Programming Principles

Week 9 Workshop

Mission Control v9.0

Sets and Dictionaries

Extend Mission Control v8.0 by using sets and dictionaries to represent information that lists alone do not represent naturally. You will first understand each idea with small examples, then use the idea in the mission program.

This workshop is written as a learning sequence. Do not simply copy the final program. At each step, run the small example, predict what it should do, and then connect it to Mission Control.

What you should be able to do

- Explain why a set is useful when duplicate values are not wanted.
- Create sets, test membership, and use union, intersection, difference and symmetric difference.
- Use set comparison operations such as subset, superset and disjointness.
- Use set methods that change a set, including update(), add(), discard(), remove(), pop() and clear().
- Explain the difference between a dictionary key and a dictionary value.
- Create, access, update and remove dictionary entries.
- Use get() and setdefault() when a key may not already exist.
- Iterate through dictionary keys, values and key-value pairs.
- Use dictionaries for counting occurrences and storing the latest value associated with a key.
- Recognise set and dictionary comprehensions.
- Understand the reference/aliasing example introduced in the Week 9 material.
- Integrate the new structures into a working Mission Control program.

Before you start

- Use the Mission Control v8.0 program as your starting point.
- Make a backup copy before changing it.
- Use Python 3.9 or later for the dictionary merge operator | used in this workshop.
- Run the program frequently. Small tests make errors much easier to locate.
- When a set is printed, its display order should not be treated as meaningful. The workshop sorts set output when predictable display is useful.

A list answers 'what items are stored, and in what order?'. A set answers 'which distinct values are present?'. A dictionary answers 'what value is associated with this key?'.

Part A: Understand sets

The Week 9 lecture describes sets as unordered collections of unique values. That makes them useful when the important question is whether a value is present, rather than where it occurs in a sequence.

Create a set of mission statuses

Why? Mission Control can contain many readings with the same status. For example, ten readings might contain SAFE many times. If we only want to know which different statuses have appeared, a list is unnecessarily repetitive.

Do this. Open a Python shell or a small test file and run:

```
statuses = {"SAFE", "WARNING", "SAFE", "CRITICAL", "WARNING"}
print(statuses)
```

What should you understand? The repeated SAFE and WARNING values are stored only once. This is the defining feature we need from a set.

Try it: Change the example so that the same status appears three or four times. Does the set contain duplicates?

Notice the empty-set syntax

Why? Curly braces are also used for dictionaries, so {} does not mean an empty set.

Do this. Run these two statements:

```
empty_set = set()
empty_dictionary = {}

print(type(empty_set))
print(type(empty_dictionary))
```

What should you understand? Use set() when you need an empty set. The expression {} creates an empty dictionary.

Test membership and count distinct values

Why? A common set question is simply whether something is present. The in operator is a natural fit, and len() tells us how many distinct values are present.

Do this. Run:

```
statuses = {"SAFE", "WARNING", "CRITICAL"}

print("SAFE" in statuses)
print("MAINTENANCE" in statuses)
print(len(statuses))
```

What should you understand? Membership asks whether a value belongs to the set. len(statuses) counts distinct values, not the number of times those values originally appeared.

Learn union: values in either set

Why? Mission Control can compare two groups of statuses, such as statuses observed by two scans. Union combines the values while removing duplicates.

Do this. Run:

```
morning = {"SAFE", "WARNING"}
afternoon = {"WARNING", "CRITICAL"}

print(morning | afternoon)
print(morning.union(afternoon))
```

What should you understand? Both forms produce the same set: values appearing in either input set.

Try it: Which statuses occur in at least one of the two scans?

Learn intersection: values in both sets

Why? Sometimes we want the overlap between two groups. Intersection keeps only values that occur in both sets.

Do this. Run:

```
morning = {"SAFE", "WARNING"}
afternoon = {"WARNING", "CRITICAL"}

print(morning & afternoon)
print(morning.intersection(afternoon))
```

What should you understand? The result contains WARNING because it appears in both sets.

Learn difference: values in the first set but not the second

Why? Difference is directional. It answers 'what is in the first set that is not in the second?' This is useful when comparing what has been observed with what is still missing.

Do this. Run:

```
all_statuses = {"SAFE", "WARNING", "CRITICAL"}
observed = {"SAFE", "WARNING"}

print(all_statuses - observed)
print(observed - all_statuses)
```

What should you understand? all_statuses - observed gives statuses not yet observed. Reversing the operands asks a different question.

Learn symmetric difference: values in only one set

Why? Sometimes the overlap is not what we want. Symmetric difference keeps values that belong to exactly one of the sets.

Do this. Run:

```
first = {"SAFE", "WARNING"}
second = {"WARNING", "CRITICAL"}

print(first ^ second)
print(first.symmetric_difference(second))
```

What should you understand? WARNING is removed because it occurs in both sets. SAFE and CRITICAL remain because each occurs in only one set.

Compare sets

Why? A mission can have required values. We can ask whether every required value is present, whether one set contains another, or whether two sets have no common values.

Do this. Run:

```
required = {"SAFE", "WARNING"}
observed = {"SAFE", "WARNING", "CRITICAL"}

print(required <= observed)
print(required < observed)
print(observed >= required)
print(observed > required)
print(required.isdisjoint({"MAINTENANCE"}))
```

What should you understand? <= and >= allow subset/superset tests. < and > require a proper subset/superset. isdisjoint() is True when the two sets have no values in common.

A key distinction: operation versus method

The lecture shows two equivalent styles for the four main set operations. For example, s1 | s2 and s1.union(s2) both produce the union. Learning both forms is useful because you will see both in Python programs.

Learn the methods that change a set

The previous operations create and return a result. The following update methods instead change the set they are called on and return None. This distinction matters.

```
a = {"SAFE", "WARNING"}
b = {"WARNING", "CRITICAL"}

a.update(b)
print(a)

a = {"SAFE", "WARNING"}
a.intersection_update(b)
print(a)

a = {"SAFE", "WARNING"}
a.difference_update(b)
print(a)

a = {"SAFE", "WARNING"}
a.symmetric_difference_update(b)
print(a)
```

Why this matters: If you write result = a.update(b), result will be None. update() changes a; it is not the same kind of expression as a | b.

Add and remove individual set values

The lecture also introduces add(), discard(), remove(), pop(), clear() and del. The important difference is what happens when the requested item does not exist.

```
statuses = {"SAFE", "WARNING"}

statuses.add("CRITICAL")
print(statuses)
```

```
statuses.discard("MAINTENANCE")
print(statuses)

statuses.remove("WARNING")
print(statuses)

removed = statuses.pop()
print("Removed:", removed)
print(statuses)

statuses.clear()
print(statuses)
```

discard() does nothing if the value is absent. remove() raises an error if the value is absent. pop() removes and returns an arbitrary set element, so do not write a program that depends on which element pop() chooses.

Apply the set idea to Mission Control

Now we have a real reason to add a set to the program. The history already stores individual readings as tuples. We can derive a set containing the distinct statuses observed in the history.

```
def get_observed_statuses(reading_history):
    return {reading[2] for reading in reading_history}
```

Why reading[2]? Each reading is stored as a tuple (sensor_number, temperature, status), so index 2 contains the status. The set automatically removes duplicates.

We can then compare the observed statuses with the set of all possible statuses:

```
observed_statuses = get_observed_statuses(reading_history)
all_statuses = {"SAFE", "WARNING", "CRITICAL"}

print(all_statuses - observed_statuses)
print(observed_statuses == all_statuses)
```

Checkpoint: Before continuing, make sure you can explain in words what each of these means: all_statuses - observed_statuses, all_statuses & observed_statuses, and observed_statuses == all_statuses.

Part B: Understand dictionaries

A dictionary stores key-value pairs. The key identifies an entry, and the value is the information associated with that key. Unlike a list, a dictionary is not accessed by a zero-based position.

Create and access a dictionary

Why? Mission Control needs a way to associate a sensor number with information about that sensor. A dictionary is designed for this key-to-value relationship.

Do this. Run:

```
sensor_names = {
    1: "Thermal Sensor",
    2: "Pressure Sensor",
    3: "Oxygen Sensor"
}

print(sensor_names[2])
```

What should you understand? The key 2 is used to retrieve the value 'Pressure Sensor'. The dictionary is not asking for the second position.

Add a new entry and replace an existing value

Why? Using [] with a new key adds an entry. Using [] with an existing key replaces its value.

Do this. Run:

```
counts = {"SAFE": 5, "WARNING": 2}

counts["CRITICAL"] = 1
counts["SAFE"] = 6

print(counts)
```

What should you understand? The dictionary still has one SAFE entry; its value has changed from 5 to 6. Dictionary keys are unique.

Use in and not in with dictionary keys

Why? Before looking up a key directly, it can be useful to check whether that key exists.

Do this. Run:

```
phonebook = {
    "John": 28630,
    "Bela": 28761,
    "Alan": 28671
}

print("Alan" in phonebook)
print("Maria" not in phonebook)
```

What should you understand? Dictionary membership tests keys. It does not search through values unless you explicitly ask it to.

Use get() for safer lookup

Why? Direct lookup such as data['CRITICAL'] raises a KeyError when the key is absent. get() lets us provide a default instead.

Do this. Run:

```
counts = {"SAFE": 5, "WARNING": 2}

print(counts.get("SAFE"))
print(counts.get("CRITICAL"))
print(counts.get("CRITICAL", 0))
```

What should you understand? get('CRITICAL', 0) means: return the CRITICAL value if the key exists; otherwise use 0.

Learn keys(), values() and items()

Why? A dictionary is often processed with a loop. These methods give us the keys, values, or key-value pairs we want to process.

Do this. Run:

```
counts = {"SAFE": 5, "WARNING": 2, "CRITICAL": 1}

print(list(counts.keys()))
print(list(counts.values()))
print(list(counts.items()))

for status, count in counts.items():
    print(status, count)
```

What should you understand? items() is especially useful when a loop needs both the key and its associated value.

Use setdefault()

Why? setdefault() is useful when a key might not exist. It returns the existing value, or inserts the default value when the key is missing.

Do this. Run:

```
counts = {"SAFE": 5}

print(counts.setdefault("SAFE", 0))
print(counts.setdefault("CRITICAL", 0))
print(counts)
```

What should you understand? SAFE already existed, so its value remains 5. CRITICAL did not exist, so the dictionary gains CRITICAL: 0.

Update and remove dictionary entries

The lecture introduces update(), pop(), popitem(), del and clear(). These methods support different kinds of changes.

```
data = {"SAFE": 5, "WARNING": 2}

data.update({"SAFE": 6, "CRITICAL": 1})
```

```
print(data)

removed = data.pop("CRITICAL")
print("Removed value:", removed)
print(data)

pair = data.popitem()
print("Removed pair:", pair)
print(data)

data["CRITICAL"] = 1
del data["CRITICAL"]
print(data)

data.clear()
print(data)
```

pop(key) removes a named key and returns its value. popitem() removes and returns a key-value pair. del removes a selected entry, while clear() removes all entries.

Create a dictionary with fromkeys()

fromkeys() can create a dictionary from a collection of keys and give every key the same initial value.

```
statuses = ["SAFE", "WARNING", "CRITICAL"]
counts = dict.fromkeys(statuses, 0)

print(counts)
```

This is convenient when we already know the complete set of keys and want every initial count to be zero.

Merge dictionaries with |

The Week 9 material introduces dictionary merging with | from Python 3.9. When a key occurs in both dictionaries, the value from the right-hand dictionary is used.

```
first = {"SAFE": 5, "WARNING": 2}
second = {"WARNING": 3, "CRITICAL": 1}

merged = first | second
print(merged)
```

Remember: The two dictionaries are not changed by this expression. A new merged dictionary is produced.

Part C: Use dictionaries for real Mission Control problems

This is where the new data structure becomes genuinely useful. A list of tuples is good for storing individual readings. A dictionary is better when we want to organise information by a key.

Count how many times each status occurs

Why? A frequency table is a natural dictionary problem: the status is the key, and the count is the value.

Do this. Start with an empty dictionary and process each reading:

```
counts = {}

for reading in reading_history:
    status = reading[2]
    counts[status] = counts.get(status, 0) + 1
```

What should you understand? For the first SAFE reading, get('SAFE', 0) gives 0, so the count becomes 1. For the next SAFE reading, get('SAFE', 0) gives 1, so the count becomes 2.

Why get() makes counting simpler

The lecture first shows a longer if/else version and then simplifies it with get(). The key pattern is:

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

Read it from right to left: obtain the current count (or 0 if the key is absent), add 1, then store the new count under the same key.

Count readings by sensor

Why? The same pattern can answer a different question. This time the sensor number is the key.

Do this. Add a second function:

```
def build_sensor_counts(reading_history):
    counts = {}

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

    return counts
```

What should you understand? The same dictionary technique can solve many counting problems. Only the key we extract from each item changes.

Store the latest reading for each sensor

Why? A dictionary is also useful when we want one value associated with each key. Because reading_history is processed from oldest to newest, assigning the same sensor key again replaces its earlier value.

Do this. Run the function below with the mission history:

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

    for reading in reading_history:
        sensor_number = reading[0]
```

```
        latest[sensor_number] = reading

    return latest
```

What should you understand? If sensor 2 appears several times, the final assignment to latest[2] is its latest reading.

Put the dictionary analysis together

The Mission Control v9 program combines the three dictionary ideas: counts by status, counts by sensor, and latest reading by sensor.

```
status_counts = build_status_counts(reading_history)
sensor_counts = build_sensor_counts(reading_history)
latest = build_latest_readings(reading_history)

for status in sorted(status_counts):
    print(status, status_counts[status])

for sensor in sorted(sensor_counts):
    print(sensor, sensor_counts[sensor])

for sensor in sorted(latest):
    print(format_reading(latest[sensor]))
```

Why sorted() is used here: Dictionary key order is insertion-preserving in modern Python, but sorting makes the displayed analysis predictable and easier for students to compare across runs.

Part D: Comprehensions and references

Set comprehension

The Week 9 material shows set comprehensions as a compact way to create a set from an iterable. They are similar in structure to list comprehensions, but the surrounding braces create a set.

```
squares = {n ** 2 for n in range(6)}
print(squares)

remainders = {n % 3 for n in range(20)}
print(remainders)
```

The second example produces only 0, 1 and 2 because a set removes duplicates.

Dictionary comprehension

A dictionary comprehension adds a key:value expression inside the braces.

```
squares = {n: n ** 2 for n in range(6)}
print(squares)
```

Each n becomes a key, and n ** 2 becomes its associated value.

Apply comprehensions to Mission Control

We can create the set of sensors that have at least one SAFE reading:

```
safe_sensors = {
    reading[0]
    for reading in reading_history
    if reading[2] == "SAFE"
}
```

The important idea is not the compact syntax itself. First understand the equivalent loop, then use the comprehension when the shorter form is clear.

References, mutation and reassignment

The Week 9 material also includes an example showing that assigning one list variable to another does not copy the list. Both names can refer to the same object.

```
readings = [1, 2]
mission_readings = readings

readings.append(3)

print(readings)
print(mission_readings)
```

Both variables show [1, 2, 3] because append() changed the shared list. Reassignment is different:

```
new_readings = [10, 20]
readings = new_readings

print(readings)
print(mission_readings)
```

Now readings refers to a different list. mission_readings still refers to the original list.

Common mistake: Changing an object and changing which object a variable refers to are two different operations. This distinction becomes important when working with lists, dictionaries and sets.

Part E: Add the Week 9 features to Mission Control

At this point, you should understand the individual concepts. The final integration adds new commands without removing the features built in earlier weeks.

Add the Week 9 commands

The program adds these commands:

Command	Purpose	Main Week 9 idea
sets	Analyse distinct statuses and sensors	Sets and set operations
setmethods	See the main set methods	Set methods
dictionaries	Analyse counts and latest readings	Dictionaries
dictmethods	See the main dictionary methods	Dictionary methods
comprehensions	See set/dictionary-style compact construction	Comprehensions
references	See aliasing, mutation and reassignment	References

Why the new features are separate commands

Keeping the demonstrations in separate commands makes the program easier to learn and debug. You can test one concept without having to run every feature. It also makes it clear which part of the program is responsible for each Week 9 idea.

Test the completed program

Run the complete program and first use help. Then try the Week 9 commands before scanning new data.

```
help
sets
setmethods
dictionaries
dictmethods
comprehensions
references
scan
report
shutdown
```

For the scan, try temperatures such as 20, 85 and 101. This gives you SAFE, WARNING and CRITICAL data, so the set and dictionary analyses have meaningful values to work with.

A useful test scenario

Use the following sequence. Predict what should happen before you run each command.

Step	What to test
1	Run sets before any scan. What are the observed statuses?
2	Scan 20, 85, 101. Notice that the scan stops after CRITICAL.
3	Run sets again. Which statuses are now observed?
4	Run dictionaries. Compare the status counts with the history.
5	Run comprehensions. Which sensors have at least one SAFE reading?
6	Run references. Explain why both variables change after append().
7	Run report and history to check that the older Mission Control features still work.

Common mistakes to watch for

Mistake	Better approach
Using {} when you want an empty set	Use set() for an empty set.
Expecting a set to preserve a meaningful order	Treat a set as an unordered collection; sort it only when predictable display is useful.
Confusing | for set union and dictionary merge	The same symbol has related meanings depending on the operand type.
Writing result = my_set.update(other)	update() changes my_set and returns None.
Using remove() when an item may be absent	Use discard() if absence should not be an error.
Assuming dictionary membership searches values	key in dictionary tests keys.
Using dictionary positions like a list	Dictionaries are accessed by keys.
Accessing a missing dictionary key with []	Use get() when a missing key is a normal possibility.
Forgetting that dictionary keys are unique	Assigning an existing key replaces its value.
Depending on which element set.pop() removes	The method removes an arbitrary set element.

Your own modifications

Once the supplied version works, make at least one small change yourself. The aim is to demonstrate that you understand the data structure rather than only copying the code.

- Add a command that reports which sensor numbers have been observed.
- Add a command that reports whether all three statuses have appeared.
- Use a dictionary to count how many readings came from each sensor.
- Modify the dictionary analysis so that it also reports the total number of distinct statuses.
- Create a small set of your own and test all four main set operations.

Quick checklist

☐ I can explain why a set removes duplicates.

☐ I know that set() creates an empty set and {} creates an empty dictionary.

☐ I can explain |, &, -, and ^ for sets.

☐ I know the difference between an operation that returns a new set and an update method that changes a set.

☐ I know the difference between discard() and remove().

☐ I understand subset, superset and disjoint comparisons.

☐ I can create a dictionary with key:value pairs.

☐ I can add or replace a dictionary entry using [].

☐ I can use get() to supply a default value.

☐ I can iterate through keys(), values() and items().

☐ I can use a dictionary to count occurrences.

☐ I understand how assigning the same dictionary key again replaces its value.

☐ I recognise set and dictionary comprehensions.

☐ I can explain the reference/aliasing example.

Appendix: Mission Control v9.0 code

The complete program is provided below. You can copy and paste it directly into a Python file. It includes the earlier Mission Control functionality plus the Week 9 sets, dictionaries, comprehensions and reference demonstrations.

If you are learning, first build the program step by step using the sections above. Use this complete version to compare your work, recover from mistakes, or run the final program.

```
# ============================================================
# MISSION CONTROL v9.0
# Week 9: Sets and Dictionaries
# Based on Mission Control v8.0
# ============================================================

import sys

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


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


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


def format_reading(reading):
    sensor_number, temperature, status = reading
    return f"Sensor {sensor_number:<2} | {temperature:>6.1f} C | {status:<8}"


def load_history(filename):
    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 ("SAFE", "WARNING", "CRITICAL"):
                        raise ValueError("invalid status")

                    reading = (sensor_number, temperature, status)
                    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.")

    return reading_history


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


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


def create_new_report(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."
        )


def scan_sensors():
    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 = 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")
            warning_count += 1
            break
        elif status == "WARNING":
            print("WARNING")
            warning_count += 1
        else:
            print("SAFE")
            safe_count += 1

    return scan_readings, safe_count, warning_count


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

def get_observed_statuses(reading_history):
    """Return the distinct statuses currently present in the history."""
    return {reading[2] for reading in reading_history}


def get_sensor_numbers(reading_history):
    """Return the distinct sensor numbers currently present in the history."""
    return {reading[0] for reading in reading_history}


def show_sets(reading_history):
    """Use sets to analyse distinct mission values."""
    print("\n--- SET ANALYSIS ---")

    observed_statuses = get_observed_statuses(reading_history)
    all_statuses = {"SAFE", "WARNING", "CRITICAL"}

    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 = get_sensor_numbers(reading_history)
    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():
    """Demonstrate the main set operations and methods from Week 9."""
    print("\n--- SET METHODS DEMO ---")

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

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

    # Non-mutating set operations.
    print("Union:", sorted(s1 | s2))
    print("Intersection:", sorted(s1 & s2))
    print("Difference s1 - s2:", sorted(s1 - s2))
    print("Symmetric difference:", sorted(s1 ^ s2))

    # Set comparisons.
    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))

    # Mutating operations.
    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))

    # Individual-element methods.
    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))

    # Shorthand assignment operators.
    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():
    """Demonstrate the main dictionary operations and methods from Week 9."""
    print("\n--- DICTIONARY METHODS DEMO ---")

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

    # [] can add a new key or replace the value of an existing key.
    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()))

    # get() avoids a KeyError when a key may be absent.
    print("get('SAFE'):", data.get("SAFE"))
    print("get('MAINTENANCE'):", data.get("MAINTENANCE"))
    print("get('MAINTENANCE', 0):", data.get("MAINTENANCE", 0))

    # setdefault() keeps an existing value, or inserts a default.
    data.setdefault("WARNING", 0)
    data.setdefault("MAINTENANCE", 0)
    print("After setdefault():", data)

    # update() adds/replaces several key-value pairs.
    data.update({"SAFE": 7, "OTHER": 0})
    print("After update():", data)

    # pop() returns and removes a selected key.
    removed = data.pop("OTHER")
    print("pop('OTHER') returned:", removed)
    print("After pop():", data)

    # popitem() returns and removes one key-value pair.
    removed_pair = data.popitem()
    print("popitem() returned:", removed_pair)
    print("After popitem():", data)

    # del removes a selected entry.
    del data["CRITICAL"]
    print("After del data['CRITICAL']:", data)

    # fromkeys() creates a new dictionary from an iterable of keys.
    template = dict.fromkeys(
        ["SAFE", "WARNING", "CRITICAL"], 0
    )
    print("fromkeys():", template)

    # | merges two dictionaries. If a key occurs in both,
    # the value from the right-hand dictionary is used.
    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(reading_history):
    """Build a dictionary mapping each status to its frequency."""
    counts = {}

    for reading in reading_history:
        status = reading[2]
        counts[status] = counts.get(status, 0) + 1

    return counts


def build_sensor_counts(reading_history):
```

```
    """Build a dictionary mapping each sensor number to its reading count."""
    counts = {}

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

    return counts


def build_latest_readings(reading_history):
    """Build a dictionary mapping each sensor to its latest reading."""
    latest = {}

    for reading in reading_history:
        sensor_number = reading[0]
        latest[sensor_number] = reading

    return latest


def show_dictionaries(reading_history):
    """Use dictionaries to organise and summarise mission data."""
    print("\n--- DICTIONARY ANALYSIS ---")

    status_counts = build_status_counts(reading_history)
    sensor_counts = build_sensor_counts(reading_history)
    latest = build_latest_readings(reading_history)

    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(format_reading(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(reading_history):
    """Demonstrate set and dictionary comprehensions."""
    print("\n--- COMPREHENSION EXAMPLES ---")

    safe_sensors = {
        reading[0]
        for reading in reading_history
        if reading[2] == "SAFE"
    }

    average_by_sensor = {}

    for reading in reading_history:
        sensor_number = reading[0]
        temperature = reading[1]

        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():
    """Show aliasing, mutation and reassignment."""
    print("\n--- REFERENCE DEMO ---")

    readings = [1, 2]
    mission_readings = readings

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

```
    print(f"Before mutation: mission_readings = {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(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)

    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(reading_history):
```

```
    print("\n--- READING HISTORY ---")

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

    lines = []

    for reading in reading_history:
        lines.append(format_reading(reading))

    print("\n".join(lines))


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

    recent_readings = reading_history[-3:]

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

    for reading in recent_readings:
        print(format_reading(reading))


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 reading in matching_readings:
        print(format_reading(reading))


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} C")


def show_mission(mission_name, destination):
    print("\n--- MISSION INFORMATION ---")
```

```
    print(f"Mission: {mission_name}")
    print(f"Destination: {destination}")

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


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

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

    matches = []

    for reading in reading_history:
        line = format_reading(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(reading_history):
    status = input(
        "Enter status (SAFE/WARNING/CRITICAL): "
    ).strip().upper()

    if status in ("SAFE", "WARNING", "CRITICAL"):
        show_status(reading_history, status)
    else:
        print("Invalid status.")


def build_log_lines(reading_history):
    lines = []

    for reading in reading_history:
        lines.append(format_reading(reading))

    return lines


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

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

    lines = build_log_lines(reading_history)
```

```
    log = "\n".join(lines)

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


def save_current_report(reading_history):
    report_lines = show_report(reading_history)

    if not report_lines:
        return

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


def create_report_file(reading_history):
    report_lines = show_report(reading_history)

    if not report_lines:
        return

    create_new_report(REPORT_FILE, report_lines)


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

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


def show_help():
```

```
    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_mission_control():
    reading_history = load_history(HISTORY_FILE)

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

    show_header(f"MISSION CONTROL v9.0 — {mission_name}")
    print(f"Destination: {destination}")
    print(f"Loaded {len(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":
            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 == "sets":
```

```
            show_sets(reading_history)
            continue

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

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

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

        if command == "comprehensions":
            show_comprehensions(reading_history)
            continue

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

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

        if command == "mission":
            show_mission(mission_name, destination)
            continue

        if command == "search":
            search_history(reading_history)
            continue

        if command == "status":
            status_command(reading_history)
            continue

        if command == "log":
            show_log(reading_history)
            continue

        if command == "save":
            save_current_report(reading_history)
            continue

        if command == "newreport":
            create_report_file(reading_history)
            continue

        if command == "preview":
            show_file_preview(HISTORY_FILE)
            continue

        if command == "reset":
            reset_history(HISTORY_FILE)
            reading_history = []
            continue
```

```
        if command == "help":
            show_help()
            continue

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

        scan_readings, scan_safe, scan_warning = scan_sensors()

        if scan_readings:
            reading_history.extend(scan_readings)
            append_history(HISTORY_FILE, scan_readings)

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

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


run_mission_control()
```

Mission Control v9.0 now uses two new compound data structures for different purposes. Sets represent distinct values and support operations such as union, intersection and difference. Dictionaries associate keys with values and support direct lookup, updating and counting. The program therefore moves beyond simply storing readings: it can also organise and analyse them.

The most important goal is not memorising every method. It is being able to look at a problem and choose an appropriate structure: use a list when order and sequence matter, a tuple for a fixed group of values, a set when uniqueness/membership matters, and a dictionary when a key-to-value relationship matters.