PROGRAMMING FUNDAMENTALS

WEEK 6 WORKSHOP CHAPTER

MISSION CONTROL v6.0

From individual sensor values to structured sensor data

Lists • Tuples • Indexing • Slicing • Mutation • Methods • Comprehensions • Unpacking

Student Workshop Chapter

Lei Wang (Griffith University)

WORKSHOP RULE
Do not paste the final program first.

Build one change at a time: PREDICT → CODE → RUN → TEST → EXPLAIN.

IMPORTANT
This chapter starts from the Mission Control v5.

Keep your working v5 unchanged as a backup.

The goal is to upgrade the data model, not to replace the application with an unrelated program.

The Week 6 Mission

Mission Control v5 already works. It can scan five sensors, classify temperatures, keep session statistics, show a report, and remain active until shutdown.

The problem is that v5 mainly remembers summaries. It does not keep a complete record of every valid reading.

This week we upgrade the program so that Mission Control stores the data itself. A collection can grow as scans happen, and each stored reading can keep several related values together.

Your mission

- Start from your working mission_control_v5.py.
- Create a session history using a list.
- Represent each valid sensor reading as a tuple: (sensor_number, temperature, status).
- Use append() to add one reading to a scan.
- Use extend() to add a completed scan to the session history.
- Use indexing and slicing to inspect stored data.
- Use tuple unpacking when processing a reading.
- Use list comprehensions to create useful derived lists.
- Use len(), max(), min(), sum(), count(), in, sort() and reverse() appropriately.
- Add HISTORY and RECENT commands.
- Add useful filtering commands for SAFE, WARNING and CRITICAL readings.
- Keep the useful Week 4 loops and Week 5 functions.

BIG IDEA
Week 5 taught the program how to organise behaviour into functions.

Week 6 teaches the program how to organise data into collections.

What Is New?

The important new ideas are not simply new syntax. They change what Mission Control can remember and analyse.

Before	Week 6 upgrade	Why it matters
Separate summary variables	One reading_history list	The program keeps the underlying readings.
One temperature at a time	Tuple per reading	Sensor number, temperature and status stay together.
Manual repeated summaries	Derived lists + comprehensions	Information can be recalculated from stored data.

Only scan/report/shutdown	history/recent/status/top3 commands	The operator can inspect the stored data.

What We Keep from Weeks 1–5

Do not think of v6 as a completely new program. It is the same semester story, extended.

Week	Main idea	Where it appears in v6
1	Values, types, arithmetic, expressions, output	Temperatures, calculations, print(), formatting
2	Variables, assignment, input and conversion	Sensor input and command input
3	Comparisons, Boolean logic, if/elif/else	Temperature classification and command decisions
4	for, while, range, break, continue, counters	Sensor scan and command loop
5	Functions, parameters, return values and scope	show_header(), classify_temperature(), scan_sensors(), reports
6	Lists, tuples and collection processing	reading_history and stored sensor tuples

Before You Start

- Make a copy of your working mission_control_v5.py.
- Rename the copy mission_control_v6.py.
- Run v5 once before making any changes.
- Test scan, report and shutdown.
- Keep v5 unchanged as your rollback copy.

Quick v5 architecture

```
run_mission_control()
    |
    +-- show_header()
    |
    +-- scan_sensors()
    |       |
    |       +-- classify_temperature()
    |
    +-- show_report()
    |
    +-- shutdown
```

We keep this structure. The main change is that the program now stores complete readings in a collection.

The Data Model: List of Tuples

A list is the collection

```
reading_history = []
```

The list is mutable and can grow as new readings arrive.

A tuple is one fixed reading

```
reading = (3, 72.5, "SAFE")
```

The tuple contains three related values: sensor number, temperature and status.

The combined structure

```
reading_history = [
    (1, 50.0, "SAFE"),
    (2, 82.0, "WARNING"),
    (3, 72.5, "SAFE")
]
```

RULE
Use a list when the collection itself needs to change.

Use a tuple when several values form one fixed, ordered group.

Mission 1 — Create the Reading History

Open your v6 file and begin by changing the session state inside run_mission_control().

```
def run_mission_control():
    reading_history = []
    ...
```

This list belongs to the whole Mission Control session, so it is created before the command loop.

CHECK YOUR UNDERSTANDING: Why should reading_history be created before while True?

Answer: So the same collection survives across multiple commands and multiple scans.

Why: If it were created inside the loop, it would be reset repeatedly and previous readings would be lost.

Mission 2 — Store One Reading as a Tuple

Inside scan_sensors(), create a list for the current scan.

```
def scan_sensors():
    scan_readings = []
    ...
```

After an invalid reading has been skipped, build one tuple.

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

The tuple is one item in the list. It contains three values, but append() adds the whole tuple as one element.

CHECK YOUR UNDERSTANDING: If scan_readings is [(1, 50.0, 'SAFE')] and we append (2, 82.0, 'WARNING'), how many list elements are there?

Answer: Two.

Why: Each tuple is one element of the list.

Mission 3 — append() and extend()

append(): add one item

```
history = [(1, 50.0, "SAFE")]
reading = (2, 82.0, "WARNING")
history.append(reading)
print(history)
```

extend(): add many items

```
history = [(1, 50.0, "SAFE")]
new_scan = [
    (2, 82.0, "WARNING"),
    (3, 70.0, "SAFE")
]
history.extend(new_scan)
print(history)
```

The classic mistake

```
history.append(new_scan)  # creates a nested list
```

For Mission Control, we want a flat list of reading tuples, so the completed scan is added with extend().

CHECK YOUR UNDERSTANDING: Which should the main program use when adding a completed scan: append() or extend()?

Answer: extend().

Why: The scan already contains several reading tuples, and we want those tuples added individually to history.

Mission 4 — Indexing

Lists and tuples use zero-based indexing.

```
reading_history = [
    (1, 50.0, "SAFE"),
    (2, 82.0, "WARNING"),
    (3, 70.0, "SAFE")
]

print(reading_history[0])
print(reading_history[-1])
print(reading_history[0][1])
```

The outputs represent the first reading, the last reading, and the first reading's temperature.

INDEXING
Index 0 is the first element.

Index -1 is the last element.

Indexing selects one element.

Mission 5 — Slicing

Slicing uses start:stop:step. The start is included; the stop is excluded.

```
recent = reading_history[-3:]
print(recent)
```

This is especially useful for a monitoring system because it gives the most recent three stored readings without manually calculating the length.

```
reading_history[1:4]
reading_history[::2]
reading_history[::-1]
```

CHECK YOUR UNDERSTANDING: If reading_history has 8 elements, what does reading_history[-3:] contain?

Answer: The last three elements.

Why: The slice starts three positions from the end and continues to the end.

Mission 6 — Tuple Unpacking

```
reading = (2, 82.0, "WARNING")
sensor_number, temperature, status = reading
print(sensor_number)
print(temperature)
print(status)
```

Unpacking assigns the tuple's values to variables in order.

Use unpacking directly in a loop

```
for sensor_number, temperature, status in reading_history:
    print(
        f"Sensor {sensor_number}: "
        f"{temperature:.1f} -> {status}"
    )
```

WHY THIS IS BETTER

Tuple unpacking gives meaningful names.

It is usually easier to read than repeatedly writing reading[0], reading[1] and reading[2].

Mission 7 — List Comprehensions

A list comprehension creates a new list from an existing iterable.

```
temperatures = [reading[1] for reading in reading_history]
```

Read it as: 'for each reading in reading_history, take reading[1] and put it into the new list.'

Filtering

```
safe_readings = [
    reading
    for reading in reading_history
    if reading[2] == "SAFE"
]
```

The final if is a filter. Only readings satisfying the condition are included.

CHECK YOUR UNDERSTANDING: Which readings enter safe_readings?

Answer: Only tuples whose status is 'SAFE'.

Why: The comprehension checks reading[2] == 'SAFE' before adding the tuple.

Mission 8 — Count and Membership

```
statuses = [reading[2] for reading in reading_history]

print("SAFE" in statuses)
print(statuses.count("SAFE"))

if "CRITICAL" in statuses:
    print("A critical reading occurred.")
```

Membership answers whether a value exists. count() tells how many times it occurs.

index() — use carefully

```
if "CRITICAL" in statuses:
    position = statuses.index("CRITICAL")
    print(position)
```

index() returns the first matching position, but raises ValueError if the value is absent. Checking with in first is safer when absence is possible.

Mission 9 — Sort a Derived List

We want to analyse temperatures without changing the chronological order of reading_history.

```
temperatures = [reading[1] for reading in reading_history]
temperatures.sort()
print(temperatures)

temperatures.reverse()
print(temperatures)
```

sort() and reverse() modify the list in place.

DESIGN DECISION
Sort the derived temperatures list, not the original reading history.

History order tells us when readings arrived.

Mission 10 — HISTORY and RECENT Commands

Add history and recent to the command prompt.

```
command = input(
    "\nCommand [scan/report/history/recent/shutdown]: "
).strip().lower()
```

Then add the command branches before the scan fallback.

```
if command == "history":
    show_history(reading_history)
    continue

if command == "recent":
    show_recent(reading_history)
    continue
```

The continue is deliberate: after handling the command, the program returns to the top of the command loop.

show_history()

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

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

    for sensor_number, temperature, status in reading_history:
        print(
            f"Sensor {sensor_number}: "
            f"{temperature:.1f} -> {status}"
        )
```

show_recent()

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

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

    for sensor_number, temperature, status in reading_history[-3:]:
        print(
            f"Sensor {sensor_number}: "
            f"{temperature:.1f} -> {status}"
        )
```

Mission 11 — Rebuild the Report from the Collection

This is one of the most important design changes. The collection becomes the source of truth.

```
def show_report(reading_history):
    print("\n--- CURRENT REPORT ---")

    total_valid = len(reading_history)
    print(f"Valid readings: {total_valid}")

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

    temperatures = [reading[1] for reading in reading_history]
```

```
    statuses = [reading[2] for reading in reading_history]

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

    average = sum(temperatures) / len(temperatures)

    print(f"Safe readings: {safe_count}")
    print(f"Warnings: {warning_count}")
    print(f"Critical readings: {critical_count}")
    print(f"Average: {average:.1f}")
    print(f"Highest: {max(temperatures):.1f}")
    print(f"Lowest: {min(temperatures):.1f}")
```

Notice the conceptual improvement: v5 maintained several session-level summary variables. v6 stores the actual readings and derives the report from them. This reduces duplicated state.

Mission 12 — Add Status Filters

We can make one reusable function instead of writing three nearly identical functions.

```
def show_status(reading_history, required_status):
    matching_readings = [
        reading
        for reading in reading_history
        if reading[2] == required_status
    ]

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

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

    for sensor_number, temperature, status in matching_readings:
        print(
            f"Sensor {sensor_number}: "
            f"{temperature:.1f} -> {status}"
        )
```

Then the command loop can reuse the same function.

```
if command == "safe":
    show_status(reading_history, "SAFE")
    continue

if command == "warning":
    show_status(reading_history, "WARNING")
    continue

if command == "critical":
    show_status(reading_history, "CRITICAL")
    continue
```

REUSE
One function plus a parameter is better than three copies of the same logic. This continues the Week 5 lesson about functions.

Mission 13 — Add a Top-3 Analysis

This extension combines a comprehension, sorting, slicing and a loop.

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

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

    temperatures = [reading[1] for reading in reading_history]
    temperatures.sort()

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

The expression temperatures[-3:][::-1] first selects the three highest values and then reverses them so the highest is displayed first.

Mission 14 — Final Integration

The final integration has three important responsibilities:

- scan_sensors() builds a list of tuples for one scan.
- run_mission_control() extends the session history with the completed scan.
- Analysis functions read the history and derive reports or filtered views.

```
scan_readings = scan_sensors()
reading_history.extend(scan_readings)
```

The result is a clean data flow:

```
sensor input
    ↓
classify_temperature()
    ↓
tuple: (sensor, temperature, status)
    ↓
scan_readings list
    ↓
extend()
    ↓
reading_history list
    ↓
history / recent / report / filters / top3
```

Full Mission Control v6 Code

This is the complete reference solution. Students should attempt the missions first. Use this section to compare, debug and explain—not as a substitute for understanding.

```
# ============================================================
# MISSION CONTROL v6.0
# Collection-Based Sensor Monitor
# Week 6: Lists and Tuples
# ============================================================

def show_header(title):
    """Display a Mission Control header."""
    print("=" * 60)
    print(title.center(60))
    print("=" * 60)


def classify_temperature(temperature):
    """Return the status for one temperature reading."""
    if temperature < 0:
        return "INVALID"
    elif temperature >= 100:
        return "CRITICAL"
    elif temperature > 80:
        return "WARNING"
    else:
        return "SAFE"


def scan_sensors():
    """Scan up to five sensors and return this scan's readings."""
    scan_readings = []

    for sensor_number in range(1, 6):
        temperature = float(
            input(f"Sensor {sensor_number} temperature: ")
        )

        status = classify_temperature(temperature)

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

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

        if status == "CRITICAL":
            print("CRITICAL - scan stopped")
            break
        elif status == "WARNING":
            print("WARNING")
        else:
            print("SAFE")

    return scan_readings


def show_report(reading_history):
    """Display statistics calculated from the reading history."""
    print("\n--- CURRENT REPORT ---")

    total_valid = len(reading_history)
    print(f"Valid readings: {total_valid}")

    if not reading_history:
```

```
        print("No valid sensor data yet.")
        return

    temperatures = [reading[1] for reading in reading_history]
    statuses = [reading[2] for reading in reading_history]

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

    average = sum(temperatures) / len(temperatures)

    print(f"Safe readings: {safe_count}")
    print(f"Warnings: {warning_count}")
    print(f"Critical readings: {critical_count}")
    print(f"Average: {average:.1f}")
    print(f"Highest: {max(temperatures):.1f}")
    print(f"Lowest: {min(temperatures):.1f}")


def show_history(reading_history):
    """Display every valid reading stored in the session."""
    print("\n--- READING HISTORY ---")

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

    for sensor_number, temperature, status in reading_history:
        print(
            f"Sensor {sensor_number}: "
            f"{temperature:.1f} -> {status}"
        )


def show_recent(reading_history):
    """Display the three most recent valid readings."""
    print("\n--- RECENT READINGS ---")

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

    for sensor_number, temperature, status in reading_history[-3:]:
        print(
            f"Sensor {sensor_number}: "
            f"{temperature:.1f} -> {status}"
        )


def show_status(reading_history, required_status):
    """Display readings matching one status."""
    matching_readings = [
        reading
        for reading in reading_history
        if reading[2] == required_status
    ]

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

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

    for sensor_number, temperature, status in matching_readings:
        print(
            f"Sensor {sensor_number}: "
            f"{temperature:.1f} -> {status}"
```

```
        )


def show_highest_three(reading_history):
    """Display the three highest recorded temperatures."""
    print("\n--- HIGHEST THREE TEMPERATURES ---")

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

    temperatures = [reading[1] for reading in reading_history]
    temperatures.sort()

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


def run_mission_control():
    """Run the main Mission Control command loop."""
    reading_history = []

    show_header("MISSION CONTROL v6.0 - COLLECTION MONITOR")

    while True:
        command = input(
            "\nCommand [scan/report/history/recent/"
            "safe/warning/critical/top3/shutdown]: "
        ).strip().lower()

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

        if command == "report":
            show_report(reading_history)
            continue

        if command == "history":
            show_history(reading_history)
            continue

        if command == "recent":
            show_recent(reading_history)
            continue

        if command == "safe":
            show_status(reading_history, "SAFE")
            continue

        if command == "warning":
            show_status(reading_history, "WARNING")
            continue

        if command == "critical":
            show_status(reading_history, "CRITICAL")
            continue

        if command == "top3":
            show_highest_three(reading_history)
            continue

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

        scan_readings = scan_sensors()
        reading_history.extend(scan_readings)
```

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


run_mission_control()
```

Worked Trace

Trace one scan before running it. Suppose the operator enters:

```
50, 82, -5, 101, 70
```

The scan behaves as follows:

Input	Status	Stored?	Control action	Resulting scan list
50	SAFE	Yes	Continue normally	[(1, 50.0, 'SAFE')]
82	WARNING	Yes	Continue normally	adds (2, 82.0, 'WARNING')
-5	INVALID	No	continue	unchanged
101	CRITICAL	Yes	break	adds (4, 101.0, 'CRITICAL')
70	not reached	No	scan already stopped	unchanged

Final scan_readings:

```
[
    (1, 50.0, "SAFE"),
    (2, 82.0, "WARNING"),
    (4, 101.0, "CRITICAL")
]
```

After extend(), those three tuples are added individually to reading_history.

Prediction Games

Do not run these first. Predict the output, then execute the code.

Indexing

```
L = ["A", "B", "C", "D"]
print(L[0])
print(L[-1])
print(L[1:3])
```

CHECK YOUR UNDERSTANDING: What is the output?

Answer: A; D; ['B', 'C']

Mutation

```
L = [5, 2, 8]
L[1] = 9
L.append(3)
print(L)
```

CHECK YOUR UNDERSTANDING: What is the output?

Answer: [5, 9, 8, 3]

pop() and count()

```
L = [4, 1, 4, 2]
x = L.pop(1)
print(x)
print(L)
print(L.count(4))
```

CHECK YOUR UNDERSTANDING: What is the output?

Answer: 1; [4, 4, 2]; 2

Comprehension

```
result = [n * 2 for n in range(6) if n % 2 == 1]
print(result)
```

CHECK YOUR UNDERSTANDING: What is the output?

Answer: [2, 6, 10]

Unpacking

```
a, b = 7, 3
a, b = b, a
print(a, b)
```

CHECK YOUR UNDERSTANDING: What is the output?

Answer: 3 7

Sequence comparison

```
print((1, 9) < (2, 0))
print((1, 9) < (1, 8, 100))
```

CHECK YOUR UNDERSTANDING: What is the output?

Answer: True; False

Debugging Lab

Bug A — append vs extend

```
history = []
scan = [(1, 50.0, "SAFE"), (2, 60.0, "SAFE")]
history.append(scan)
print(history)
```

Repair / explanation: history becomes a list containing one list. Use history.extend(scan) if the desired structure is a flat list of reading tuples.

Bug B — wrong index

```
reading = (3, 72.5, "SAFE")
print(reading[3])
```

Repair / explanation: IndexError. The valid indices are 0, 1 and 2.

Bug C — tuple mutation

```
reading = (3, 72.5, "SAFE")
reading[1] = 75.0
```

Repair / explanation: TypeError. Tuples are immutable.

Bug D — unsafe index()

```
statuses = ["SAFE", "WARNING"]
print(statuses.index("CRITICAL"))
```

Repair / explanation: ValueError. Check 'CRITICAL' in statuses before calling index().

Bug E — sorting original history

```
reading_history.sort()
```

Repair / explanation: This changes the arrival order of the history. Sort a derived temperature list instead.

Systematic Testing

A program that runs without an error is not automatically correct. Test behaviour deliberately.

Test	Example	Expected result
Report before scan	report	No valid sensor data yet.
History before scan	history	No valid sensor readings recorded.

Recent before scan	recent	No valid sensor readings recorded.
All safe	50, 60, 70, 80, 75	Five readings stored; all SAFE.
Warning boundary	80.1, 50, 60, 70, 75	80.1 is WARNING.
Invalid	-5, 50, 60, 70, 75	-5 is skipped and not stored.
Critical	50, 101, 60, 70, 80	Scan stops at 101.
Repeated scans	scan twice	History contains readings from both scans.
Recent slice	More than 3 readings	Only the last three are shown.
Unknown command	hello	Unknown command.
Shutdown	shutdown	Program exits.

Boundary testing

- For the SAFE/WARNING boundary, test 79.9, 80 and 80.1.
- For the WARNING/CRITICAL boundary, test 99.9, 100 and 100.1.
- For the invalid boundary, test -0.1 and 0.
- For collection behaviour, test 0 readings, 1 reading, 3 readings and more than 3 readings.

Engineering Workflow and Responsible AI

Useful AI prompts

```
Explain the difference between append() and extend()
using Mission Control sensor readings. Give a small example
and ask me one prediction question.

Review my Mission Control function. Do not rewrite it.
Identify one logical error or edge case and explain why it matters.

Suggest normal, boundary, invalid and empty tests for my
reading_history list. Explain what each test verifies.
```

DO NOT DO THIS

Do not ask AI to produce the entire workshop solution and submit it without understanding it.

If AI gives code, compare it against your own algorithm and test it.

Mission Challenges with Model Answers

Challenge 1 — Count critical readings

Task: Create a status list and use count().

```
statuses = [reading[2] for reading in reading_history]
critical_count = statuses.count("CRITICAL")
print(critical_count)
```

Challenge 2 — Show only warnings

Task: Use a filtered list comprehension.

```
warnings = [
    reading
    for reading in reading_history
    if reading[2] == "WARNING"
]
print(warnings)
```

Challenge 3 — Highest three temperatures

Task: Create a derived temperature list, sort it, slice the final three and reverse them.

```
temperatures = [reading[1] for reading in reading_history]
temperatures.sort()
highest_three = temperatures[-3:][::-1]
print(highest_three)
```

Challenge 4 — Find the first critical sensor

Task: Use membership before index(), then use the same position in reading_history.

```
statuses = [reading[2] for reading in reading_history]

if "CRITICAL" in statuses:
    position = statuses.index("CRITICAL")
    print("First critical reading:", reading_history[position])
```

Challenge 5 — Tuple unpacking function

Task: Write a function that receives one tuple and prints a message.

```
def describe_reading(reading):
    sensor_number, temperature, status = reading
    print(
        f"Sensor {sensor_number}: "
        f"{temperature:.1f} -> {status}"
    )
```

Challenge 6 — Student Design Challenge

Add a command called safe that displays only SAFE readings. Use a function, a list comprehension and tuple unpacking.

```
def show_safe(reading_history):
    safe_readings = [
        reading
        for reading in reading_history
        if reading[2] == "SAFE"
    ]

    for sensor_number, temperature, status in safe_readings:
        print(
            f"Sensor {sensor_number}: "
            f"{temperature:.1f} -> {status}"
        )
```

Common Mistakes

Common mistake	Correct idea
First item is [1]	First item is [0].
-1 is second last	-1 is the last element.
append([3, 4]) adds two elements	It adds one item: the list [3, 4].
extend() adds one item	It adds the elements from another iterable.
remove(2) removes index 2	remove() removes the first occurrence of the value 2.
pop(2) searches for value 2	pop(2) removes the item at index 2 and returns it.
Tuples can be changed	Tuples are immutable.
sort() returns a new sorted list	sort() changes the existing list.
count() finds a position	count() counts occurrences; index() finds a first position.
index() is always safe	It raises ValueError if the value is absent.
Every comprehension is better	Use a comprehension when it makes the code clearer.

Explain Your Program Without Reading It

Close your editor. If you can answer these questions clearly, you understand the design.

Why is reading_history a list? Because the collection grows as scans occur.

Why is one sensor reading a tuple? Because sensor number, temperature and status form one fixed group.

Why does scan_sensors() use append()? Because one tuple is added at a time.

Why does run_mission_control() use extend()? Because a completed scan contains several reading tuples.

What does reading_history[-1] mean? The last stored reading.

What does reading_history[-3:] mean? The last three stored readings, or all available if fewer than three exist.

Why can a tuple be indexed but not modified? Indexing reads an element; mutation would change the tuple, which is not allowed.

Why does the report create temperatures? To analyse temperature values separately from the full reading records.

Why is count() useful? It counts how many times a status occurs.

Why check in before index()? Because index() raises ValueError when the item is absent.

Why avoid sorting reading_history? Its original order represents arrival order.

How does v6 still use Week 4 and Week 5? Loops perform repetition and functions organise the program into reusable jobs.

Final Checklist and Looking Ahead

Student completion checklist

☐ I can explain why collections are useful.

☐ I can distinguish a list from a tuple.

☐ I understand mutable versus immutable.

☐ I can create lists and tuples.

☐ I can use positive and negative indices.

☐ I can trace start, stop and step in a slice.

☐ I can use len(), max(), min(), sum(), count() and membership.

☐ I understand append() versus extend().

☐ I understand remove() versus pop().

☐ I can use list comprehensions and filtered comprehensions.

☐ I can unpack tuples.

☐ I understand tuple-based returned values.

☐ I can sort a derived list without destroying history order.

☐ I can explain how Mission Control stores sensor history.

☐ I can explain every important line of my final program.

☐ I tested normal, boundary, invalid, empty and repeated-use cases.

☐ I can explain how I used AI responsibly, if I used it.

The Week 6 upgrade in one picture

```
v5
individual summary variables
        ↓
v6
reading_history
    ├── (sensor, temperature, status)
    ├── (sensor, temperature, status)
    ├── (sensor, temperature, status)
    └── ...

        ↓

indexing / slicing
        +
tuple unpacking
        +
list comprehensions
        +
count / membership
        +
sorting
        ↓

history / recent / report / filters / top3
```

GOLDEN HABIT
PREDICT → CODE → RUN → TEST → EXPLAIN. A working program is the starting point; understanding and verification are the goal.

Looking ahead: this collection-based design prepares Mission Control for later topics such as files, modules and larger data structures. The key transition this week is from 'the program remembers a few totals' to 'the program remembers the data and can calculate different views from it.'