs to "carry over" 60 seconds into the minutes column. This isn't automatically done unless you fix it in the function.

4. Fixing the Function:

To fix the function without using a loop (which would keep checking and fixing until the seconds are less than 60), you can do it step-by-step:

- If seconds go over 60, increase the minutes and subtract 60 from the seconds.
- If minutes go over 60, increase the hours and subtract 60 from the minutes.

The goal here is to adjust the time properly without repeating the process too many times.

5. Pure Function Version:

Now, a pure function would do the same thing, but instead of modifying the original Time object, it would create a new Time object with the updated values and return that new object.

This means that:

- The original Time object stays unchanged.
- The function returns a new object with the updated time.

For example, if you have a Time of 1 hour and 20 minutes and add 30 seconds, the function would return a new Time object with the updated time (but it wouldn't modify the original one).

6. Why Use Pure Functions?:

Pure functions are often safer and easier to understand because they don't change anything outside the function. They can also make debugging and testing easier because you always know that calling the function with the same inputs will give you the same result every time.

Conclusion

- Modifiers: These functions change the objects they work on directly.
- Pure functions: These don't change anything outside the function. They just return new objects with the result.
- In this case, the goal is to write a function that adds seconds to a time, and the modifier will change the original time, while the pure function will return a new time without changing the original one.

4. PROTOTYPING VERSUS PLANNING

- Prototyping means quickly building a basic version of your function or solution (called the prototype) and then fixing any problems as you go along. It's like building something fast and fixing issues later.
- Planning means thinking ahead and understanding the problem deeply before you start coding. This helps you come up with a cleaner, more efficient solution right away.

2. Prototype and Patch Approach:

In the prototype and patch approach:

- You start with a simple solution (the prototype) that does a basic job, like adding times together.
- Then, you test it, find mistakes, and fix them along the way (this is the patching part).
- The problem is that this can lead to complicated code as you handle special cases, and sometimes you might not catch every possible error.

3. Base 60 Insight:

In this case, the time object (with hours, minutes, and seconds) can be thought of like a base 60 number.

- The seconds are like the "ones" place, minutes are like the "sixties" place, and hours are like the "thirty-six hundredths" place (because 60 * 60 = 3600).
- This idea helps simplify things because instead of thinking in terms of hours, minutes, and seconds, you can think of time like a base 60 number. This way, the rules of math (like adding numbers) apply directly to time.

4. Converting Time to Integer:

- One way to work with time more easily is to convert time (with hours, minutes, and seconds) into a single number (an integer).
- For example:
    - 1 hour 20 minutes and 30 seconds could be converted into a single number like 4800 seconds.
    - Then, you can do math (like adding or subtracting) on these numbers more easily.

5. Converting Integer Back to Time:

Once you've done the math on the integer, you can convert it back into hours, minutes, and seconds. This is like switching between normal time and a single number that's easier to work with**.**

6. Why This Helps:

- By converting time into a single number, you avoid the need to handle lots of tricky cases, like carrying over when minutes or seconds exceed 60.
- You can use simple math, which is easier to write, test, and debug.
- You can also handle new features, like subtracting times, in a simpler way.

7. Final Solution:

Once you've written the functions to convert time to an integer and back, your code becomes:

- Shorter.
- Easier to understand.
- Easier to fix errors.
- Easier to add new features (like subtracting two times).

8. Making Things Harder Can Make Them Easier:

- Sometimes, solving a problem in a more general way (like thinking of time as base 60) can actually simplify things in the long run.
- Instead of dealing with lots of special cases (like when minutes or seconds are over 60), you can just work with the time as a single number and let the computer do the hard work for you.

Conclusion

- Prototype and patch: You make a basic solution, then fix mistakes as you go.
- Base 60 approach: Time is like a number in base 60 (like how you have units for hours, minutes, and seconds).
- Converting to integers: You turn time into a number so math is easier, then convert it back.
- This approach helps make the code simpler, faster to fix, and more reliable

5. OBJECT-ORIENTED FEATURES

What is Object-Oriented Programming (OOP)? Object-oriented programming (OOP) is a way of writing programs that focuses on objects. In OOP:

- Programs use classes and methods.
- Instead of just doing operations with numbers or functions, OOP focuses on objects, which are things that represent something in the real world (like a Time object or a Rectangle object).
- Methods are actions or behaviors that these objects can do. For example, a Rectangle object might have methods to calculate its area or its perimeter.

2. How Does Python Support OOP?Python is a programming language that supports OOP. This means Python provides tools that help you create classes and define methods that are associated with those classes.

3. What is a Class and a Method?

- A class is like a blueprint for creating objects. It defines what properties (attributes) and behaviors (methods) the objects will have.
- A method is a function that belongs to a class. It defines the actions that can be performed on objects of that class.

For example:

- A Time class could define how a time object should look (with hours, minutes, and seconds), and it could have methods to add or subtract time.
- A Rectangle class could define how a rectangle looks (with width and height), and it could have methods to calculate area and perimeter.

4. Why Use Classes and Methods?

- Classes and methods make it easier to organize and manage your code. Instead of having a bunch of functions floating around, you can group them together in a class.
- Methods allow you to do things directly with an object. For example, instead of calling a function and passing a Time object every time, you can simply call a method on the Time object itself.

5. Difference Between Functions and Methods:

- A function is a standalone block of code that can perform a task (like adding two numbers). It works independently.
- A method is just like a function, but it is part of a class and is linked to an object. For example, the add_time method would be something you can call on a Time object to add more time to it.

6. How Do You Turn Functions Into Methods?

- In object-oriented programming, you can turn a function into a method by placing it inside a class definition.
- When you define methods inside a class, the method will work directly with the object's data, and you will call it using a special syntax (like my_time.add_time() instead of just calling add_time(my_time)).

7. Example:

Let's say you have a Time object, and you want to add time to it.

- Without methods: You would write a function like add_time(time_object, more_time) that takes a time object and adds more time to it.
- With methods: You would define a method inside the Time class called add_time(self, more_time) that does the same thing. Then, you would call my_time.add_time(more_time) on your time object, and it will add the time directly to that object.

Conclusion:

- Classes are like blueprints that define what objects should look like and what they can do.
- Methods are functions that are part of a class and define what an object can do.
- Using classes and methods makes your code more organized and easier to manage.

6. PRINTING OBJECTS

1. What is Printing Objects?

When we define a class in Python (for example, a Time class), we can have methods inside that class. A method is like a function, but it belongs to a class and works with objects of that class.

When we want to print out the details of an object, we often need a special function or method to tell Python how to print it. For example, if we want to print a Time object in a readable way, we need to define a method that explains how to display it.

2. Function vs Method in Python:

- A function is a block of code that does something (like printing).
- A method is a function that is part of a class and works with objects of that class.

```
def print_time(t):
    print(f"{t.hours}:{t.minutes}:{t.seconds}")
```

This function works by taking a Time object t and printing out the hours, minutes, and seconds.

Calling the Function:

To use this function, you would call it like this:

```
start = Time(12, 30, 45)  # Create a Time object
print_time(start)  # Call the function and pass the object
```

This would print: 12:30:45.

4. Turning the Function into a Method:

Now, instead of calling print_time as a standalone function, we can **turn it into a method** that belongs to the Time class. This means we move the function inside the class, and it becomes part of that class.

Here's how we define the print_time method inside the Time class:

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

    def print_time(self):
        print(f"{self.hours}:{self.minutes}:{self.seconds}")
```

How to Use the Method:

Now that print_time is a method, we can **call it using the dot notation** on a Time object:

```
start = Time(12, 30, 45)  # Create a Time object
start.print_time()  # Call the method directly on the object
```

6. Difference Between Function and Method:

- **Function Syntax (less common)**: print_time(start) — You call the function directly, passing the object as an argument.
- **Method Syntax (more common)**: start.print_time() — You call the method directly on the object.

7. Why Methods Are Useful:

In object-oriented programming, methods make it clear that the **object itself** (like start in the example) is responsible for its own behavior. So instead of saying "Hey, print_time, print this object," you say, "Hey, start (the object), print yourself."

It's like asking the object to do something instead of asking a function to do something with the object.

8. Why the Change Makes Sense:

In simple terms:

- **Before (function call)**: You are asking the function to handle the object.
- **After (method call)**: You are letting the object handle itself.

This makes the code more natural and easier to read, especially as the number of objects and behaviors grows.

Conclusion

- A **function** is a block of code that works with objects (or values), and you call it with arguments.
- A **method** is a function that belongs to a class and works with objects of that class.
- The main difference is the syntax: object.method() vs function(object).
- Using methods is more natural in object-oriented programming because the object manages itself.

7. ANOTHER EXAMPLE

What are Methods?

A method is like a function that is part of an object. When we use methods, we're working with something (an object) and telling it to do something (the method).

Example of a Method:

Imagine we have a Time object, like a clock, that tells time with hours, minutes, and seconds.

Here's how we define a method to add seconds to the clock:

```
class Time:
    def __init__(self, hours, minutes, seconds):
        self.hours = hours
        self.minutes = minutes
        self.seconds = seconds
    def add_seconds(self, seconds_to_add):
        self.seconds += seconds_to_add  # Add the seconds
        if self.seconds >= 60:  # If seconds go over 60, add to minutes
            self.minutes += self.seconds // 60  # Add extra minutes
            self.seconds = self.seconds % 60  # Keep the remaining seconds
        if self.minutes >= 60:  # If minutes go over 60, add to hours
            self.hours += self.minutes // 60  # Add extra hours
            self.minutes = self.minutes % 60  # Keep the remaining minutes
```

8. THE INIT METHOD

What is the __init__ method?

The __init__ method is like a constructor in object-oriented programming. It is automatically called when you create a new object (an instance) of a class. Think of it as the method that "sets up" the object with its initial state.

For example, if you're creating a Time object that has hours, minutes, and seconds, you want to initialize these values when the object is created.

Example: __init__ for a Time class

Let's say you have a class Time that stores the hours, minutes, and seconds. When you create a new Time object, you want to pass values for hours, minutes, and seconds to initialize the object.

```
class Time:
    def __init__(self, hours=0, minutes=0, seconds=0):
        self.hours = hours  # Store the hours in the object
        self.minutes = minutes  # Store the minutes in the object
        self.seconds = seconds  # Store the seconds in the object
```

How does the __init__ method work?

- def __init__(self, hours=0, minutes=0, seconds=0) defines the __init__ method.

- self refers to the **current object** you're creating (so you don't need to pass it when you create an object).
- hours=0, minutes=0, seconds=0 are **default parameters**. This means if you don't pass a value when you create the object, it will default to 0 for each value.
- Inside the __init__ method, self.hours = hours means the **hours** you pass in will be saved inside the object.

Creating an Object

Now, when you create a new Time object, the __init__ method will be called automatically to set up the object:

- my_time gets the default values of 0 for hours, minutes, and seconds.
- my_time2 gets the values 5 for hours, 30 for minutes, and 45 for seconds.

Why Default Parameters?

The __init__ method can have **default parameters** so that if you don't pass values, the object still gets sensible starting values. In the example above:

- If no values are given when creating the object, it will default to 0 for hours, minutes, and seconds.

Exercise Example

The exercise asks you to write an __init__ method for a Point class. The Point class has two attributes (we'll call them x and y), and you need to give the option to pass those values when creating a Point object.

Here's how you would write it:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x  # Store x in the object
        self.y = y  # Store y in the object
```

Creating Point Objects

Now, you can create Point objects like this:

```
point1 = Point()  # Default x=0, y=0
point2 = Point(3, 4)  # x=3, y=4
print(point1.x, point1.y)  # Output: 0 0
print(point2.x, point2.y)  # Output: 3 4
```

Conclusion:

- The __init__ method is automatically called when you create an object.
- It is used to **set up** the object with initial values.
- You can give **default values** so that if no argument is passed, the object still works.
- self refers to the current object you are creating and lets you store values inside it.

9. THE str_ METHOD

What is the __str__ method?

The __str__ method is a special method in Python that is used to return a string representation of an object. This string representation is usually what Python will show you when you print an object.

It's like telling Python: "Hey, if you want to print this object, here's how I want it to look!"

Example: __str__ Method for Time Class

Let's say you have a Time class and you want to make sure that when you print an object of this class, it shows the time in a nice format, like 5:30:45.

Here's how you would write the __str__ method for the Time class:

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

    # This is the __str__ method
    def __str__(self):
        return f"{self.hours}:{self.minutes}:{self.seconds}"

# Create a Time object
my_time = Time(5, 30, 45)

# Print the Time object
print(my_time)  # Output: 5:30:45
```

How it works:

1. **The __str__ method** tells Python how to convert the Time object into a string.
2. In this case, f"{self.hours}:{self.minutes}:{self.seconds}" creates a string that looks like 5:30:45 (with hours, minutes, and seconds).
3. When you call print(my_time), Python automatically uses the __str__ method to convert the object into a string and then prints it.

Why use __str__?

- **Easier debugging**: When you're printing objects during testing or debugging, having a clear representation makes it easier to understand the object's values.
- **Custom output**: You can customize how objects are displayed, making it more meaningful for the user or developer.

Exercise: __str__ for Point Class You can also use the __str__ method for other classes, like the Point class, which has x and y coordinates.

Here's an example:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
# __str__ method for Point class
    def __str__(self):
        return f"Point({self.x}, {self.y})"
# Create a Point object
point = Point(3, 4)
# Print the Point object
print(point)  # Output: Point(3, 4)
```

How it works:

- The __str__ method for the Point class converts the object to a string like "Point(3, 4)", which makes it clear that it's a point with coordinates (3, 4).
- When you print the point object, Python calls __str__ automatically and prints the output.

Conclusion:

- The __str__ method allows you to **define** how an object will look when printed.
- It's super useful for making objects easier to understand, especially when debugging.
- You can customize the string output to make it more human-friendly.

10. OPERATOR OVERLOADING

Operator overloading means that you can change how operators (like +, -, *, etc.) work with objects you create in Python. You can define special methods in your class to specify how operators should behave when used with objects of that class.

For example, you can change what happens when you add two objects of your Time class using the + operator. By defining a special method for +, Python will know what to do when you use the + operator with Time objects.

Example:

Exercise: Implement the __add__ Method for Point Class

If you have a Point class with x and y coordinates, you might want to add two Point objects together by adding their x and y coordinates.

Here's an example:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    # Overloading the + operator using __add__
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)
    # String representation of Point object
    def __str__(self):
        return f"Point({self.x}, {self.y})"
# Create two Point objects
point1 = Point(2, 3)
point2 = Point(4, 5)
# Add the two Point objects using the + operator
result = point1 + point2
# Print the result
print(result)  # Output: Point(6, 8)
```

Explanation:

- The __add__ method here adds the x and y values of two Point objects.
- So, when you write point1 + point2, it will return a new Point object where the x and y values are added together.

In Summary:

Operator overloading allows you to define how operators like +, -, *, etc., behave when applied to your objects.

You do this by defining special methods (like __add__ for +, __sub__ for -, etc.) inside your class.

This makes your code cleaner, more intuitive, and easier to work with.

Why use Operator Overloading?

Custom behavior: It allows you to change how operators work with your objects. For example, you could define what happens when two points are added, or when two Time objects are subtracted.

Cleaner code: Instead of calling a method like add_time(time1, time2), you can just use the + operator, which is more intuitive and readable.

11. TYPE-BASED DISPATCH

type-based dispatch means deciding what to do based on the type of an object or value you are working with. When you're adding two objects (or even an object and a number), Python can behave differently depending on their types.

Scenario: Adding a Time object with another Time object or an integer.

In the previous example, we showed how you can add two Time objects using the + operator, but what if we want to add an integer (like 5) to a Time object? We need to decide how to handle that.

Python doesn't know how to add an integer to a Time object by default, so we use type-based dispatch to check the type of the second operand and act accordingly.

Step-by-Step Explanation

1. __add__ Method for Time Class:
   - In the Time class, we overload the + operator using the __add__ method.
   - Inside the __add__ method, we check if the second operand (other) is a Time object or a number (like an integer).
2. Dispatching Based on Type:
   - If the second operand is another Time object, we perform the usual addition (add hours, minutes, and seconds).
   - If the second operand is an integer, we add that integer to the Time object, adjusting the time accordingly.

The __add__ method with Type-based Dispatch:

Point Class Example with + Operator:

```
class Point:
   def __init__(self, x=0, y=0):
      self.x = x
      self.y = y
   def __add__(self, other):
      if isinstance(other, Point):
         return Point(self.x + other.x, self.y + other.y)
      elif isinstance(other, tuple):
         return Point(self.x + other[0], self.y + other[1])
      return NotImplemented
  def __str__(self):
      return f"Point({self.x}, {self.y})"
```

```
# Usage
p1 = Point(1, 2)
p2 = Point(3, 4)
t = (5, 6)
print(p1 + p2)  # Point(4, 6)
print(p1 + t)   # Point(6, 8)
```

Explanation:

- **__add__ method**: Handles adding Point to Point or Point to a tuple.
- **__str__ method**: Returns the string representation of a Point.

OUTPUT:

```
Point(4, 6)
Point(6, 8)
```

12. POLYMORPHISM

Polymorphism is a concept where a function or method works with different types of data or objects, without needing to be changed or rewritten for each type.

Here's an easy way to understand it:

Example:Imagine you have a function that works with strings to count how many times each letter appears. The same function can work with other data types, like lists, tuples, or even dictionaries, as long as the elements in those data types are "compatible" (meaning they can be processed in a similar way).

Polymorphism in Action:

1. You write a function that counts elements (like letters in a word or numbers in a list).
2. Instead of writing a new function for every possible type (strings, lists, etc.), the function works for any type that can be counted or processed in a similar way.
3. This makes your code more reusable, saving time and effort.

Real-World Example:

- If you have a Time object (like hours, minutes, etc.), and you use the + operator to combine two Time objects, the + operator knows how to add Time objects together.
- But Polymorphism allows that same + operator to work with other objects as well. So if you have a list of Time objects, you can use a built-in function like sum() to add all the Time objects together, as long as they have a way to add to each other.

Why is it Useful?

- You don't need to rewrite functions for every type of object.

- Functions can work with many types of data if they "understand" how to handle them.

```
# A simple polymorphic function that works with different types

def add_elements(sequence):

    return sum(sequence)

# Works with a list of numbers:

print(add_elements([1, 2, 3]))  # Output: 6

# Works with a list of strings (concatenating them):

print(add_elements(["a", "b", "c"]))  # Output: "abc"
```

13. INTERFACE AND IMPLEMENTATION

Imagine you have a remote control to operate your TV. The buttons on the remote (the interface) allow you to do things like change the channel, adjust the volume, or turn the TV on and off. You don't need to know how the remote works inside (the implementation). You just press the buttons, and it does what you want.

In software, interface is like the buttons on the remote, and implementation is like the internal mechanics of how the remote actually works.

What does this mean in programming?

- Interface: The methods or functions that the class provides to the outside world. It's how other parts of the program interact with the class.
- Implementation: The internal details of how the class works (how the data is stored, how methods are calculated, etc.).

Why is it important to keep them separate?

1. Flexibility: If you decide to change how something works internally, but keep the same "buttons" (the interface), nothing else in the program needs to change. You can change the internal workings without affecting how other parts of the program use it.
2. Easier Maintenance: If you want to improve or fix something inside the class, you don't need to worry about breaking other parts of the program that are using the class, because those parts only rely on the interface, not the internal details.

Example: Time Class

- Interface: You might have methods like:
    - get_time() - Returns the time.

    - add_time() - Adds more time.
- Implementation: How time is stored inside the class:
    - You could store time as separate hours, minutes, and seconds.
    - Or you could store it as a single number representing seconds since midnight.

Both ways work, but as long as the interface (the methods you provide like get_time(), add_time()) stays the same, users of the class don't need to care how the time is stored or calculated.

Why does it matter?

- If you change how the time is stored (maybe you switch to storing it as seconds instead of hours, minutes, seconds), you don't need to change the methods (the interface) that other parts of the program use. The outside world keeps using the same functions (like get_time()), and your internal changes won't break anything.

In summary, separating the interface from the implementation makes your software more flexible and easier to maintain. It allows you to make changes internally without disrupting how other parts of your program work