PROGRAMMING FUNDAMENTALS

CHAPTER 2

Strings, Variables, Input and Output

Dr Lei Wang (Griffith University)

CHAPTER THEME

Week 1 taught Python how to calculate.

This chapter teaches Python how to represent text, remember information, receive data from users, transform that data into useful types, and communicate results.

Student Edition | Week 2

Chapter roadmap

A program becomes much more useful when its values are not permanently written into the source code.

In this chapter you move from fixed calculations to programs that can work with text, store values under meaningful names, accept information at run time, and produce personalised output.

Section	What you will learn
1	From Week 1 to Week 2
2	Strings: text is data
3	Combining strings
4	Type conversion
5	Variables and assignment
6	Reassignment and program state
7	Naming variables well
8	Augmented and multiple assignment
9	print(), sep and end
10	input() and runtime data
11	INPUT -> STORE -> PROCESS -> OUTPUT
12	Comments and good programming habits
13	Common mistakes and debugging
14	Practice and exam-style skills
15	Summary and quick reference

THE CENTRAL IDEA

Think of a program as a data journey: INPUT -> STORE -> PROCESS -> OUTPUT.

This pattern will return throughout the course.

Learning outcomes

- Create and recognise string (str) values using single, double and triple quotes.
- Explain why 25 and "25" are different values and types.
- Combine strings using concatenation and explain how + behaves for different types.
- Use int(), float() and str() to convert compatible values.
- Explain variables as meaningful names that let a program refer to values.

- Use valid Python variable names and the snake_case convention.
- Explain assignment, reassignment and augmented assignment.
- Trace a variable's value through a sequence of statements.
- Use multiple assignment and understand Python's value-swapping pattern.
- Use print() with multiple arguments, sep and end.
- Use input() and explain that it returns a string.
- Choose int() or float() appropriately when converting user input.
- Build a short program using INPUT -> STORE -> PROCESS -> OUTPUT.
- Use comments to explain purpose and maintain clear code.
- Predict, test and debug short Python programs.

1. From Week 1 to Week 2

In Week 1, a program could calculate a fixed answer:

```
print(28000 * 36)
```

The calculation is correct, but speed and duration are built into the source code. If the mission changes, the programmer must edit the program.

THINK BEFORE YOU RUN

How could we make the same program work for many missions without rewriting the calculation every time?

Hint: The program needs a way to name information and receive new information.

THE WEEK 2 UPGRADE

Week 2 adds strings, variables, assignment, input, type conversion and richer output.

Together these features let one program work with many different values.

2. Strings: text is data

A string is a sequence of characters used to represent text.

Python's type name for strings is str.

```
print("Mission Control")
print('Rocket ready')
print("123")
print(type("123"))
```

Value	Type	Meaning
42	int	Whole-number value
42.0	float	Floating-point value
"42"	str	Text containing the characters 4 and 2
"Mission"	str	Text

COMMON CONFUSION

Quotation marks matter. 25 is an integer value, while "25" is text.

They may look similar to a human, but Python treats them differently.

THINK BEFORE YOU RUN

What is the type of 123? What is the type of "123"? Why might this difference matter when a user enters data?
Hint: Look carefully for quotation marks.

2.1 Choosing quote styles

```
print("Hello")
print('Hello')
print("I'm ready for launch")
print('The spacecraft is "Aurora"')
```

Single and double quotes both create strings. Choose the style that makes the text easiest to read. The opening and closing quotes must match.

2.2 Triple-quoted strings

```
print("""
==============================
       MISSION BRIEFING
==============================
Destination: Moon
Status: Preparing
==============================
""")
```

Triple quotes can represent multi-line strings. They are useful for multi-line text and, later, documentation strings.

3. Combining strings

The + operator can concatenate strings: it joins strings together. The same symbol performs numeric addition when its operands are numbers.

```
print("Mission" + " " + "Control")
print(12 + 3)
print("12" + "3")
```

Expression	Result	Why?
12 + 3	15	Both operands are numbers, so + adds.
"12" + "3"	"123"	Both operands are strings, so + joins them.
"Mission" + " Control"	"Mission Control"	The strings are concatenated.

KEY INSIGHT

Do not memorise '+ means addition'. A better rule is: the operation depends on the types of the values involved.

THINK BEFORE YOU RUN

Why does 12 + 3 produce 15 while "12" + "3" produces "123"?

3.1 Repeating strings

```
print("ha" * 5)
print("=" * 40)
```

The * operator can repeat a string when one operand is a string and the other is an integer.

4. Type conversion

Sometimes a value has the wrong type for the operation you want. Python provides built-in conversion functions.

```
print(int("42"))
print(float("42"))
print(float("3.14"))
print(str(42))
```

Function	Typical purpose	Example	Result
int()	Convert compatible data to integer	int("42")	42
float()	Convert compatible data to floating point	float("42")	42.0
str()	Convert a value to text	str(42)	"42"

THE TWO-QUESTION RULE

Ask:

(1) What type do I have now?

(2) What type does the next operation require?

THINK BEFORE YOU RUN

What happens when Python evaluates int("rocket")?

Hint: Not every piece of text can be interpreted as an integer.

Conversions can fail when the value is not compatible with the requested type.

5. Variables: giving data meaningful names

A variable is a name that lets a program refer to a value. Meaningful names make programs easier to understand, test and change.

```
rocket_speed = 28000
mission_hours = 36
distance = rocket_speed * mission_hours
print(distance)
```

Read the first line as: 'assign the value 28000 to rocket_speed.' Do not initially read = as mathematical equality.

Part	Example	Role
Name	rocket_speed	Name used by the program
Assignment	=	Stores the result under the name
Value	28000	Data being assigned
Expression	rocket_speed * mission_hours	Produces a value

DESIGN INSIGHT

rocket_speed communicates meaning. A name such as x does not.

Readable variable names are part of good software design.

6. Reassignment and program state

A variable can later refer to a different value. This is called reassignment.

```
x = 12.2
print(x)

x = 100
print(x)
```

The second assignment changes what x currently refers to. The state of a program can change as the program executes.

6.1 The famous x = x + 1

```
x = 10
x = x + 1
print(x)
```

Python first evaluates the expression on the right using the current value of x, then assigns the new result back to x.

Step	Statement	Value of x
1	x = 10	10
2	x = x + 1	11

PROGRAMMING VS ALGEBRA

In algebra, x = x + 1 is impossible as an equality. In programming, = is assignment: it changes program state.

THINK BEFORE YOU RUN

If x starts at 7, what is x after x = x * 3? Explain the evaluation steps.

7. Naming variables well

7.1 Rules for legal names

- A name must start with a letter or underscore.
- After the first character, letters, digits and underscores may be used.
- Spaces are not allowed.
- Symbols such as #, . and * cannot be used in an ordinary variable name.
- Names are case-sensitive: speed, Speed and SPEED are different names.
- Python reserved words, such as class, cannot be used as variable names.

Name	Legal?	Comment
rocket_speed	Yes	Good snake_case name
rocket2	Yes	Legal
_speed	Yes	Legal
2rocket	No	Cannot start with a digit
rocket-speed	No	Hyphen is not allowed in a variable name
class	No	Reserved Python keyword
RocketSpeed	Yes	Legal, but not preferred course style

7.2 Good names make code explain itself

```
# Difficult to understand
x1q3z9ocd = 35.0
x1q3z9afd = 12.50
x1q3p9afd = x1q3z9ocd * x1q3z9afd

# Much clearer
hours = 35.0
rate = 12.50
pay = hours * rate
```

Professional programmers read and modify existing code constantly. Clear names reduce the mental effort required to understand code.

7.3 snake_case convention

```
first_name = "Mia"
room_rate = 50
mission_duration = 36

PI = 3.1412
SPEED_OF_LIGHT = 300000
```

For ordinary variables, lower-case snake_case is a useful Python convention.

Uppercase names often communicate values intended not to change.

8. Assignment shortcuts and multiple assignment

Long form	Augmented form
x = x + 2	x += 2
x = x - 2	x -= 2
x = x * 2	x *= 2
x = x / 2	x /= 2

```
x = 10
x += 2
x *= 3
x -= 4
x /= 2
print(x)
```

THINK BEFORE YOU RUN

Trace x after every line rather than jumping to the final answer.

Hint: Use a table: statement -> value of x -> type of x.

8.1 Multiple assignment

```
x = y = z = "Griffith"
print(x, y, z)

x, y, z = 1, "ab", 3
print(x, y, z)
```

Python can assign several names in one statement.

8.2 A useful Python trick: swapping values

```
left = "Moon"
right = "Earth"
left, right = right, left
print(left, right)
```

WHY THIS IS INTERESTING

Python can swap two values without a temporary variable. Focus on understanding the values before and after the statement.

9. print(): communicating results

print() can receive multiple values and lets you control separators and line endings.

```
speed = 28000
print("Rocket speed:", speed, "km/h")
```

9.1 sep: control the separator

```
print("Earth", "Moon", "Mars", sep=" -> ")
print(3, 8, 2026, sep="/")
print("MISSION", "READY", sep=" | ")
```

sep controls what appears between multiple values supplied to print().

9.2 end: control the line ending

```
print("T-", end="")
print(10)

print("Loading", end="...")
print("complete")
```

By default, print() ends with a newline. The end argument lets you replace that ending.

9.3 Small formatting tricks

```
print("=" * 50)
print("MISSION CONTROL".center(50))
print("=" * 50)
print("Rocket Speed".ljust(25), "28000 km/h")
```

ENRICHMENT

center() and ljust() are useful for attractive terminal output. They are enrichment rather than the main assessed Week 2 ideas.

10. input(): receiving data at runtime

input() pauses the program, displays a prompt, waits for the user to type and press Enter, and returns the entered data as a string.

```
name = input("Who are you? ")
print("Welcome", name)
```

THE MOST IMPORTANT WEEK 2 FACT

input() returns str. Even if the user types digits, the returned value is text until you explicitly convert it.

10.1 The input trap

```
age = input("Enter your age: ")
print(type(age))
print(age * 2)
```

If the user enters 20, age contains the string "20".

Multiplying a string by 2 repeats it, so the result is 2020, not 40.

THINK BEFORE YOU RUN

If the user enters 20, what exactly will the program print?
Hint: Remember: input() gives the string "20".

10.2 Fixing the input trap

```
age = int(input("Enter your age: "))
print(type(age))
print(age * 2)
```

Now int() converts the user's text into an integer before multiplication.

10.3 int() or float()?

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

Data	Likely type	Reason
Crew size	int	Normally a whole number
Rocket speed	float	A measurement may contain a decimal

Mission name	str	Text

MODELLING INSIGHT

Choosing a type is a decision about what the data means, not merely a syntax decision.

11. INPUT -> STORE -> PROCESS -> OUTPUT

This model connects almost everything in Weeks 1 and 2.

```
USER
  |
  v
input()  ->  str
             |
       int() / float()
             |
             v
         VARIABLE
             |
             v
       EXPRESSION
             |
             v
        RESULT
             |
             v
        print()
```

Stage	Question	Typical Python
INPUT	What information do we need?	input()
STORE	What should we call it?	variable = ...
PROCESS	What should we calculate or transform?	expressions and operators
OUTPUT	What should the user see?	print()

WHY THIS MODEL MATTERS

When you see an unfamiliar program, first identify input, what is stored, what processing occurs, and what output is produced.

11.1 Complete interactive example

```
# INPUT
hours = float(input("Type in the hours: "))
rate = float(input("Type in the pay rate: "))

# PROCESS
pay = hours * rate

# OUTPUT
print("The amount of payment is $", pay)
```

The structure is simple but important: receive data, convert it, store it, process it, and communicate the result.

11.2 Mission example

```
# INPUT
rocket_speed = float(input("Rocket speed (km/h): "))
mission_hours = float(input("Mission duration (hours): "))

# PROCESS
distance = rocket_speed * mission_hours

# OUTPUT
print("Distance travelled:", distance, "km")
```

The calculation is familiar from Week 1. The new capability is that the program can now work with values supplied at runtime.

12. Comments, indentation and good habits

```
# Rocket speed in km/h
rocket_speed = 28000

# Calculate distance travelled
distance = rocket_speed * 36
```

Anything after # on that line is ignored by Python.

Good comments explain purpose or reasoning rather than repeating obvious code.

GOOD HABIT
Write code that another person can understand. Meaningful names, clear spacing and useful comments become increasingly valuable as programs grow.

Indentation will become structurally important when decisions and loops are introduced. Develop consistent indentation habits now.

13. Common mistakes you should catch early

Mistake	Correct idea
Thinking 25 and "25" are the same	25 is int; "25" is str.
Forgetting that input() returns str	Convert numeric input with int() or float().
Reading = as mathematical equality	= performs assignment.
Thinking x = x + 1 is impossible	Evaluate the right side first, then assign back to x.
Using 2rocket as a name	A variable name cannot start with a digit.
Assuming speed and Speed are the same	Variable names are case-sensitive.
Expecting "12" + "3" to produce 15	String + string performs concatenation.
Combining incompatible types	Convert values before the operation.
Using unclear variable names	Prefer meaningful names such as mission_duration.

14. Practice: predict, trace, test

A strong programming habit is to predict what code will do before running it, then compare your prediction with Python's behaviour.

Practice 1 - Type detective

```
42
42.0
"42"
"4" + "2"
int("42")
str(42)
10 / 2
10 // 2
```

Expression	Predicted value	Predicted type
42		
42.0		
"42"		
"4" + "2"		
int("42")		
str(42)		

10 / 2		
10 // 2		

Practice 2 - Trace the variable

```
x = 5
x = x + 3
x *= 2
x -= 4
x /= 2
print(x)
```

Statement	Value of x	Type of x
x = 5		
x = x + 3		
x *= 2		
x -= 4		
x /= 2		

EXAM TIP

For tracing questions, use a table and work one line at a time.

Practice 3 - Variable name detective

```
rocket_speed
rocket2
2rocket
rocket-speed
_speed
class
RocketSpeed
rocket_speed
```

First identify legal names. Then identify which legal names follow the preferred course convention.

Practice 4 - Predict exact output

```
name = "Mars"
number = 3
print(name * number)
print("12" + "3")
```

```
print(12 + 3)
print("A", "B", "C", sep="-")
print("Launch", end=" ")
print("Ready")
```

Write the exact output before running the program, including spaces and line breaks.

Practice 5 - Debug the program

```
2mission = "Artemis"
speed = input("Rocket speed: ")
hours = input("Mission hours: ")
distance = speed * hours
print("Distance: " + distance + " km")
```

Problem	Why is it a problem?	How would you fix it?
Variable name		
Type of speed		
Type of hours		
Distance calculation		
Output construction		

Practice 6 - Build a small interactive program

Create a Mission Travel Calculator using only concepts from Weeks 1 and 2.

1. Ask for the mission name.
2. Ask for rocket speed in km/h.
3. Ask for mission duration in hours.
4. Convert numeric input to an appropriate type.
5. Calculate distance = speed * duration.
6. Print a clear personalised report.
7. Use meaningful variable names.
8. Include at least one useful comment.

```
# MISSION TRAVEL CALCULATOR
# INPUT

# PROCESS

# OUTPUT
```

BUILDING STRATEGY

Build INPUT first, PROCESS second, OUTPUT last. Test after each stage.

15. A little Python beyond the core lesson

These short examples expand your curiosity. They are not the main assessed content for this chapter.

```
# Swap values
first = 10
second = 20
first, second = second, first
print(first, second)

# Repeat text
print("ha" * 5)
```

LEARNING BOUNDARY

Seeing a powerful feature does not mean you must master it immediately. Learn to distinguish what you need now from what you are simply curious about.

16. The Week 2 concept map

```
               USER
                 |
                 v
              input()
                 |
                 v
                str
                 |
        int() / float()
                 |
                 v
            VARIABLE
                 |
                 v
            EXPRESSION
                 |
                 v
              RESULT
                 |
                 v
             print()
```

If you can explain this flow in your own words, you understand the central connection between the major Week 2 ideas.

17. Self-test: can you explain these?

What is the difference between 25 and "25"?

Your answer: _____

What type does input() return?

Your answer: _____

Why might float(input(...)) be necessary?

Your answer: _____

What does assignment do?

Your answer: _____

Why is mission_duration a better name than x?

Your answer: _____

What is the difference between x = x + 1 and x += 1?

Your answer: _____

What does sep control in print()?

Your answer: _____

What does end control in print()?

Your answer: _____

Why does "12" + "3" differ from 12 + 3?

Your answer: _____

How would you describe a program as INPUT -> STORE -> PROCESS -> OUTPUT?

Your answer: _____

Why should you predict and test code rather than only looking at the final output?

Your answer: _____

18. Chapter checklist

- [ ] I can identify int, float and str values.
- [ ] I understand that quotation marks change a value into text.
- [ ] I can concatenate strings using +.
- [ ] I can use int(), float() and str() when appropriate.
- [ ] I understand variables and assignment.

- [ ] I can trace reassignment and augmented assignment.
- [ ] I can identify legal and illegal variable names.
- [ ] I use meaningful snake_case names.
- [ ] I know that input() returns str.
- [ ] I can convert keyboard input before numeric arithmetic.
- [ ] I can predict print() output, including sep and end.
- [ ] I can identify INPUT, STORE, PROCESS and OUTPUT.
- [ ] I can spot common type and variable errors.
- [ ] I can write a short interactive Python program using Weeks 1 and 2 concepts.

19. Week 2 quick reference

Concept	Remember	Example
str	Text / characters	"Mission"
int()	Convert compatible data to integer	int("42")
float()	Convert compatible data to float	float("3.14")
str()	Convert a value to text	str(42)
Variable	Meaningful name for a value	mission_hours = 36
Assignment	Evaluate right side, then store	x = x + 1
Augmented assignment	Short update form	x += 1
Concatenation	Join strings	"A" + "B"
input()	Returns user input as str	name = input("Name: ")
print()	Displays output	print("Ready")
sep	Controls text between print arguments	print(1, 2, sep="-")
end	Controls what print ends with	print("Hi", end="!")
snake_case	Preferred ordinary variable style	mission_duration
IPO model	Input -> Store -> Process -> Output	ask -> calculate -> display

20. Chapter summary

Week 2 transforms Python from a mostly fixed calculator into a simple interactive programming language. Strings let programs represent text. Variables give values meaningful names. Assignment and reassignment let program state change. Type conversion lets us move between compatible representations. input() brings data into the program, while print() communicates results back to the user.

THE BIG TAKEAWAY

The important skill is not memorising isolated commands. It is understanding how data is represented, how its type affects operations, how values are stored and changed, and how a program moves data from input through processing to useful output.

21. Looking ahead

Our programs can now accept information, but they still treat every input as if it were acceptable. The next major step is to let a program evaluate a situation and choose what to do.

```
if condition:
    do_something()
```

In the next chapter, Boolean values and if statements will give programs the ability to make decisions. The Mission Control workshop will apply that new capability separately to the evolving dashboard.

22. Further reading and useful resources

Resource	Use it for
Python official documentation	Language features, built-in functions and reference material
Python REPL	Quick experiments and checking small ideas
Development environment	Writing, running and debugging .py programs
Course lecture material	Concept explanations and worked examples
Course workshop book	Hands-on application through the evolving Mission Control project

ONE SENTENCE TO REMEMBER

A useful Python program does not just calculate a value; it receives data, gives that data meaning, processes it, and communicates a result.