

# 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	<code>sit()</code>
Couch	<code>sit()</code> , <code>lie_down()</code>
Drawer	<code>put_item()</code> , <code>take_item()</code>

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	<code>set_name()</code> , <code>add_cuisine()</code> , <code>add_review()</code>
Hotel	<code>set_name()</code> , <code>add_amenity()</code> , <code>add_review()</code>

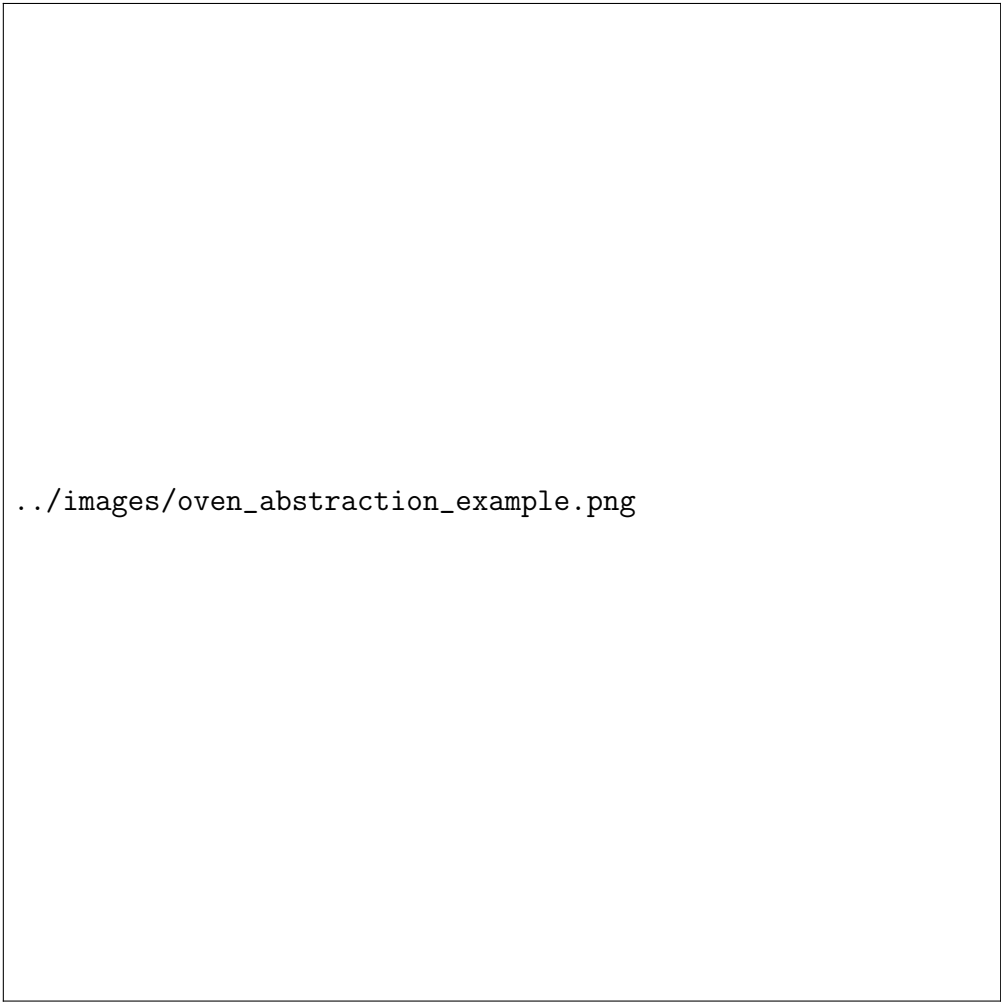
*By organizing code this way, we create programs that are easier to read, extend, and maintain.*

## 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.

### Example:



`../images/oven_abstraction_example.png`

- 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!*

## 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.

## 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?

## Visual Summary



`../images/grouping_data_into_classes.png`

# 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

## Visual Summary

images/oven\_abstraction\_example.png



# 4

---

## Class and Instance Object Types

---

### 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
```


### 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}")
```



images/class\_and\_instance\_objects.png

---

### 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.

---

### 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

---

### 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.

## Visual Summary



`../images/seat_reservation_diagram.png`





# 6

---

## Class Constructors

---

### 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.

### 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.

### 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.

## 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.

## 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.

## 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  $\geq 12$ .
3. Rewrite `RaceTime` so that it accepts total minutes instead of hours and minutes separately.