PROGRAMMING PRINCIPLES

WEEK 10 WORKSHOP

Mission Control v10.0

Object-Oriented Programming: Classes, Objects, Methods, Encapsulation and Inheritance

Starting point: Mission Control v9.0 from Week 9. The Week 9 program already uses functions, tuples, lists, sets, dictionaries, comprehensions, references, files, exceptions and a command loop.

Week 10 goal: Refactor the program so that Mission Control and individual sensor readings become objects. We will keep the useful Week 9 functionality while introducing classes, attributes, methods, __init__, self, class attributes, __str__, object relationships and a clean object-oriented program structure.

Important: Do not try to rewrite the entire program at once. This chapter deliberately builds the Week 10 version in small stages so that every change has a clear purpose.

Workshop Overview

In Week 9, Mission Control was organised mainly as a collection of functions operating on a list of tuple readings. That design works, but the program now has a natural opportunity for object-oriented design.

Week 9 representation	Week 10 representation	Why it helps
(sensor, temperature, status)	SensorReading object	One reading becomes a meaningful object with its own data and behaviour.
reading_history list	MissionControl.reading_history	Mission state belongs to the MissionControl object.
Functions receive reading_history	MissionControl methods use self	Related data and operations are grouped together.
Tuple indexes reading[0], reading[1], reading[2]	reading.sensor_number, reading.temperature, reading.status	Code becomes easier to read.
format_reading(reading)	SensorReading.__str__()	Printing a reading becomes natural.
Global mission_name/destination	MissionControl attributes	Mission information belongs to the mission object.

Big idea: Week 10 is not about adding classes simply because classes are new. The goal is to decide what data and behaviour naturally belong together, then place them in classes.

Learning Outcomes

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

- Explain why SensorReading is a good candidate for a class.
- Create a class with an __init__ method.
- Explain what self means in an instance method.
- Create and use multiple SensorReading objects.
- Replace tuple indexing with named object attributes.
- Use __str__ to define a useful string representation.
- Use a class attribute for values shared by all SensorReading objects.
- Create a MissionControl object that owns mission state.
- Use methods to operate on the MissionControl object's state.
- Recognise the association between a MissionControl object and its SensorReading objects.
- Preserve Week 9 sets, dictionaries, comprehensions and file handling inside the new OOP structure.
- Use the final program to practise inheritance concepts through an optional extension.

Before You Start

- Keep your working Week 9 mission_control_v9.py available as a reference.
- Create a new file named mission_control_v10.py. Do not overwrite your Week 9 version.
- Use the exact Week 10 final code in the appendix as your reference implementation.
- Run the program after each major stage rather than waiting until the very end.
- Keep mission_history.txt and mission_report.txt in the same folder as the Python program.

Part A: Understand the OOP Refactor

Look at the Week 9 Reading Representation

In Week 9, one sensor reading was represented as a tuple containing three values: sensor number, temperature and status.

```
reading = (2, 85.0, "WARNING")

print(reading[0])
print(reading[1])
print(reading[2])
```

This works, but reading[0], reading[1] and reading[2] do not tell a reader what each value means.

> **Think about it:** If you return to the code several weeks later, reading.temperature is much easier to understand than reading[1].

Design a SensorReading Class

Ask what a sensor reading knows and what it can do.

Question	Answer
What does a reading know?	sensor number, temperature and status
What can a reading tell us?	whether it is SAFE, WARNING or CRITICAL
How should it display itself?	in the familiar Mission Control reading format

This gives us a natural class design:

```
class SensorReading:
    def __init__(self, sensor_number, temperature, status):
        self.sensor_number = sensor_number
        self.temperature = temperature
        self.status = status
```

The three assignments create three instance attributes. Every SensorReading object gets its own values.

Create SensorReading Objects

```
reading1 = SensorReading(1, 25.0, "SAFE")
reading2 = SensorReading(2, 85.0, "WARNING")
reading3 = SensorReading(3, 105.0, "CRITICAL")
```

Object	sensor_number	temperature	status
reading1	1	25.0	SAFE
reading2	2	85.0	WARNING
reading3	3	105.0	CRITICAL

> **Key insight:** The class is the definition. reading1, reading2 and reading3 are three separate objects. Their attributes are independent.

Part B: Add Behaviour to SensorReading

Add Methods

A class should not only store data. It can also provide operations related to that data.

```
class SensorReading:
    def __init__(self, sensor_number, temperature, status):
        self.sensor_number = sensor_number
        self.temperature = temperature
        self.status = status

    def is_safe(self):
        return self.status == "SAFE"

    def is_warning(self):
        return self.status == "WARNING"

    def is_critical(self):
        return self.status == "CRITICAL"
```

Now the code that uses a reading does not need to repeatedly compare the status string.

```
if reading.is_critical():
    print("CRITICAL - scan stopped")
```

Why this is useful: The object knows about its own state. Code using the object can ask a clear question: is this reading critical?

Understand self

Inside reading.is_critical(), self refers to the particular SensorReading object used for the call.

```
reading1 = SensorReading(1, 25.0, "SAFE")
reading2 = SensorReading(2, 105.0, "CRITICAL")

print(reading1.is_critical())
print(reading2.is_critical())
```

Output:

```
False
True
```

Trace it: During reading1.is_critical(), self is reading1. During reading2.is_critical(), self is reading2.

Add a Class Attribute

The Week 10 material distinguishes class members from instance members. A class attribute is useful when the same value is shared by all objects.

```
class SensorReading:
    VALID_STATUSES = {"SAFE", "WARNING", "CRITICAL"}

    def __init__(self, sensor_number, temperature, status):
        self.sensor_number = sensor_number
        self.temperature = temperature
        self.status = status
```

Why a class attribute? Every reading uses the same set of valid status names. There is no need to create a separate copy of that definition as an instance attribute for every reading.

Add __str__

Week 10 introduces special methods. __str__ controls the useful human-readable representation of an object.

```
def __str__(self):
    return (
        f"Sensor {self.sensor_number:<2} | "
        f"{self.temperature:>6.1f} C | "
        f"{self.status:<8}"
    )
```

Now:

```
print(reading1)
```

can directly produce the familiar Mission Control format.

Important: __str__ must return a string. It should not call print() itself.

Part C: Create the MissionControl Class

Identify the Second Object

The program itself also has state: mission name, destination and reading history. Instead of keeping those as separate global variables, create a MissionControl object to own them.

```
class MissionControl:
    def __init__(self, mission_name, destination):
        self.mission_name = mission_name
        self.destination = destination
        self.reading_history = []
```

Attribute	What it stores
self.mission_name	The mission name
self.destination	The mission destination
self.reading_history	A list of SensorReading objects

Important design change: reading_history is now an attribute of the MissionControl object. This is a major Week 10 improvement: mission state and mission behaviour are grouped together.

Store Objects Inside the History List

```
mission = MissionControl("Apollo", "Moon")

reading1 = SensorReading(1, 25.0, "SAFE")
reading2 = SensorReading(2, 85.0, "WARNING")

mission.reading_history.append(reading1)
mission.reading_history.append(reading2)
```

This is an example of association: the MissionControl object has a list containing SensorReading objects.

Picture: MissionControl → reading_history → SensorReading objects. The mission owns the collection; each reading owns its own sensor data.

Add Small Methods for Managing Readings

```
def add_reading(self, reading):
    self.reading_history.append(reading)

def add_readings(self, readings):
    self.reading_history.extend(readings)

def clear_history(self):
    self.reading_history.clear()
```

Now the rest of the program does not need to manipulate the history list directly every time.

```
mission.add_reading(reading1)
mission.add_readings([reading2])
```

Connection to Week 9: The list and its methods are still being used. Week 10 changes where the list belongs and how the program interacts with it.

Part D: Move Existing Functions into the Class

Move Temperature Classification

In Week 9, classify_temperature() was a standalone function. In Week 10 it becomes a MissionControl method because temperature classification is part of the mission's sensor-processing behaviour.

```
def classify_temperature(self, temperature):
    if temperature < 0:
        return "INVALID"
    elif temperature >= 100:
        return "CRITICAL"
    elif temperature > 80:
        return "WARNING"
    else:
        return "SAFE"
```

The main difference is self. The method can now be called as:

```
status = mission.classify_temperature(85)
```

Notice: The temperature is still supplied as an argument. self identifies which MissionControl object is performing the operation.

Refactor scan_sensors()

The scan process now creates SensorReading objects instead of tuples.

```
reading = SensorReading(
    sensor_number, temperature, status
)
scan_readings.append(reading)
```

The rest of the logic can ask the object about itself:

```
if reading.is_critical():
    print("CRITICAL - scan stopped")
    warning_count += 1
    break
elif reading.is_warning():
    print("WARNING")
    warning_count += 1
else:
    print("SAFE")
    safe_count += 1
```

Why this is clearer: The Week 9 version had to inspect reading[2]. The Week 10 version asks the reading object directly.

Refactor File Loading

The history file still contains simple text such as 1|25.0|SAFE. The difference is what happens after the line is parsed: the program constructs a SensorReading object.

```
reading = SensorReading(
    sensor_number, temperature, status
)
self.reading_history.append(reading)
```

This demonstrates an important OOP pattern: external data is converted into objects when it enters the program.

Refactor File Saving

When writing objects back to the history file, use their attributes because the file format is different from the display format.

```
for reading in readings:
    file.write(
        f"{reading.sensor_number}|"
        f"{reading.temperature}|"
        f"{reading.status}\n"
    )
```

Important distinction: The history file uses the compact machine-readable format sensor|temperature|status. __str__ is used for human-readable screen output.

Part E: Use Objects

Sets with SensorReading Objects

The Week 9 set analysis is still useful. Only the way we obtain the values changes.

```
def get_observed_statuses(self):
    return {reading.status for reading in self.reading_history}

def get_sensor_numbers(self):
    return {reading.sensor_number for reading in self.reading_history}
```

The set itself is still a set. The comprehension now reads a named attribute rather than a tuple index.

```
observed_statuses = self.get_observed_statuses()
all_statuses = SensorReading.VALID_STATUSES

print(all_statuses - observed_statuses)
print(all_statuses & observed_statuses)
```

Week 9 → Week 10 connection: The set concept has not changed. OOP changes the representation of each reading.

Dictionaries with SensorReading Objects

The status-counting dictionary is also retained.

```
def build_status_counts(self):
    counts = {}

    for reading in self.reading_history:
        status = reading.status
        counts[status] = counts.get(status, 0) + 1

    return counts
```

Likewise, the latest-reading dictionary can store the entire SensorReading object as its value.

```
def build_latest_readings(self):
    latest = {}

    for reading in self.reading_history:
        latest[reading.sensor_number] = reading
```

```
    return latest
```

Useful insight: A dictionary does not have to store only numbers or strings. Its values can be objects. This is a natural bridge from Week 9 data structures to Week 10 OOP.

Comprehensions with Objects

```
safe_sensors = {
    reading.sensor_number
    for reading in self.reading_history
    if reading.is_safe()
}
```

This is especially readable because the object provides is_safe().

Read it as English: Give me the sensor numbers for readings that are safe.

Part F: Move the Remaining Mission Features

Reports

The report method now extracts temperatures and statuses from object attributes.

```
temperatures = [
    reading.temperature for reading in self.reading_history
]

statuses = [
    reading.status for reading in self.reading_history
]
```

The calculations from earlier weeks remain unchanged: count statuses, calculate average, find maximum and minimum.

History and Recent Readings

```
def show_history(self):
    print("\n--- READING HISTORY ---")

    if not self.reading_history:
        print("No valid sensor readings recorded.")
        return

    for reading in self.reading_history:
        print(reading)
```

Notice the importance of __str__. The method can simply print each object.

Without __str__: print(reading) would normally show a technical object representation rather than the useful Mission Control format.

Search

```
for reading in self.reading_history:
    line = str(reading)

    if keyword in line.lower():
        matches.append(line)
```

The object is converted to its human-readable string representation before searching.

Reset

```
def reset_history(self, filename):
    with open(filename, "w") as file:
        file.write("")

    self.clear_history()
    print(f"History file '{filename}' has been cleared.")
```

There are now two states to keep consistent: the file and the in-memory object state. Reset clears both.

Part G: Build the Object-Oriented Main Program

Create the Mission Object

```
def main():
    mission_name = input("Mission name: ").strip().title()
    destination = input("Destination: ").strip().title()

    mission = MissionControl(mission_name, destination)
    mission.run()
```

The main program creates one MissionControl object. The object then controls the rest of the program.

> **Design improvement:** Instead of passing mission_name, destination and reading_history into many separate functions, the methods can access the state through self.

The run() Method

The command loop becomes a method of MissionControl.

```
class MissionControl:
    ...

    def run(self):
        self.load_history(HISTORY_FILE)

        ...

        while True:
            command = input(...).strip().lower()

            if command == "report":
                self.show_report()
                continue
```

The command loop is therefore operating on one mission object.

Command	**Week 10 method**
report	self.show_report()
history	self.show_history()
sets	self.show_sets()
dictionaries	self.show_dictionaries()
comprehensions	self.show_comprehensions()
scan	self.scan_sensors()
save	self.save_current_report()

shutdown	break

The Final Object Relationship

```
mission = MissionControl("Apollo", "Moon")

mission.reading_history
        |
        +--> SensorReading(...)
        +--> SensorReading(...)
        +--> SensorReading(...)
```

This is the central OOP design of Mission Control v10. MissionControl has a collection of SensorReading objects. The two classes have different responsibilities.

Class	**Main responsibility**
SensorReading	Represent one reading and answer questions about that reading.
MissionControl	Manage the mission, collection of readings, files, reports, analysis and commands.

Part H: Where Inheritance Fits

Week 10 also introduces inheritance. We do not force inheritance into the core Mission Control v10 design simply for the sake of using it. The two classes above already provide a natural and useful object-oriented structure. However, the following extension shows how inheritance could be introduced when there is a genuine 'is-a' relationship.

Optional Extension: Specialised Sensor Readings

Suppose a future version needs a CriticalReading with an additional emergency message. A CriticalReading is a SensorReading, so inheritance can be justified.

```
class CriticalReading(SensorReading):
    def __init__(self, sensor_number, temperature, message):
        super().__init__(
            sensor_number, temperature, "CRITICAL"
        )
        self.message = message

    def __str__(self):
        return (
            super().__str__()
            + " | Emergency: "
            + self.message
        )
```

- CriticalReading is the subclass.
- SensorReading is the superclass.
- super().__init__() reuses the parent initialisation.
- __str__ is overridden to add specialised information.

Teaching point: This is an optional extension, not a required change to the core v10 program. Use inheritance when the relationship is genuinely 'is a', not simply because two classes share some code.

Part I: Testing Mission Control v10

Basic Startup Test

Run the program and enter:

```
Mission name: Apollo
Destination: Moon
```

The program should display Mission Control v10.0 and show that zero historical readings are loaded if no history file exists.

Scan Test

Try temperatures such as:

Sensor	Temperature	Expected status
1	20	SAFE
2	85	WARNING
3	105	CRITICAL, scan stops

After the scan, the history contains three SensorReading objects and the file contains three corresponding lines.

Analysis Test

Try these commands:

```
report
history
recent
sets
dictionaries
comprehensions
top3
```

What to look for: The outputs should be consistent even though the internal data representation has changed from tuples to SensorReading objects.

Persistence Test

1. Run a scan and shut down.
2. Run the program again.
3. Enter the same or a different mission name and destination.
4. Check history or report.

The old readings should be reconstructed as SensorReading objects when the file is loaded.

Reset Test

```
reset
history
shutdown
```

After reset, the in-memory list and history file should both be empty.

Part J: Common OOP Mistakes in Mission Control

Mistake	Problem	Correct idea

Forgetting self	The method cannot refer to the current object.	Use self as the first instance-method parameter.
Writing temperature instead of self.temperature	Uses a local/parameter name instead of object state.	Use self.temperature.
Calling SensorReading.is_safe()	No SensorReading object is supplied as self.	Use reading.is_safe().
Using reading[0]	That assumes the object is still a tuple.	Use reading.sensor_number.
Printing object without __str__	Output may be an unhelpful object representation.	Define __str__ and return a string.
Creating reading_history outside MissionControl	Mission state is scattered.	Store it as self.reading_history.
Forgetting super().__init__() in a subclass	Inherited state may not be initialised.	Call the parent constructor when required.
Using inheritance without an is-a relationship	The class design becomes artificial.	Use inheritance only when the relationship is appropriate.
Changing the file format unnecessarily	Existing history files may no longer load.	Keep the v9 history format: sensor|temperature|status.

Part K: Student Challenges

Challenge 1 — Add a temperature category

Add a method to SensorReading that returns:

- "FREEZING" if temperature < 10
- "NORMAL" if temperature is between 10 and 80 inclusive
- "HOT" if temperature is greater than 80

Hint: The method should use self.temperature. Do not pass the temperature again if the object already stores it.

Challenge 2 — Add a mission summary method

Add a MissionControl method named show_summary() that prints the mission name, destination and number of stored readings.

Challenge 3 — Find the hottest reading

Write a MissionControl method that returns the SensorReading object with the highest temperature. Consider what should happen when there are no readings.

Challenge 4 — Count safe readings using a dictionary

Modify the dictionary analysis so that it can report how many SAFE readings occurred for each sensor.

Challenge 5 — Inheritance

Create a subclass called MaintenanceReading that inherits from SensorReading and adds a maintenance_note attribute. Override __str__ so the note appears in the output.

Check Your Understanding

Question: Why is SensorReading a useful class?

Answer: Because one reading has related data and behaviour that naturally belong together.

Question: What does self represent?

Answer: The current object on which an instance method is operating.

Question: Why is reading.temperature better than reading[1]?

Answer: It names the meaning of the data directly and is easier to read and maintain.

Question: What is MissionControl.reading_history?

Answer: An instance attribute containing SensorReading objects for one MissionControl object.

Question: Why is VALID_STATUSES a class attribute?

Answer: All SensorReading objects use the same set of valid status names.

Question: What does __str__ do?

Answer: It provides a human-readable string representation of a SensorReading.

Question: What is the relationship between MissionControl and SensorReading?

Answer: Association: a MissionControl object has a collection of SensorReading objects.

Question: Did Week 10 remove the Week 9 sets and dictionaries?

Answer: No. The same Week 9 concepts remain, but they now operate on objects and object attributes.

Question: Why is super().__init__() useful?

Answer: It lets a subclass reuse the superclass constructor to initialise inherited state.

Week 10 Revision Checklist

- I can explain the difference between a class and an object.
- I can write __init__ and instance attributes.
- I understand self.
- I can create multiple independent objects.
- I can write methods that use object state.
- I understand class attributes.
- I can use __str__.
- I can explain encapsulation at an introductory level.
- I understand association between MissionControl and SensorReading.
- I can explain inheritance and superclass/subclass.
- I can use super().__init__().
- I understand method overriding.
- I can explain polymorphism.
- I can recognise special methods.
- I can trace the state of multiple objects.
- I can explain how Week 9 data structures still work inside the Week 10 OOP design.
- I can design a class from a programming problem rather than simply copying syntax.

Appendix A — Complete Mission Control v10.0 Code

The following is the complete final Week 10 implementation. It is included without line numbers so that students can copy and paste it directly into a Python file.

```
# ============================================================
# MISSION CONTROL v10.0
# Week 10: Object-Oriented Programming
# Based on Mission Control v9.0
# ============================================================

import sys

HISTORY_FILE = "mission_history.txt"
REPORT_FILE = "mission_report.txt"


def show_header(title):
    print("=" * 60)
    print(title.center(60))
    print("=" * 60)


# ----------------------------------------------------------------------
# Week 10: Classes and objects
# ----------------------------------------------------------------------

class SensorReading:
    """Represent one sensor reading."""

    VALID_STATUSES = {"SAFE", "WARNING", "CRITICAL"}

    def __init__(self, sensor_number, temperature, status):
        self.sensor_number = sensor_number
        self.temperature = temperature
        self.status = status

    def __str__(self):
        return (
            f"Sensor {self.sensor_number:<2} | "
            f"{self.temperature:>6.1f} C | "
            f"{self.status:<8}"
        )

    def is_safe(self):
        return self.status == "SAFE"

    def is_warning(self):
        return self.status == "WARNING"

    def is_critical(self):
        return self.status == "CRITICAL"


class MissionControl:
    """Manage mission information and sensor-reading objects."""

    VALID_STATUSES = {"SAFE", "WARNING", "CRITICAL"}

    def __init__(self, mission_name, destination):
        self.mission_name = mission_name
        self.destination = destination
```

```
    self.reading_history = []

def add_reading(self, reading):
    self.reading_history.append(reading)

def add_readings(self, readings):
    self.reading_history.extend(readings)

def clear_history(self):
    self.reading_history.clear()

def classify_temperature(self, temperature):
    if temperature < 0:
        return "INVALID"
    elif temperature >= 100:
        return "CRITICAL"
    elif temperature > 80:
        return "WARNING"
    else:
        return "SAFE"

def scan_sensors(self):
    scan_readings = []
    safe_count = 0
    warning_count = 0

    for sensor_number in range(1, 6):
        while True:
            try:
                temperature = float(
                    input(f"Sensor {sensor_number} temperature: ")
                )
                break
            except ValueError:
                print("Invalid temperature. Please enter a number.")

        status = self.classify_temperature(temperature)

        if status == "INVALID":
            print("Invalid reading - skipped.")
            continue

        reading = SensorReading(
            sensor_number, temperature, status
        )
        scan_readings.append(reading)

        if reading.is_critical():
            print("CRITICAL - scan stopped")
            warning_count += 1
            break
        elif reading.is_warning():
            print("WARNING")
            warning_count += 1
        else:
            print("SAFE")
            safe_count += 1

    return scan_readings, safe_count, warning_count

def load_history(self, filename):
    self.reading_history = []
```

```
    try:
        with open(filename, "r") as file:
            for line in file:
                line = line.strip()

                if line == "":
                    continue

                parts = line.split("|")

                if len(parts) != 3:
                    print(
                        f"Warning: skipped malformed line: {line}",
                        file=sys.stderr
                    )
                    continue

                try:
                    sensor_number = int(parts[0])
                    temperature = float(parts[1])
                    status = parts[2]

                    if status not in SensorReading.VALID_STATUSES:
                        raise ValueError("invalid status")

                    reading = SensorReading(
                        sensor_number, temperature, status
                    )
                    self.reading_history.append(reading)

                except ValueError:
                    print(
                        f"Warning: skipped invalid reading: {line}",
                        file=sys.stderr
                    )

    except FileNotFoundError:
        print(
            "No existing history file found. "
            "Starting a new mission log."
        )

def append_history(self, filename, readings):
    with open(filename, "a") as file:
        for reading in readings:
            file.write(
                f"{reading.sensor_number}|"
                f"{reading.temperature}|"
                f"{reading.status}\n"
            )

def save_report(self, filename, report_lines):
    with open(filename, "w") as file:
        file.write("\n".join(report_lines))
        file.write("\n")

def create_new_report(self, filename, report_lines):
    try:
        with open(filename, "x") as file:
            file.write("\n".join(report_lines))
            file.write("\n")
        print(f"New report created: {filename}")
    except FileExistsError:
```

```
        print(
            f"Report '{filename}' already exists. "
            "Use the save command to replace it."
        )

# ------------------------------------------------------------------
# Week 9: Sets
# ------------------------------------------------------------------

def get_observed_statuses(self):
    return {reading.status for reading in self.reading_history}

def get_sensor_numbers(self):
    return {reading.sensor_number for reading in self.reading_history}

def show_sets(self):
    print("\n--- SET ANALYSIS ---")

    observed_statuses = self.get_observed_statuses()
    all_statuses = SensorReading.VALID_STATUSES

    print(f"Observed statuses: {sorted(observed_statuses)}")
    print(f"All possible statuses: {sorted(all_statuses)}")

    print(
        "Statuses not yet observed:",
        sorted(all_statuses - observed_statuses)
    )

    print(
        "Statuses shared by both sets:",
        sorted(all_statuses & observed_statuses)
    )

    print(
        "Are all possible statuses represented?",
        observed_statuses == all_statuses
    )

    sensor_numbers = self.get_sensor_numbers()
    required_sensors = set(range(1, 6))

    print(f"Observed sensors: {sorted(sensor_numbers)}")
    print(
        "Are all five sensors represented?",
        required_sensors <= sensor_numbers
    )

    if required_sensors.isdisjoint(sensor_numbers):
        print("No required sensor has been observed.")
    else:
        print("At least one required sensor has been observed.")

def demonstrate_set_methods(self):
    print("\n--- SET METHODS DEMO ---")

    s1 = {"SAFE", "WARNING"}
    s2 = {"WARNING", "CRITICAL"}

    print(f"s1: {sorted(s1)}")
    print(f"s2: {sorted(s2)}")

    print("Union:", sorted(s1 | s2))
```

```
    print("Intersection:", sorted(s1 & s2))
    print("Difference s1 - s2:", sorted(s1 - s2))
    print("Symmetric difference:", sorted(s1 ^ s2))

    print("s1 subset of s2:", s1 <= s2)
    print("s1 proper subset of s2:", s1 < s2)
    print("s1 superset of s2:", s1 >= s2)
    print("s1 proper superset of s2:", s1 > s2)
    print("s1 and s2 disjoint:", s1.isdisjoint(s2))

    working = set(s1)
    working.update(s2)
    print("After update():", sorted(working))

    working = set(s1)
    working.intersection_update(s2)
    print("After intersection_update():", sorted(working))

    working = set(s1)
    working.difference_update(s2)
    print("After difference_update():", sorted(working))

    working = set(s1)
    working.symmetric_difference_update(s2)
    print("After symmetric_difference_update():", sorted(working))

    working = {"SAFE", "WARNING"}
    print("Starting individual-method set:", sorted(working))

    working.add("CRITICAL")
    print("After add('CRITICAL'):", sorted(working))

    working.discard("MAINTENANCE")
    print("After discard('MAINTENANCE'):", sorted(working))

    working.remove("WARNING")
    print("After remove('WARNING'):", sorted(working))

    removed = working.pop()
    print("pop() removed:", removed)
    print("After pop():", sorted(working))

    working.clear()
    print("After clear():", sorted(working))

    a = {1, 2, 3}
    b = {3, 4}

    a |= b
    print("After |=:", sorted(a))

    a = {1, 2, 3}
    a &= b
    print("After &=:", sorted(a))

    a = {1, 2, 3}
    a -= b
    print("After -=:", sorted(a))

# ---------------------------------------------------------------------
# Week 9: Dictionaries
# ---------------------------------------------------------------------
```

```
def demonstrate_dictionary_methods(self):
    print("\n--- DICTIONARY METHODS DEMO ---")

    data = {"SAFE": 5, "WARNING": 2}
    print("Starting dictionary:", data)

    data["CRITICAL"] = 1
    data["SAFE"] = 6
    print("After [] assignment:", data)

    print("keys():", list(data.keys()))
    print("values():", list(data.values()))
    print("items():", list(data.items()))

    print("get('SAFE'):", data.get("SAFE"))
    print("get('MAINTENANCE'):", data.get("MAINTENANCE"))
    print("get('MAINTENANCE', 0):", data.get("MAINTENANCE", 0))

    data.setdefault("WARNING", 0)
    data.setdefault("MAINTENANCE", 0)
    print("After setdefault():", data)

    data.update({"SAFE": 7, "OTHER": 0})
    print("After update():", data)

    removed = data.pop("OTHER")
    print("pop('OTHER') returned:", removed)
    print("After pop():", data)

    removed_pair = data.popitem()
    print("popitem() returned:", removed_pair)
    print("After popitem():", data)

    del data["CRITICAL"]
    print("After del data['CRITICAL']:", data)

    template = dict.fromkeys(
        ["SAFE", "WARNING", "CRITICAL"], 0
    )
    print("fromkeys():", template)

    left = {"SAFE": 5, "WARNING": 2}
    right = {"WARNING": 3, "CRITICAL": 1}
    merged = left | right
    print("Merged with |:", merged)

    data.clear()
    print("After clear():", data)

def build_status_counts(self):
    counts = {}

    for reading in self.reading_history:
        status = reading.status
        counts[status] = counts.get(status, 0) + 1

    return counts

def build_sensor_counts(self):
    counts = {}

    for reading in self.reading_history:
        sensor_number = reading.sensor_number
```

```
        counts[sensor_number] = counts.get(sensor_number, 0) + 1

    return counts

def build_latest_readings(self):
    latest = {}

    for reading in self.reading_history:
        latest[reading.sensor_number] = reading

    return latest

def show_dictionaries(self):
    print("\n--- DICTIONARY ANALYSIS ---")

    status_counts = self.build_status_counts()
    sensor_counts = self.build_sensor_counts()
    latest = self.build_latest_readings()

    print("\nStatus counts:")
    if status_counts:
        for status in sorted(status_counts):
            print(f"{status:<10} {status_counts[status]}")
    else:
        print("No readings yet.")

    print("\nSensor reading counts:")
    if sensor_counts:
        for sensor_number in sorted(sensor_counts):
            print(
                f"Sensor {sensor_number}: "
                f"{sensor_counts[sensor_number]}"
            )
    else:
        print("No readings yet.")

    print("\nLatest reading for each sensor:")
    if latest:
        for sensor_number in sorted(latest):
            print(latest[sensor_number])
    else:
        print("No readings yet.")

    print("\nDictionary views:")
    print(f"Keys:   {sorted(status_counts.keys())}")
    print(f"Values: {list(status_counts.values())}")
    print(f"Items:  {sorted(status_counts.items())}")

    print("\nSafe lookup with get():")
    print(f"CRITICAL count: {status_counts.get('CRITICAL', 0)}")
    print(
        "MAINTENANCE count:",
        status_counts.get("MAINTENANCE", 0)
    )

# ------------------------------------------------------------------
# Week 9: Comprehensions and references
# ------------------------------------------------------------------

def show_comprehensions(self):
    print("\n--- COMPREHENSION EXAMPLES ---")

    safe_sensors = {
```

```
            reading.sensor_number
            for reading in self.reading_history
            if reading.is_safe()
        }

        average_by_sensor = {}

        for reading in self.reading_history:
            sensor_number = reading.sensor_number
            temperature = reading.temperature

            if sensor_number not in average_by_sensor:
                average_by_sensor[sensor_number] = []

            average_by_sensor[sensor_number].append(temperature)

        average_by_sensor = {
            sensor: sum(values) / len(values)
            for sensor, values in average_by_sensor.items()
        }

        print(
            "Sensors with at least one SAFE reading:",
            sorted(safe_sensors)
        )

        print("Average temperature by sensor:")
        if average_by_sensor:
            for sensor in sorted(average_by_sensor):
                print(
                    f"Sensor {sensor}: "
                    f"{average_by_sensor[sensor]:.1f} C"
                )
        else:
            print("No readings yet.")

    def show_reference_demo(self):
        print("\n--- REFERENCE DEMO ---")

        readings = [1, 2]
        mission_readings = readings

        print(f"Before mutation: readings = {readings}")
        print(
            "Before mutation: mission_readings = "
            f"{mission_readings}"
        )

        readings.append(3)

        print(f"After readings.append(3): readings = {readings}")
        print(
            "After mutation, mission_readings = "
            f"{mission_readings}"
        )

        new_readings = [10, 20]
        readings = new_readings

        print(f"After reassignment: readings = {readings}")
        print(
            "mission_readings still refers to the old list: "
            f"{mission_readings}"
```

```
    )

# ---------------------------------------------------------------------
# Existing Mission Control features from earlier weeks
# ---------------------------------------------------------------------

def show_report(self):
    print("\n--- CURRENT REPORT ---")
    total_valid = len(self.reading_history)

    if not self.reading_history:
        print("No valid sensor data yet.")
        return []

    temperatures = [
        reading.temperature for reading in self.reading_history
    ]
    statuses = [
        reading.status for reading in self.reading_history
    ]

    safe_count = statuses.count("SAFE")
    warning_count = statuses.count("WARNING")
    critical_count = statuses.count("CRITICAL")

    average = sum(temperatures) / len(temperatures)

    report_lines = [
        "--- CURRENT REPORT ---",
        f"Valid readings: {total_valid}",
        f"Safe readings: {safe_count}",
        f"Warnings: {warning_count}",
        f"Critical readings: {critical_count}",
        f"Average: {average:.1f} C",
        f"Highest: {max(temperatures):.1f} C",
        f"Lowest: {min(temperatures):.1f} C"
    ]

    for line in report_lines[1:]:
        print(line)

    return report_lines

def show_history(self):
    print("\n--- READING HISTORY ---")

    if not self.reading_history:
        print("No valid sensor readings recorded.")
        return

    for reading in self.reading_history:
        print(reading)

def show_recent(self):
    print("\n--- RECENT READINGS ---")

    recent_readings = self.reading_history[-3:]

    if not recent_readings:
        print("No valid sensor readings recorded.")
        return

    for reading in recent_readings:
```

```
        print(reading)

def show_status(self, required_status):
    matching_readings = [
        reading
        for reading in self.reading_history
        if reading.status == required_status
    ]

    print(f"\n--- {required_status} READINGS ---")

    if not matching_readings:
        print(f"No {required_status} readings recorded.")
        return

    for reading in matching_readings:
        print(reading)

def show_highest_three(self):
    print("\n--- HIGHEST THREE TEMPERATURES ---")

    if not self.reading_history:
        print("No valid sensor readings recorded.")
        return

    temperatures = [
        reading.temperature for reading in self.reading_history
    ]
    temperatures.sort()

    for temperature in temperatures[-3:][::-1]:
        print(f"{temperature:.1f} C")

def show_mission(self):
    print("\n--- MISSION INFORMATION ---")
    print(f"Mission: {self.mission_name}")
    print(f"Destination: {self.destination}")

    if len(self.mission_name) >= 3:
        print(f"Mission code: {self.mission_name[:3].upper()}")
    else:
        print("Mission code: " + self.mission_name.upper())

def search_history(self):
    keyword = input("Search text: ").strip().lower()

    if keyword == "":
        print("Search text cannot be empty.")
        return

    matches = []

    for reading in self.reading_history:
        line = str(reading)

        if keyword in line.lower():
            matches.append(line)

    print("\n--- SEARCH RESULTS ---")

    if not matches:
        print("No matching readings found.")
    else:
```

```
        print("\n".join(matches))

    def status_command(self):
        status = input(
            "Enter status (SAFE/WARNING/CRITICAL): "
        ).strip().upper()

        if status in SensorReading.VALID_STATUSES:
            self.show_status(status)
        else:
            print("Invalid status.")

    def build_log_lines(self):
        return [str(reading) for reading in self.reading_history]

    def show_log(self):
        print("\n--- MISSION LOG ---")

        if not self.reading_history:
            print("No valid sensor readings recorded.")
            return

        lines = self.build_log_lines()
        log = "\n".join(lines)

        print(log)
        print(f"\nLog contains {len(lines)} readings.")

    def save_current_report(self):
        report_lines = self.show_report()

        if not report_lines:
            return

        self.save_report(REPORT_FILE, report_lines)
        print(f"Report saved to {REPORT_FILE}.")

    def create_report_file(self):
        report_lines = self.show_report()

        if not report_lines:
            return

        self.create_new_report(REPORT_FILE, report_lines)

    def show_file_preview(self, filename):
        print(f"\n--- FILE PREVIEW: {filename} ---")

        try:
            with open(filename, "r") as file:
                lines = file.readlines()

            if not lines:
                print("File is empty.")
                return

            print(f"Lines in file: {len(lines)}")
            print("First two lines:")

            for line in lines[:2]:
                print(line.rstrip())

            if len(lines) > 2:
```

```
            print("Last two lines:")

            for line in lines[-2:]:
                print(line.rstrip())

    except FileNotFoundError:
        print(f"File '{filename}' was not found.")

def reset_history(self, filename):
    with open(filename, "w") as file:
        file.write("")

    self.clear_history()
    print(f"History file '{filename}' has been cleared.")

def show_help(self):
    command_text = (
        "scan report history recent safe warning critical "
        "sets setmethods dictionaries dictmethods comprehensions "
        "references top3 mission search status log save newreport "
        "preview reset help shutdown"
    )

    commands = command_text.split()

    print("\n--- AVAILABLE COMMANDS ---")
    print(" | ".join(commands))

def run(self):
    self.load_history(HISTORY_FILE)

    show_header(f"MISSION CONTROL v10.0 — {self.mission_name}")
    print(f"Destination: {self.destination}")
    print(
        f"Loaded {len(self.reading_history)} "
        "historical readings."
    )

    while True:
        command = input(
            "\nCommand [scan/report/history/recent/safe/warning/"
            "critical/sets/setmethods/dictionaries/dictmethods/"
            "comprehensions/references/top3/mission/search/status/"
            "log/save/newreport/preview/reset/help/shutdown]: "
        ).strip().lower()

        if command == "shutdown":
            print("Shutting down Mission Control...")
            break

        if command == "report":
            self.show_report()
            continue

        if command == "history":
            self.show_history()
            continue

        if command == "recent":
            self.show_recent()
            continue

        if command == "safe":
```

```
            self.show_status("SAFE")
            continue

        if command == "warning":
            self.show_status("WARNING")
            continue

        if command == "critical":
            self.show_status("CRITICAL")
            continue

        if command == "sets":
            self.show_sets()
            continue

        if command in ("setmethods", "set_methods"):
            self.demonstrate_set_methods()
            continue

        if command in ("dictionaries", "dict"):
            self.show_dictionaries()
            continue

        if command in ("dictmethods", "dict_methods"):
            self.demonstrate_dictionary_methods()
            continue

        if command == "comprehensions":
            self.show_comprehensions()
            continue

        if command in ("references", "reference"):
            self.show_reference_demo()
            continue

        if command == "top3":
            self.show_highest_three()
            continue

        if command == "mission":
            self.show_mission()
            continue

        if command == "search":
            self.search_history()
            continue

        if command == "status":
            self.status_command()
            continue

        if command == "log":
            self.show_log()
            continue

        if command == "save":
            self.save_current_report()
            continue

        if command == "newreport":
            self.create_report_file()
            continue
```

```
            if command == "preview":
                self.show_file_preview(HISTORY_FILE)
                continue

            if command == "reset":
                self.reset_history(HISTORY_FILE)
                continue

            if command == "help":
                self.show_help()
                continue

            if command != "scan":
                print("Unknown command.")
                continue

            scan_readings, scan_safe, scan_warning = self.scan_sensors()

            if scan_readings:
                self.add_readings(scan_readings)
                self.append_history(HISTORY_FILE, scan_readings)

            print(
                f"Scan complete: {len(scan_readings)} valid readings, "
                f"{scan_safe} safe, {scan_warning} warning/critical."
            )


def main():
    mission_name = input("Mission name: ").strip().title()
    destination = input("Destination: ").strip().title()

    mission = MissionControl(mission_name, destination)
    mission.run()

    print("=" * 60)
    print("SESSION ENDED".center(60))
    print("=" * 60)


if __name__ == "__main__":
    main()
```

Appendix B — What Changed from v9 to v10?

Area	Week 9	Week 10
Reading representation	Tuple	SensorReading object
Mission state	Local variables/list passed between functions	MissionControl object attributes
Reading formatting	format_reading()	SensorReading.__str__()
Status checks	Compare tuple element	is_safe(), is_warning(), is_critical()
Mission operations	Standalone functions	MissionControl methods
Sets/dictionaries	Operate on tuple data	Operate on object attributes
File loading	Build tuples	Build SensorReading objects
Main loop	Standalone run_mission_control()	MissionControl.run()

OOP relationship	Not explicit	MissionControl has SensorReading objects

What was deliberately preserved: The Week 9 command set, history file format, report format, sensor thresholds, file handling, sets, dictionaries, comprehensions, reference demonstration and earlier Mission Control functionality remain available in v10.

Appendix C — Suggested Student Workflow

1. Read Part A and understand why the two classes are needed.
2. Type the SensorReading class yourself instead of immediately copying the final code.
3. Create three SensorReading objects and inspect their attributes.
4. Add and test the SensorReading methods.
5. Create a MissionControl object.
6. Move one Mission Control feature at a time into the class.
7. Run the program after each major change.
8. Compare your implementation with the complete appendix only after attempting the task.
9. Use the final code as a reference, not as a substitute for understanding the design.
10. Complete at least two of the challenges before moving to the next week.

End of Week 10 Mission Control Workshop Chapter