CS2 Workbook: Object-Oriented Programming (Chapter 9)

Jeremy Evert
Southwestern Oklahoma State University

October 27, 2025

Contents

1

Classes: Introduction

1.1 Grouping Related Items into Objects

The physical world is made up of materials such as **wood**, **metal**, **plastic**, and **fabric**. To make sense of it, we group materials into higher-level concepts such as *chairs*, *tables*, and *televisions*. Similarly, in programming, we group lower-level data and functions into **objects**.

> An **object** is a bundle of data (variables) and the operations (methods) that act on that data.

Example: Thinking in Objects

Object	Operations (Methods)
Chair	`sit()`
Couch	`sit()`, `lie_down()`
Drawer	`put_item()`, `take_item()`

Objects let us think about the world in terms of *what things do*, rather than what they are made of.

Participation Discussion

- What real-world object do you interact with daily that could be modeled as a class?
- What are its attributes (data) and behaviors (methods)?

1.1.1 Programs Viewed as Objects

A program consists of variables and functions, but object-oriented programming encourages us to group related data and actions together.

Object Type	Possible Actions
Restaurant	`set_name()`, `add_cuisine()`, `add_review()`
Hotel	`set_name()`, `add_amenity()`, `add_review()`

By organizing code this way, we create programs that are easier to read, extend, and maintain.

1.2 Abstraction and Information Hiding

Abstraction occurs when we use an interface (like an oven's knob) to hide complex inner details (like heating elements).

> Objects simplify complexity by hiding details and exposing only essential operations.

- A car hides the details of its engine behind a steering wheel, pedals, and a dashboard.
- A Python object hides the details of its data, offering you methods like `.append()` or `.lower()`.

1.3 Built-in Objects in Python

Python automatically provides built-in objects, like:

- `str` — string data type (ex: `"Hello"`)
- `int` — integer data type (ex: `42`)

Example:

```
s1 = "Hello!"
print(s1.upper())   # Output: HELLO!
i1 = 130
print(i1.bit_length())  # Output: 8
```

Even built-in types are objects with data and methods!

1.4 Reflection Questions

1. What does it mean to say "a program is made of objects"?
2. Why does abstraction make code easier to understand?
3. Can you think of three real-world items that could become classes in code?

2

Classes: Grouping Data

2.1 Why Group Data into Classes?

Many variables in a program are closely related and should be bundled together. For instance, a time value consists of hours and minutes. Instead of managing two separate variables, we can define a **class** that groups them into one logical unit.

> A **class** defines a new data type that groups related data (called *attributes*) and the operations that act on them (called *methods*).

2.2 Constructing a Simple Class

The `class` Keyword

```
class ClassName:
    # Statement-1
    # Statement-2
    # ...
    # Statement-N
```

A class defines both the data and the behaviors of an object. The example below defines a class named `Time` with two attributes.

Example: Defining a Class with Two Data Attributes

```
class Time:
    """A class that represents a time of day."""
    def __init__(self):
        self.hours = 0
        self.minutes = 0
```

Here, the `__init__()` function is a special method called a **constructor**. It runs automatically when a new object (or instance) of `Time` is created.

2.3 Creating and Using an Object

```
my_time = Time()
my_time.hours = 7
my_time.minutes = 15

print(f"{my_time.hours} hours and {my_time.minutes} minutes")
```

Output:

```
7 hours and 15 minutes
```

Each variable created from the class (`my_time`) is called an **instance**. The attributes of that instance are accessed using the **dot operator** (`.`).

2.4 Multiple Instances of a Class

You can create multiple independent instances, each maintaining its own data.

```
time1 = Time()
time1.hours = 7
time1.minutes = 30

time2 = Time()
time2.hours = 12
time2.minutes = 45

print(f"{time1.hours} hours and {time1.minutes} minutes")
print(f"{time2.hours} hours and {time2.minutes} minutes")
```

Output:

```
7 hours and 30 minutes
12 hours and 45 minutes
```

2.5 Key Terms

class A grouping of related data and behaviors.

attribute A variable stored within a class or instance.

method A function that belongs to a class.

__init__ The constructor method called automatically when creating a new object.

self Refers to the instance itself within class methods.

instance An individual object created from a class.

2.6 Practice Activity

Activity 9.2.1 – Define and Instantiate a Class

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

person1 = Person()
person1.name = "Van"
print(f"This is {person1.name}")
```

Output:

```
This is Van
```

Activity 9.2.2 – Create Your Own Class

Define a class called `BookData` with three attributes: `year_published`, `title`, and `num_chapters`. Create an instance of the class and assign values to its attributes.

```
class BookData:
    def __init__(self):
        self.year_published = 0
        self.title = "Unknown"
        self.num_chapters = 0

my_book = BookData()
my_book.year_published = 2001
my_book.title = "A Tale of Two Cities"
my_book.num_chapters = 45

print(f"{my_book.title} ({my_book.year_published}) has {my_book.num_chapters} chapters."
```

Output:

```
A Tale of Two Cities (2001) has 45 chapters.
```

2.7 Reflection Questions

1. What is the difference between a class and an instance?
2. Why is the `self` keyword required in class definitions?
3. How does `__init__()` help organize data?

3

Instance Methods

3.1 What Are Instance Methods?

A **method** is a function that belongs to a class. An **instance method** operates on a specific object created from that class.

Each method must include the special first parameter `self`, which refers to the current instance of the class.

3.2 Example: Adding a Method to a Class

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

    def print_time(self):
        print(f"Hours: {self.hours}", end=" ")
        print(f"Minutes: {self.minutes}")

time1 = Time()
time1.hours = 7
time1.minutes = 15
time1.print_time()
```

Output:

```
Hours: 7 Minutes: 15
```

—

3.3 Understanding `self`

The first parameter `self` provides a reference to the instance itself. When a method is called using dot notation, like `time1.print_time()`, Python automatically passes the instance

(`time1`) as the first argument.

—

3.4 Adding Behavior to a Class

You can add more methods to model real behavior. The example below shows an `Employee` class with a method that calculates pay.

```
class Employee:
    def __init__(self):
        self.wage = 0
        self.hours_worked = 0

    def calculate_pay(self):
        return self.wage * self.hours_worked

alice = Employee()
alice.wage = 9.25
alice.hours_worked = 35
print(f"Alice's Net Pay: ${alice.calculate_pay():.2f}")
```

Output:

```
Alice's Net Pay: $323.75
```

—

3.5 Common Mistake: Forgetting `self`

If you forget to include `self` as the first parameter of a method, Python will raise an error:

```
class Employee:
    def __init__(self):
        self.wage = 0
        self.hours_worked = 0

    def calculate_pay():
        return self.wage * self.hours_worked

alice = Employee()
alice.wage = 9.25
alice.hours_worked = 35
print(alice.calculate_pay())
```

Error:

```
TypeError: calculate_pay() takes 0 positional arguments but 1 was given
```

—

3.6 Practice: Define and Use a Method

Example Activity 9.3.1 – Adding a Method

```
class Person:
    def __init__(self):
        self.first_name = ""

    def print_name(self):
        print(f"He is {self.first_name}")

person1 = Person()
person1.first_name = "Bob"
person1.print_name()
```

Output:

```
He is Bob
```

—

3.7 Challenge: Seat Class with Instance Method

```
class Seat:
    def __init__(self):
        self.row = 0
        self.col = 0

    def print_attributes(self):
        print(f"Row: {self.row}, Column: {self.col}")

seat1 = Seat()
seat1.row = 3
seat1.col = 5
seat1.print_attributes()
```

Output:

```
Row: 3, Column: 5
```

4

Class and Instance Object Types

4.1 9.4.1 Understanding Class vs. Instance Objects

A class in Python acts as a **factory** that creates **instance objects**. Each instance has its own data (attributes), but shares the same methods defined in the class.

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

time1 = Time()
time2 = Time()
time1.hours = 5
time2.hours = 7
```

—

4.2 9.4.2 Class Attributes vs. Instance Attributes

A **class attribute** is shared by all instances, while an **instance attribute** is unique to each object.

```
class MarathonRunner:
    race_distance = 42.195  # Class attribute

    def __init__(self):
        self.speed = 0      # Instance attribute

runner1 = MarathonRunner()
runner2 = MarathonRunner()
runner1.speed = 7.5
runner2.speed = 3.0
print(f"Runner1: {runner1.speed}, Runner2: {runner2.speed}")
```

—

4.3 9.4.3 Key Concept Summary

- **Class Object:** A template that defines data and behavior.
- **Instance Object:** A unique copy created from the class.
- **Class Attribute:** Shared by all instances.
- **Instance Attribute:** Belongs only to one instance.

—

4.4 9.4.4 Practice Activity

Modify the code below to add a method `print_attributes()` that prints both the class attribute and the instance attribute values.

```
class PhoneNumber:
    area_code = "405"

    def __init__(self):
        self.number = "555-1234"

# TODO: Add print_attributes method
```

What happens if you change the class attribute after creating multiple instances?

5

Class Example: Seat Reservation System

5.1 9.5 Class Example – Airline Seat Reservation System

A **class** can represent a real-world entity that manages its own data and actions. The example below models a simple airline seat reservation system. Each `Seat` object stores a passenger's name and the amount paid.

```
class Seat:
    def __init__(self):
        self.first_name = ""
        self.last_name = ""
        self.paid = 0.0

    def reserve(self, f_name, l_name, amt_paid):
        self.first_name = f_name
        self.last_name = l_name
        self.paid = amt_paid

    def make_empty(self):
        self.first_name = ""
        self.last_name = ""
        self.paid = 0.0

    def is_empty(self):
        return self.first_name == ""

    def print_seat(self):
        print(f"{self.first_name} {self.last_name}, Paid: {self.paid:.2f}")


def make_seats_empty(seats):
    for s in seats:
        s.make_empty()
```

```
def print_seats(seats):
    for i in range(len(seats)):
        print(f"{i}:", end=" ")
        seats[i].print_seat()


num_seats = 5
available_seats = []

for i in range(num_seats):
    available_seats.append(Seat())

command = input("Enter command (p/r/q):\n")

while command != "q":
    if command == "p":  # Print seats
        print_seats(available_seats)
    elif command == "r":  # Reserve a seat
        seat_num = int(input("Enter seat num:\n"))
        if not available_seats[seat_num].is_empty():
            print("Seat not empty")
        else:
            fname = input("Enter first name:\n")
            lname = input("Enter last name:\n")
            paid = float(input("Enter amount paid:\n"))
            available_seats[seat_num].reserve(fname, lname, paid)
    else:
        print("Invalid command.")

    command = input("Enter command (p/r/q):\n")
```

Discussion

This example demonstrates:

- Encapsulation of related data and behaviors within a class.
- The use of methods to modify and access object state.
- A program structure that makes expansion (like saving or loading seats) simple.

6

Class Constructors

6.1 9.6 Overview

A class constructor is a special method that defines how new objects are created and initialized. In Python, the constructor method is named `__init__()`. Constructors are used to set up instance attributes and can accept parameters to configure each new object.

—

6.2 Adding Parameters to a Constructor

```
class RaceTime:
    def __init__(self, start_time, end_time, distance):
        """Initialize race data."""
        self.start_time = start_time    # Format: "H:MM"
        self.end_time = end_time
        self.distance = distance        # In miles

# Create RaceTime objects
time_jason = RaceTime("3:15", "7:45", 26.21875)
time_bobby = RaceTime("3:15", "6:30", 26.21875)
```

Listing 6.1: A simple constructor with parameters.

Here, each new object receives its starting time, ending time, and race distance. This is more powerful than setting everything to zero — each instance can have its own data immediately upon creation.

—

6.3 Complete RaceTime Example

```
class RaceTime:
    def __init__(self, start_hrs, start_mins, end_hrs, end_mins,
       ↪ dist):
```

```
        self.start_hrs = start_hrs
        self.start_mins = start_mins
        self.end_hrs = end_hrs
        self.end_mins = end_mins
        self.distance = dist

    def print_time(self):
        if self.end_mins >= self.start_mins:
            minutes = self.end_mins - self.start_mins
            hours = self.end_hrs - self.start_hrs
        else:
            minutes = 60 - self.start_mins + self.end_mins
            hours = self.end_hrs - self.start_hrs - 1
        print(f"Time to complete race: {hours}:{minutes:02d}")

    def print_pace(self):
        total_minutes = (self.end_hrs * 60 + self.end_mins) - \
                        (self.start_hrs * 60 + self.start_mins)
        pace = total_minutes / self.distance
        print(f"Average pace: {pace:.2f} mins/mile")

# Example interaction
distance = 5.0
start_hrs = int(input("Enter starting time hours: "))
start_mins = int(input("Enter starting time minutes: "))
end_hrs = int(input("Enter ending time hours: "))
end_mins = int(input("Enter ending time minutes: "))

race_time = RaceTime(start_hrs, start_mins, end_hrs, end_mins,
    ↪ distance)
race_time.print_time()
race_time.print_pace()
```

Listing 6.2: RaceTime class with methods.

—

6.4 Default Constructor Parameters

Constructors can also include default values for convenience. This reduces repetition and helps when typical defaults are common.

```
class Employee:
    def __init__(self, name, wage=8.25, hours=20):
        """Default employee works part-time and earns minimum wage."
            ↪ ""
        self.name = name
        self.wage = wage
```

```
        self.hours = hours

employees = [
    Employee("Todd"),                    # uses defaults
    Employee("Jason"),                   # uses defaults
    Employee("Tricia", 12.50, 40)        # manager: custom values
]

for e in employees:
    print(f"{e.name} earns ${e.wage * e.hours:.2f} per week")
```

Listing 6.3: Employee class with default parameters.

—

6.5 Constructors in Practice: Student Example

```
class Student:
    def __init__(self, name, grade=9, honors=False, athletics=False)
       ↪ :
        self.name = name
        self.grade = grade
        self.honors = honors
        self.athletics = athletics

johnny = Student("Johnny", grade=11, honors=True)
tommy = Student("Tommy")

print(f"{johnny.name}: grade {johnny.grade}, honors={johnny.honors}"
   ↪ )
print(f"{tommy.name}: grade {tommy.grade}, athletics={tommy.
   ↪ athletics}")
```

Listing 6.4: Constructor with several defaults.

—

6.6 Constructor Exercises

1. Modify `Employee` so that it tracks yearly pay in addition to hourly.
2. Add a method `is_manager()` that returns `True` if wage ¿ 12.
3. Rewrite `RaceTime` so that it accepts total minutes instead of hours and minutes separately.

7

Class Interfaces

7.1 Overview

A **class interface** defines the set of methods that a programmer uses to interact with an instance of a class. The interface describes *what* operations can be performed on an object, while the **implementation** describes *how* those operations work internally.

In this section, we will explore:

- How to design class interfaces for readability and reuse.
- How to separate interface from implementation.
- The use of internal methods to simplify class logic.

Key Idea: A well-designed class acts like a mini toolbox—you can use its tools (the methods) without worrying about how each tool is built inside.

7.2 Example: The RaceTime Class Interface

Consider the following example that models a simple race timer. The interface includes three methods:

- `__init__()` — creates a new RaceTime instance.
- `print_time()` — prints the time to complete the race.
- `print_pace()` — prints the average pace per mile.

```
class RaceTime:
    def __init__(self, start_time, end_time, distance):
        self.start_time = start_time
        self.end_time = end_time
        self.distance = distance

    def print_time(self):
```

```
        # Implementation details hidden from user
        ...

    def print_pace(self):
        # Implementation details hidden from user
        ...
```

Listing 7.1: A class interface consists of methods to interact with an instance.

Instructor Note: Emphasize that users of the class only need to know how to call `print_time()` or `print_pace()`, not how they compute their results internally.

7.3 Internal (Private) Methods

Sometimes, a class includes internal helper methods that are not meant to be accessed directly. By convention, these methods begin with an underscore (_), signaling to programmers that they are used only inside the class.

```
class RaceTime:
    def __init__(self, start_hrs, start_mins, end_hrs, end_mins,
       ↪ dist):
        self.start_hrs = start_hrs
        self.start_mins = start_mins
        self.end_hrs = end_hrs
        self.end_mins = end_mins
        self.dist = dist

    def print_time(self):
        total_time = self._diff_time()
        print(f"Time to complete race: {total_time[0]}:{total_time
           ↪ [1]:02}")

    def print_pace(self):
        total_time = self._diff_time()
        total_minutes = total_time[0]*60 + total_time[1]
        print(f"Avg pace (mins/mile): {total_minutes / self.dist:.2f
           ↪ }")

    def _diff_time(self):
        """Internal helper method"""
        if self.end_mins >= self.start_mins:
            minutes = self.end_mins - self.start_mins
            hours = self.end_hrs - self.start_hrs
        else:
            minutes = 60 - self.start_mins + self.end_mins
            hours = self.end_hrs - self.start_hrs - 1
        return (hours, minutes)
```

Listing 7.2: RaceTime class with internal helper method.

7.4 Abstract Data Types (ADTs) and Information Hiding

A class can also represent an **Abstract Data Type (ADT)**, which hides the details of how data is stored or manipulated internally. The ADT provides a public interface for the user while keeping private methods and variables hidden.

- The user sees the interface (the public methods).
- The developer maintains control of the internal logic.
- This separation prevents accidental misuse and simplifies debugging.

In Python, private members are not enforced, but the underscore convention signals to other programmers that a method is intended for internal use only.

> *"An ADT is like a vending machine—you press a button, and magic happens inside."*

7.5 Quick Review

1. The **interface** of a class defines what users can do with it.
2. The **implementation** defines how it does those things.
3. Internal methods usually begin with an underscore (_).
4. A well-designed class separates its interface from its implementation.

7.6 Participation Check

- True or False: A class interface consists of methods that a programmer should use to modify or access the class.
- True or False: Internal methods should begin with an underscore in their name.
- True or False: Internal methods cannot be called outside the class.
- True or False: A well-designed class separates its interface from its implementation.

Next Up: In Section 9.8, we'll learn about *class customization*—defining how objects behave with built-in operators like `==`, `<`, or `+`.

8

Class Customization

8.1 9.8 Class Customization

Class customization is the process of defining how an instance of a class should behave for certain operations—such as printing, comparing, or arithmetic. Python provides *special method names* (sometimes called "dunder methods") that begin and end with double underscores. By defining these, programmers can customize how built-in behaviors interact with objects.

Implementing __str__(): Custom Printing

Example 9.8.1: Implementing __str__ alters how the class is printed.

Normal Printing:

```
class Toy:
    def __init__(self, name,
       ↪ price, min_age):
        self.name = name
        self.price = price
        self.min_age = min_age

truck = Toy("Monster Truck XX",
   ↪ 14.99, 5)
print(truck)
```

Customized Printing:

```
class Toy:
    def __init__(self, name,
       ↪ price, min_age):
        self.name = name
        self.price = price
        self.min_age = min_age

    def __str__(self):
        return (f"{self.name}
           ↪ costs only ${self.
           ↪ price:.2f}. "
                f"Not for
                   ↪ children
                   ↪ under {self
                   ↪ .min_age}!"
                   ↪ )

truck = Toy("Monster Truck XX",
   ↪ 14.99, 5)
print(truck)
```

Output:

```
Monster Truck XX costs only $14.99. Not for children under 5!
```

Try It: Customizing Print Output

```
class Car:
    def __init__(self, make, model, year, miles, price):
        self.make = make
        self.model = model
        self.year = year
        self.miles = miles
        self.price = price

    # FIXME: add __str__() to format output
```

Desired output example:

```
1989 Chevrolet Blazer:
Mileage: 115000
Sticker price: $3250
```

Operator Overloading

Python also allows classes to redefine built-in operators such as `<`, `>`, `==`, and `+`. This is done through special methods such as `__lt__`, `__gt__`, `__eq__`, etc.

Example 9.8.2: Overloading the ¡ Operator

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

    def __str__(self):
        return f"{self.hours}:{self.minutes:02d}"

    def __lt__(self, other):
        if self.hours < other.hours:
            return True
        elif self.hours == other.hours:
            return self.minutes < other.minutes
        return False
```

These comparison methods are known as **rich comparison methods**.

Method	Operator
__lt__	less than (¡)
__le__	less than or equal (¡=)
__gt__	greater than (¿)
__ge__	greater than or equal (¿=)
__eq__	equal (==)
__ne__	not equal (!=)

Practice: Comparing Quarterbacks

```
class Quarterback:
    def __init__(self, yrds, tds, cmps, atts, ints, wins):
        self.wins = wins
        self.rating = (((8.4 * yrds) + (330 * tds)
                        + (100 * cmps) - (200 * ints)) / atts)

    def __lt__(self, other):
        return (self.rating < other.rating or
                self.wins < other.wins)

    def __gt__(self, other):
        return (self.rating > other.rating and
                self.wins > other.wins)
```

Challenge: Used Car Comparison

```
class UsedCar:
    def __init__(self, price, condition):
        self.price = price
        self.condition = condition  # 0=poor, 5=new

    def __lt__(self, other):
        return self.price < other.price

    def __le__(self, other):
        return self.price <= other.price

    def __gt__(self, other):
        return self.condition > other.condition

    def __ne__(self, other):
        return (self.price != other.price or
                self.condition != other.condition)
```

Defining __str__ for Custom Output

Challenge: Define __str__ for a simple CarRecord class.

```
class CarRecord:
    def __init__(self):
        self.year_made = 0
        self.car_vin = ""

    def __str__(self):
        return f"Year: {self.year_made}, VIN: {self.car_vin}"

my_car = CarRecord()
my_car.year_made = 2009
my_car.car_vin = "ABC321"
print(my_car)
```

Output:

```
Year: 2009, VIN: ABC321
```

Reflection

- How does __str__ help make debugging easier?
- Why might you overload comparison operators for a custom class?
- What risks exist when customizing too many class behaviors?

Summary:

- Class customization uses special method names (e.g., __str__, __lt__, __eq__).
- Enables clean printing, sorting, and comparison of class instances.
- Encourages readable, intuitive code design.

9

More Operator Overloading: Classes as Numeric Types

9.1 9.9 Classes as Numeric Types

Numeric operators such as `+`, `-`, `*`, and `/` can be overloaded using class customization techniques. A user-defined class can thus be treated as a numeric object in which instances of that class can be added, subtracted, or otherwise combined.

Consider the example below, which represents a 24-hour clock. The algorithm determines the absolute shortest distance between two given times. For instance, the time between 5:00 and 3:30 (same day) is shorter than between 22:30 and 2:40 (across midnight).

```
class Time24:
    def __init__(self, hours, minutes):
        self.hours = hours
        self.minutes = minutes

    def __str__(self):
        return f"{self.hours:02d}:{self.minutes:02d}"

    def __gt__(self, other):
        if self.hours > other.hours:
            return True
        elif self.hours == other.hours and self.minutes > other.
           ↪ minutes:
            return True
        return False

    def __sub__(self, other):
        """Calculate absolute distance between two times."""
        if self > other:
            larger, smaller = self, other
        else:
            larger, smaller = other, self

        hrs = larger.hours - smaller.hours
        mins = larger.minutes - smaller.minutes
```

```
        if mins < 0:
            mins += 60
            hrs -= 1
        if hrs > 12:
            hrs = 24 - hrs
        return Time24(hrs, mins)

t1 = Time24(5, 0)
t2 = Time24(3, 30)
print(f"Time difference: {t1 - t2}")    # Time difference: 01:30
```

Listing 9.1: Extending the Time class with an overloaded subtraction operator

The `__sub__()` method defines how subtraction behaves for two `Time24` instances. If `t1` and `t2` are both `Time24` objects, then the expression `t1 - t2` calls `t1.__sub__(t2)` and returns another `Time24` instance.

Using `isinstance()` for Safer Subtraction

To handle subtraction of arbitrary types, we can use the built-in `isinstance()` function. This ensures that our class behaves predictably even when mixed with other types.

```
class Time24:
    def __init__(self, hours, minutes):
        self.hours = hours
        self.minutes = minutes

    def __str__(self):
        return f"{self.hours:02d}:{self.minutes:02d}"

    def __sub__(self, other):
        # Handle subtraction by integer minutes
        if isinstance(other, int):
            return Time24(self.hours, self.minutes - other)
        elif isinstance(other, Time24):
            hrs = self.hours - other.hours
            mins = self.minutes - other.minutes
            if mins < 0:
                mins += 60
                hrs -= 1
            if hrs < 0:
                hrs += 24
            return Time24(hrs, mins)
        else:
            raise NotImplementedError(f"Unsupported operand type: {
                ↪ type(other)}")

t1 = Time24(10, 15)
print(t1 - 45)              # Subtract 45 minutes
```

```
print(t1 - Time24(8, 30))    # Time difference
```

Listing 9.2: The isinstance() check in operator overloading

Common Operator Methods

Python allows overloading of almost every arithmetic operator using special methods:

Method	Description
__add__(self, other)	Addition (+)
__sub__(self, other)	Subtraction (-)
__mul__(self, other)	Multiplication (*)
__truediv__(self, other)	Division (/)
__floordiv__(self, other)	Floor Division (//)
__mod__(self, other)	Modulus (%)
__pow__(self, other)	Exponentiation (**)
__abs__(self)	Absolute Value (abs())
__int__(self)	Convert to integer (int())
__float__(self)	Convert to float (float())

Participation Example: LooseChange Class

```
class LooseChange:
    def __init__(self, value):
        self.value = value    # total cents

    def __add__(self, other):
        new_value = self.value + other.value
        return LooseChange(new_value)

    def __float__(self):
        return self.value / 100.0

lc1 = LooseChange(135)
lc2 = LooseChange(115)
print(float(lc1 + lc2))    # 2.50
```

Listing 9.3: A simple numeric class using operator overloading

Challenge Example: Duration Addition

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

```
    def __add__(self, other):
        total_hours = self.hours + other.hours
        total_minutes = self.minutes + other.minutes
        if total_minutes >= 60:
            total_hours += 1
            total_minutes -= 60
        return Duration(total_hours, total_minutes)

first_trip = Duration(1, 35)
second_trip = Duration(0, 43)
total_trip = first_trip + second_trip
print(total_trip.hours, total_trip.minutes)  # Output: 2 18
```

Listing 9.4: Operator overloading for adding durations

9.2 Instructor Notes

- Emphasize how operator overloading enhances readability and reuse.
- Students often forget to return a new instance instead of a raw value.
- Discuss when operator overloading improves clarity versus when it hides complexity.

Reflection: Encourage students to design a class (e.g., `BankAccount`, `Vector2D`, or `Temperature`) that uses at least two overloaded operators and one type conversion (`__float__` or `__int__`).

10

Memory Allocation and Garbage Collection

10.1 9.10 Memory Allocation and Garbage Collection

Memory Allocation

The process of an application requesting and being granted memory is called **memory allocation**. When a Python program runs, it requests memory from the operating system. Python's runtime then divides that memory into sections for code, data, and dynamically created objects.

While some languages require programmers to manage memory manually, Python automatically allocates and releases memory on the programmer's behalf.

```
lst = []
for i in range(0, 100):
    lst.append(i)
print("List length:", len(lst))
```

Listing 10.1: Memory allocation in Python

When the list `lst` is created, Python requests space for it from the operating system. As more elements are appended, the interpreter automatically allocates additional blocks of memory as needed.

Key Points

- System memory is divided into segments managed by the OS.
- Python applications request memory through the interpreter, not directly.
- Other processes may occupy different regions of memory simultaneously.
- Allocation is usually invisible to the programmer.

> **Instructor Note:** Emphasize that memory allocation happens behind the scenes. The programmer writes code that creates objects—Python handles where and how they live.

Quick Check

1. The Python runtime requests memory from the operating system. **(True)**
2. Objects may reside in memory that has not been allocated. **(False)**
3. All programming languages perform memory allocation for the programmer. **(False)**

—

Garbage Collection

Memory deallocation is the act of freeing memory that stores variables or objects that are no longer needed. Python uses a managed model: when objects are no longer referenced by any variable, they become candidates for automatic removal by the **garbage collector**.

Each object in memory maintains a **reference count** (RC)—the number of variables that reference it. When an object's RC becomes 0, Python knows it can reclaim that space.

```
string1 = "Python"
string2 = "Computer science"
string3 = string2
string1 = "ZyBooks"
string2 = string1
```

Listing 10.2: Reference counting in Python

Explanation:

- Initially, `string1` references "Python".
- `string2` and `string3` reference "Computer science".
- Reassigning `string1` to "ZyBooks" makes "Python" unreachable. Its reference count drops to 0 and it can be deallocated.

  Instructor Note: Demonstrate using a small live example or a diagram to show how reference counts change as variables are reassigned.

True/False Practice

1. An object with RC = 0 can be deallocated by the garbage collector. **True**
2. Deallocation happens immediately when RC drops to 0. **False**
3. Swapping two variables may briefly create an object with RC = 0. **True**

—

Reflection

- Why does Python use automatic garbage collection?
- How does reference counting prevent memory leaks?
- What is one situation where garbage collection might not occur immediately?

Summary: Python automates both allocation and deallocation of memory. Objects are stored only as long as they are referenced, allowing developers to focus on logic rather than low-level memory management.

11

Inheritance and Class Hierarchies

11.1 9.11 Inheritance and Reuse in Object-Oriented Programming

Inheritance allows one class (the **child** or **subclass**) to reuse and extend the behavior of another class (the **parent** or **base class**). This reduces duplication and encourages code reuse—similar to how every car model inherits features like wheels, engines, and doors from the general concept of a vehicle.

Example: A Vehicle Family Tree

Imagine a `Vehicle` class that defines attributes and methods shared by all vehicles. Specific types of vehicles—such as `Car`, `Truck`, and `Motorcycle`— can inherit from it and add their own unique traits.

```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start(self):
        print(f"{self.make} {self.model} engine started.")

    def stop(self):
        print(f"{self.make} {self.model} engine stopped.")

class Car(Vehicle):
    def __init__(self, make, model, doors):
        super().__init__(make, model)
        self.doors = doors

    def open_trunk(self):
        print(f"{self.make} {self.model}'s trunk is now open.")
```

```
class Truck(Vehicle):
    def __init__(self, make, model, towing_capacity):
        super().__init__(make, model)
        self.towing_capacity = towing_capacity

    def tow(self, weight):
        print(f"Towing {weight} lbs out of {self.towing_capacity} lbs capacity.")
```

Discussion

Each subclass inherits the `start()` and `stop()` methods from `Vehicle`. They don't need to be rewritten unless behavior must change.

```
my_car = Car("Ford", "Focus", 4)
my_truck = Truck("Nissan", "Titan", 9000)

my_car.start()
my_car.open_trunk()
my_truck.start()
my_truck.tow(6000)
```

Output:

```
Ford Focus engine started.
Ford Focus's trunk is now open.
Nissan Titan engine started.
Towing 6000 lbs out of 9000 lbs capacity.
```

Using super()

The `super()` function lets the subclass call the constructor (or other methods) of its parent. Without it, the base attributes would not be initialized.

Method Overriding

A subclass can redefine (override) methods from its parent to create specialized behavior.

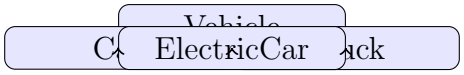
```
class ElectricCar(Car):
    def start(self):
        print(f"{self.make} {self.model} powers on silently. ")

tesla = ElectricCar("Tesla", "Model 3", 4)
tesla.start()
```

Output:

```
Tesla Model 3 powers on silently.
```

Class Hierarchy Diagram



Practice Activity 9.11.1 – Create Your Own Hierarchy

Create a base class called `Animal` with attributes `name` and `sound`. Then create subclasses `Dog` and `Cat` that each override a `make_sound()` method to print their own noises.

```
class Animal:
    def __init__(self, name):
        self.name = name
    def make_sound(self):
        print("Some generic sound")

class Dog(Animal):
    def make_sound(self):
        print(f"{self.name} says Woof!")

class Cat(Animal):
    def make_sound(self):
        print(f"{self.name} says Meow!")
```

Reflection:

- What benefits does inheritance provide for large codebases?
- When might it be better to avoid inheritance and use composition instead?
- Can a class inherit from more than one parent in Python?

Summary

- Inheritance enables code reuse and logical hierarchies.
- `super()` connects child classes to parent initialization.
- Method overriding allows specialization.
- Real-world analogy: All vehicles share traits (wheels, engines), but each subclass adds its own personality.

"Every subclass stands on the shoulders of its superclass."