PROGRAMMING FUNDAMENTALS

WEEK 2 WORKSHOP

MISSION CONTROL PANEL v2.0

Lei Wang (Griffith University)

WORKSHOP MISSION

Last week your dashboard could calculate mission information from fixed values.

This week you will make it interactive: the user supplies the mission data, the program stores it in meaningful variables, processes it, and produces a readable mission report.

What are we building?

You are not starting a completely new project.

You are upgrading the Mission Control Panel you built in Week 1.

The key design principle is that each week's new concept should solve a limitation of the previous version.

Week 1 limitation	Week 2 solution
Mission values are hard-coded.	Ask the user for mission information.
Changing a mission requires editing source code.	The same program can be reused for different missions.
Values are difficult to identify in a larger program.	Give data meaningful names such as mission_name and rocket_speed.
Output is mainly a collection of calculations.	Generate a personalised mission report.

BIG IDEA

A variable is not just a shorter way to write a value.

A meaningful variable name gives the data a role and makes the program easier to understand.

Learning goals

- Use input() to collect text from a user.
- Understand that input() returns a string.
- Convert numeric input using int() or float().
- Store information in meaningful variables.
- Use assignment and reassignment correctly.
- Reuse Week 1 arithmetic operators in a larger program.
- Use // and % for practical mission calculations.
- Use print(), sep, end and simple string techniques to create readable output.
- Organise a program as INPUT -> PROCESS -> OUTPUT.
- Predict, test and debug each stage rather than pasting a complete program.
- Explain how the Week 2 version is an improvement over Week 1.

Before you code

Create a new file, for example:

```
mission_control_v2.py
```

Run it from your terminal with:

```
python3 mission_control_v2.py
```

WORKSHOP RULE

Build the program in small sections. Run the program after each section.

If something breaks, fix that section before moving on.

This is much more useful than pasting the complete solution.

Part A - Build the dashboard header

Start with a visual identity. This reuses Week 1 string repetition and print().

```
# ============================================================
# AUSTRALIAN SPACE MISSION CONTROL - VERSION 2.0
# ============================================================

print("=" * 60)
print(" AUSTRALIAN SPACE MISSION CONTROL ".center(60, "="))
print(" Interactive Mission Planner - v2.0 ".center(60))
print("=" * 60)
```

THINK / PREDICT

Which ideas from Week 1 can you recognise here?
Hint: Look for print(), strings, *, and expressions.

Try changing the title. Do not change the structure yet.

1. Replace the title with your own mission-control name.
2. Change the version to v2.0.
3. Run the program and check the alignment.

Part B - Add mission identity

Now the dashboard asks the user who is operating the mission and what the mission is.

```
commander = input("\nCommander name: ")
mission_name = input("Mission name: ")
destination = input("Destination: ")
```

```
print("\nWelcome, " + commander + "!")
print("Configuring mission", mission_name, "to", destination + "...")
```

THINK / PREDICT

Which three variables contain strings?

Hint: The user is typing text, and input() returns str.

KEY INSIGHT

input() is both an input mechanism and a type clue: the value returned by input() is str. That is why text such as a name can be stored directly, while numeric input normally needs conversion.

Experiment

```
name = input("Enter your name: ")
print(type(name))
print(name * 3)
```

1. Enter your own name.
2. Predict what name * 3 will do.
3. Run it.
4. Explain why repetition happens.

COMMON TRAP

If a user types 25, input() gives the string "25". It is not automatically the integer 25.

Part C - Add mission parameters

Mission calculations require numbers, so convert the keyboard input before storing it.

```
rocket_speed = float(input("\nRocket speed (km/h): "))
mission_hours = float(input("Mission duration (hours): "))
fuel_required = float(input("Fuel required (litres): "))
tank_capacity = float(input("Fuel tank capacity (litres): "))
```

THINK / PREDICT

What would happen if rocket_speed were stored without float() and we later tried to multiply it by mission_hours?

Hint: Remember the type of the value returned by input().

Variable	Expected type	Why?
commander	str	Name is text.
mission_name	str	Mission title is text.
destination	str	Destination is text.
rocket_speed	float	May contain decimals and is used in arithmetic.
mission_hours	float	Duration may contain decimals.
fuel_required	float	Fuel amount may contain decimals.
tank_capacity	float	Capacity may contain decimals.

int() or float()?

Use int() when the concept is naturally a whole number. Use float() when decimal values make sense.

```
crew_size = int(input("Crew size: "))
rocket_speed = float(input("Rocket speed (km/h): "))
```

THINK / PREDICT

Would crew_size normally need float()? Why or why not?

Part D - Process the mission data

Now return to Week 1 mathematics. The important change is that the values are no longer hard-coded.

```
distance = rocket_speed * mission_hours
full_tanks = fuel_required // tank_capacity
fuel_remainder = fuel_required % tank_capacity
communication_delay = distance / 300000
```

Expression	Question answered	Operator
rocket_speed * mission_hours	How far does the rocket travel?	*
fuel_required // tank_capacity	How many complete tank-capacity units fit?	//
fuel_required % tank_capacity	How much fuel remains?	%
distance / 300000	Approximate light-travel time	/

MODELLING NOTE

The communication-delay calculation is deliberately simplified for teaching, using approximately 300,000 km/s for the speed of light. The important programming idea is how existing variables become inputs to new expressions.

THINK / PREDICT

If fuel_required is 950 and tank_capacity is 400, what should full_tanks and fuel_remainder be?

Trace one calculation

```
fuel_required = 950
tank_capacity = 400

full_tanks = fuel_required // tank_capacity
fuel_remainder = fuel_required % tank_capacity

print(full_tanks)
print(fuel_remainder)
```

Predict first. Then run. Explain why // and % form a useful pair.

USEFUL PATTERN

For a quantity divided into equal-sized groups: // tells you how many complete groups fit; % tells you what is left over.

Part E - Produce the mission report

The final stage turns raw values and calculations into information a human can read.

```
print("\n" + "=" * 60)
print(" MISSION PLAN ".center(60, "="))
print("=" * 60)

print("Commander".ljust(28), commander)
print("Mission".ljust(28), mission_name)
print("Destination".ljust(28), destination)

print("-" * 60)

print("Rocket speed".ljust(28), rocket_speed, "km/h")
print("Mission duration".ljust(28), mission_hours, "hours")
```

```
print("Distance travelled".ljust(28), distance, "km")
print("Communication delay".ljust(28), communication_delay, "seconds")
print("Full fuel tanks".ljust(28), full_tanks)
print("Fuel remainder".ljust(28), fuel_remainder, "L")

print("=" * 60)
print(("MISSION " + mission_name + " CONFIGURED").center(60))
print("=" * 60)
```

THINK / PREDICT

What do ljust(28), sep and end do?

Hint: Think about output layout: what appears between values and how text lines end.

You do not need to memorise every formatting trick. The important idea is that print() can communicate multiple values clearly.

Checkpoint - understand before continuing

Question	Your answer
What type does input() return?	
Why do we use float(input(...)) for speed?	
What does // calculate here?	
What does % calculate here?	
Why are commander and rocket_speed different types?	
What is the difference between a variable and a literal?	
What is the INPUT -> PROCESS -> OUTPUT structure?	

DEBUGGING HABIT

When the output is wrong, do not immediately rewrite everything. Ask: Did I get the input? Is its type correct? Did I store the value under the right name? Is the expression correct? Is the output displaying the right variable?

Complete Mission Control Panel v2.0

Use this only after you have built and tested the stages above.

```
# ============================================================
# AUSTRALIAN SPACE MISSION CONTROL - VERSION 2.0
# Week 2: Strings, variables, input, conversion and output
# ============================================================

print("=" * 60)
print(" AUSTRALIAN SPACE MISSION CONTROL ".center(60, "="))
print(" Interactive Mission Planner - v2.0 ".center(60))
print("=" * 60)

# INPUT - mission identity
commander = input("\nCommander name: ")
mission_name = input("Mission name: ")
destination = input("Destination: ")

print("\nWelcome, " + commander + "!")
print("Configuring mission", mission_name, "to", destination + "...")

# INPUT - mission parameters
rocket_speed = float(input("\nRocket speed (km/h): "))
mission_hours = float(input("Mission duration (hours): "))
fuel_required = float(input("Fuel required (litres): "))
tank_capacity = float(input("Fuel tank capacity (litres): "))

# PROCESS
distance = rocket_speed * mission_hours
full_tanks = fuel_required // tank_capacity
fuel_remainder = fuel_required % tank_capacity
communication_delay = distance / 300000

# OUTPUT
print("\n" + "=" * 60)
print(" MISSION PLAN ".center(60, "="))
print("=" * 60)

print("Commander".ljust(28), commander)
print("Mission".ljust(28), mission_name)
print("Destination".ljust(28), destination)

print("-" * 60)

print("Rocket speed".ljust(28), rocket_speed, "km/h")
print("Mission duration".ljust(28), mission_hours, "hours")
print("Distance travelled".ljust(28), distance, "km")
print("Communication delay".ljust(28), communication_delay, "seconds")
print("Full fuel tanks".ljust(28), full_tanks)
print("Fuel remainder".ljust(28), fuel_remainder, "L")

print("=" * 60)
print(("MISSION " + mission_name + " CONFIGURED").center(60))
print("=" * 60)
```

Test the program like a programmer

Do not test only one set of values. Try several missions.

Test	Example input idea	What to investigate
Normal mission	speed 28000, hours 36	Does distance make sense?
Decimal input	speed 28500.5, hours 2.5	Does float() work?
Small fuel amount	fuel 50, tank 400	What do // and % produce?
Exact tank division	fuel 800, tank 400	Is remainder 0?
Different names	your own commander and mission	Are strings displayed correctly?

TESTING INSIGHT
A program that works for one input is not automatically a good program. Change the inputs deliberately and investigate the boundaries and unusual cases.

Mission Control challenges

Challenge 1 - Personalise the dashboard

Modify the dashboard without changing its basic programming ideas.

- Add a mission agency or organisation name.
- Add a mission code.
- Add a launch location.
- Add a short mission description.

THINK / PREDICT
Which new pieces of information are strings? Which would naturally be numbers?

Challenge 2 - Add one new calculation

Choose ONE:

- Calculate average fuel consumption per hour.
- Calculate estimated fuel remaining after the mission.
- Calculate the number of kilometres travelled per minute.
- Calculate an estimated arrival time if the mission starts at a specified hour.

THINK / PREDICT
Before coding, write the formula in plain English. Then translate it into Python.

Challenge 3 - Output designer

Make the report look more professional using only print(), strings, sep, end and simple string methods.

```
print("=" * 60)
print("MISSION CONTROL".center(60))
print("-" * 60)
print("Commander", commander, sep=": ")
print("Mission", mission_name, sep=": ")
print("Destination", destination, sep=": ")
```

THINK / PREDICT

Can you redesign the report without changing the calculations?

Challenge 4 - Find the bug

```
speed = input("Speed: ")
hours = input("Hours: ")
distance = speed * hours
print("Distance:", distance)
```

The program may run, but it does not perform the intended numeric calculation.

4. Identify the type of speed.
5. Identify the type of hours.
6. Explain why * behaves unexpectedly.
7. Fix the program.
8. Test it with 10 and 5.

DEEP INSIGHT

Not every bug is a syntax error. Code can run successfully and still produce the wrong result. This is why understanding types and meaning is more important than merely making Python stop complaining.

Quick workshop games

Game A - Type Detective

```
42
42.0
"42"
float("42")
str(42)
10 / 2
10 // 2
```

For each expression, predict the value AND the type.

Game B - Predict the dashboard

```
x = 10
x += 5
x *= 2
x -= 4
print(x)

print("12" + "3")
print(12 + 3)
print("ha" * 4)
```

THINK / PREDICT

Predict every line before running it.

Game C - One-minute code reading

```
name = input("Name: ")
value = float(input("Value: "))
result = value * 10
print(name, result)
```

Identify, without running it:

- the input variables
- the types
- the processing expression
- the output
- one thing that would change if value were entered as text without float()

Why this workshop matters

The dashboard is becoming a real program because the same code can now work with different missions. That is a major conceptual step: instead of writing a calculation for one specific situation, you are writing a reusable process.

Week	Capability	Mission Control
Week 1	Values, types, operators, expressions, print()	Static mission calculations
Week 2	Strings, variables, assignment, input, conversion, output	User-configured mission planner
Week 3	Booleans and if	Launch and safety decisions

Week 4	while, for, break, continue	Repeated telemetry / mission processing
Week 5	Functions	Reusable mission operations

DESIGN PRINCIPLE

Ask two questions every week: What can my program not do yet? What new Python idea could make that possible?

Week 2 Mission Control checkpoint

- [ ] I can create and run a .py file.
- [ ] I can explain why input() returns str.
- [ ] I can convert numeric input with int() or float().
- [ ] I can choose appropriate variable names.
- [ ] I can explain assignment and reassignment.
- [ ] I can use +=, -=, *= and /= when appropriate.
- [ ] I can use + for numeric addition and string concatenation.
- [ ] I can use // and % in a practical calculation.
- [ ] I can use print() with multiple arguments.
- [ ] I understand sep and end.
- [ ] I can organise a program as INPUT -> PROCESS -> OUTPUT.
- [ ] I can predict and trace variables before running code.
- [ ] I can identify a logical/type error even when the program runs.
- [ ] I can explain every important line in Mission Control v2.0.

Reflection

What is the most important difference between Mission Control v1.0 and v2.0?

Your answer: _____

_____

Why is input() powerful even though it returns a string?

Your answer: _____

_____

Which variable name in your program best communicates meaning?

Your answer: _____

_____

Which calculation uses // and %? Why are both useful?

Your answer: _____

_____

What bug or unexpected result did you encounter? How did you diagnose it?

Your answer: _____

_____

What would you like Mission Control to be able to decide next week?

Your answer: _____

_____

Week 2 takeaway

FROM CALCULATION TO INTERACTIVE PROGRAMMING

Week 1 gave you values, types, operators and expressions.

Week 2 gives those ideas a purpose: strings represent text, variables give data meaningful names, input() brings information into the program, conversion gives it a useful type, expressions process it, and print() communicates the result.

Mission Control v2.0 is therefore not just a bigger program. It demonstrates the core programming pattern you will keep using throughout the course:

```
INPUT  ->  STORE  ->  PROCESS  ->  OUTPUT
```

NEXT WEEK

Mission Control currently accepts whatever the user enters and always produces the same kind of report. Next week, Boolean values and if statements will allow the dashboard to ask questions such as: "Is this mission safe?" and "Should we launch?"