PROGRAMMING FUNDAMENTALS

CHAPTER 4

LOOPS AND REPETITION

Dr Lei Wang (Griffith University)

THE BIG TRANSITION

Week 1: CALCULATE
Week 2: INTERACT
Week 3: DECIDE
Week 4: REPEAT

SEQUENCE -> SELECTION -> ITERATION

A loop is not simply a way to save typing.

It lets a program process many values, keep track of changing state, respond to repeated input, and stop for a meaningful reason.

CORE MENTAL MODEL

A loop repeatedly executes a block while following a control rule.
Good loop design answers three questions: WHAT repeats? WHAT changes? WHEN does it stop?

Chapter Roadmap

Section	What you will learn
1. From decisions to repetition	Why loops are needed and how Week 3 conditions return.
2. while loops	Indefinite repetition, state, updates and termination.
3. Infinite loops	How they happen and how to reason about them.
4. Input validation	A practical and common use of while.
5. break, continue and pass	Three statements with very different meanings.
6. for loops	Definite iteration through values.
7. range()	Generating integer sequences with start, stop and step.
8. Loop patterns	Counting, totals, filtering, searching and largest-so-far.
9. Nested loops	Repetition inside repetition and simple patterns.
10. Mission Control v4 (our workshop)	Combining Weeks 1 to 4 in one application.
11. Debugging and exam skills	Tracing, boundary cases and common traps.

What You Will Learn

- Explain why loops are useful instead of copying statements repeatedly.
- Distinguish while loops from for loops.
- Write a while loop with correct initialisation, condition and update.
- Recognise and repair common infinite-loop errors.
- Use while for input validation and repeated interaction.
- Use break to leave a loop immediately.
- Use continue to skip the remainder of one iteration.
- Use pass as a placeholder that deliberately does nothing.
- Write for loops over sequences and values.
- Use range(start, stop, step), including reverse ranges.
- Understand that the stop value in range is excluded.
- Combine loops with if statements and Boolean expressions.
- Apply counting, summing, filtering, searching and largest-so-far patterns.
- Trace counters, accumulators and other changing program state.
- Read and write simple nested loops.
- Test first, last and boundary cases.
- Explain how Mission Control v4 extends the Week 3 decision engine.

From One Decision to Repeated Decisions

In Chapter 3, your program learned to ask a Boolean question and choose a path. That is selection. But many real programs need to make the same kind of decision many times.

```
# Week 3: one decision
temperature = float(input("Temperature: "))

if temperature > 80:
    print("WARNING")
else:
    print("SAFE")
```

Suppose Mission Control receives five sensor readings. Copying the code five times would work, but it is repetitive and difficult to maintain.

```
# Repetition without a loop
print("Check sensor 1")
print("Check sensor 2")
print("Check sensor 3")
print("Check sensor 4")
print("Check sensor 5")
```

THE WEEK 4 UPGRADE

Instead of writing the same instructions again and again, describe the repetition once and let Python control how many times it happens.

Week	New capability	Typical question
1	Calculate	What is the result?
2	Interact	What data did the user provide?
3	Decide	What should happen because of that data?
4	Repeat	How can I do this useful work again and again?

IMPORTANT CONNECTION

A while loop reuses the Boolean thinking from Week 3.

An if asks a condition once; while can ask the condition repeatedly.

while Loops: Indefinite Repetition

Use while when repetition is controlled by a condition and you do not necessarily know in advance how many iterations will be needed.

```
i = 0

while i < 5:
    print(i)
    i += 1
```

A reliable mental model is:

THE WHILE LOOP RECIPE
INITIALISE -> TEST -> DO WORK -> UPDATE -> TEST AGAIN

Part	Example	Purpose
Initial state	i = 0	Give the program a starting state.
Condition	i < 5	Decide whether another iteration is allowed.
Work	print(i)	Do the repeated task.
Update	i += 1	Change state so the loop can eventually stop.

THINK BEFORE YOU RUN
Will 5 be printed? How many times does the loop body execute?
Hint: Trace i as 0, 1, 2, 3, 4, then test 5 < 5.

KEY INSIGHT
The loop condition is checked before each iteration. When it becomes False, the loop body is skipped and execution continues after the loop.

A Useful while Example: Countdown

```
count = 5

while count > 0:
    print(count)
    count -= 1

print("LIFTOFF!")
```

THINK BEFORE YOU RUN
Will 0 be printed? What changes if count -= 1 becomes count += 1?
Hint: Ask whether the update moves count toward the condition becoming False.

DIRECTION MATTERS
For while count > 0, the value should move downward toward 0. If your update moves away from the stopping condition, the loop may never end.

Infinite Loops: A Logic Bug

An infinite loop is a loop that keeps executing because its stopping condition is never reached. The syntax may be completely valid; the logic is what is wrong.

```
i = 1

while i != 100:
    print(i)
    i += 2
```

THINK BEFORE YOU RUN
Can i ever equal 100?
Hint: Starting at 1 and adding 2 produces only odd numbers: 1, 3, 5, 7, ...

DEBUGGING QUESTION
Do not only ask 'What does this loop do?'.

Ask 'What values can the changing variable actually take, and can any of them make the condition False?'

while for Input Validation

One of the most practical uses of while is asking again when the user gives invalid data.

You usually do not know how many attempts will be needed.

```
fuel = float(input("Fuel level (0-100): "))

while fuel < 0 or fuel > 100:
    print("Invalid fuel level.")
    fuel = float(input("Fuel level (0-100): "))

print(f"Accepted: {fuel}%")
```

The condition says: keep repeating while the value is outside the valid range.

WHY NOT if?

An if can reject one invalid value, but it does not automatically ask again.

while is appropriate when the number of future attempts is unknown.

THINK BEFORE YOU RUN

What happens if the user enters -5, then 120, then 75?
Hint: The loop repeats for the first two invalid values and stops when 75 satisfies the valid range.

while True and an Explicit Stop

Sometimes the cleanest design is to create a loop that is intentionally open-ended and put the stopping event inside it.

```
while True:
    command = input("Mission command: ")

    if command == "launch":
        print("Launch command accepted.")
        break

    print("Command not recognised.")

print("Control sequence finished.")
```

MENTAL MODEL

while True means 'keep going unless something inside the loop tells us to leave'.
break means 'leave this loop now'.

This pattern is especially useful for menus, command systems and repeated user interaction.

break, continue and pass

Statement	What it does	What it does NOT do
break	Immediately exits the current loop.	It does not just skip one iteration.
continue	Skips the rest of the current iteration and starts the next iteration.	It does not end the loop.
pass	Does nothing; execution continues normally.	It does not control loop repetition.

break

```
counter = 0

while True:
    number = int(input("Enter a positive number (0 or less to stop): "))

    if number <= 0:
        break

    counter += 1

print(f"You entered {counter} positive numbers.")
```

continue

```
for n in range(1, 11):
    if n % 2 == 0:
        continue

    print(n, end=" ")
```

Output: 1 3 5 7 9

BREAK VS CONTINUE
break = leave the LOOP.
continue = leave the CURRENT ITERATION and move to the next one.

pass

```
temperature = 75

if temperature > 80:
    pass  # warning logic will be added later

print("System continues.")
```

PASS IS A PLACEHOLDER
pass is useful when Python requires an indented statement but you deliberately want no action yet.

for Loops: Definite Iteration

Use for when you want to visit each value in a sequence or other iterable. The loop variable changes automatically as Python moves through the values.

```
for student in ["David", "Luke", "Bob"]:
    print("Hello", student)
```

```
for n in [2, 3, 5, 7]:
    print(n, end=" ")
```

READ IT IN PLAIN ENGLISH
For each value in the sequence, put that value into the loop variable and run the indented block.

while	for
Repeat while a condition remains True.	Visit each value in a sequence/iterable.
Useful when the number of repetitions is not known in advance.	Useful when the values/items to process are known.
You usually manage changing state yourself.	The loop variable advances automatically.

range(): Generate Integer Sequences

range() is one of the most important tools used with for loops. It represents a sequence of integers without requiring you to write every value manually.

Expression	Values visited
range(5)	0, 1, 2, 3, 4
range(2, 6)	2, 3, 4, 5
range(2, 10, 2)	2, 4, 6, 8
range(5, 0, -1)	5, 4, 3, 2, 1

THE RANGE RULE
range(start, stop, step)

start is included.
stop is EXCLUDED.
step controls how the value changes.

If start is omitted, it defaults to 0. If step is omitted, it defaults to 1.

THINK BEFORE YOU RUN
What does range(3, 10, 2) produce?
Hint: Start at 3, add 2 each time, and stop before reaching 10.

```
for i in range(3, 10, 2):
    print(i, end=" ")
```

Output: 3 5 7 9

MOST COMMON RANGE TRAP
range(5) does NOT include 5. The values are 0 through 4.

The Most Useful Loop Patterns

Learning a few patterns is much more useful than memorising isolated loop examples. These patterns appear repeatedly in programming problems.

Counting

```
count = 0

for n in range(1, 11):
    if n % 2 == 0:
        count += 1

print(f"Even numbers: {count}")
```

The variable count records how many values satisfy the condition.

Accumulating a Total

```
total = 0

for n in range(1, 6):
    total += n

print(total)
```

The accumulator starts with the identity value 0 and grows as the loop processes values.

Filtering

```
for n in range(1, 11):
    if n % 2 == 1:
        print(n, end=" ")
```

The loop visits all values; the if decides which values are allowed through.

Searching

```
found = False

for n in range(1, 101):
    if n % 17 == 0:
        found = True
        break
```

```
print(found)
```

SEARCH PATTERN
found = False -> inspect candidates -> if target found: found = True and optionally break.

Finding the Largest Value

```
values = [14, 7, 29, 11, 23]

largest = values[0]

for value in values:
    if value > largest:
        largest = value

print(largest)
```

WHY START WITH values[0]?
It gives the algorithm a real value from the data. This is safer than guessing a starting number such as -1 when the data might contain smaller values.

Combining Weeks 1 to 3 Inside a Loop

A loop becomes powerful when it contains the ideas you already know: input, conversion, arithmetic, Boolean expressions and if statements.

```
safe_readings = 0
warning_readings = 0

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

    if temperature < 0:
        print("Invalid reading - skipped.")
        continue

    if temperature > 80:
        print("WARNING: temperature too high")
        warning_readings += 1
    else:
        print("Sensor within safe range")
        safe_readings += 1

print(f"Safe readings: {safe_readings}")
print(f"Warnings: {warning_readings}")
```

THE COURSE SO FAR
Week 1: values and operators

Week 2: input, variables, conversion and output
Week 3: Boolean logic and decisions
Week 4: repeated processing, counting and filtering

This is the point where the course starts to feel like real programming: the program can process a stream of values rather than one isolated input.

Nested Loops: Repetition Inside Repetition

A nested loop is a loop inside another loop. The inner loop completes all of its iterations for each iteration of the outer loop.

```
for row in range(3):
    for column in range(4):
        print("*", end="")
    print()
```

There are 3 rows and 4 stars per row, so 3 x 4 = 12 stars are printed.

```
n = 5

for row in range(1, n + 1):
    for column in range(row):
        print("#", end="")
    print()
```

Output:

```
#
##
###
####
#####
```

NESTED LOOP MENTAL MODEL

Outer loop = the larger structure, such as rows.
Inner loop = repeated work inside each outer iteration, such as columns.

Designing a Loop Algorithm

LOOP DESIGN RECIPE

1. What should repeat?
2. Do I know the number of values/items in advance?
3. Choose for or while.
4. What state should exist before the loop? (count, total, found, largest...)
5. What decision happens inside each iteration?

6. What changes after each iteration?
7. What makes the loop stop?
8. Do I need break or continue?
9. What boundary cases should I test?
10. Trace a few iterations manually before trusting the program.

This recipe is particularly useful for assessment problems.

Before writing Python, describe the algorithm in ordinary language.

Mission Control v4: From One Decision to Continuous Monitoring

Mission Control v3 could evaluate a mission once.

Version 4 uses loops to repeatedly monitor sensor data, apply Week 3 decisions, count outcomes and respond to commands.

```
print("=" * 58)
print("MISSION CONTROL v4.0 - SENSOR MONITOR".center(58))
print("=" * 58)

safe_readings = 0
warning_readings = 0

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

    if temperature < 0:
        print("Invalid reading - skipped.")
        continue

    if temperature > 80:
        print("WARNING: temperature too high")
        warning_readings += 1
    else:
        print("Sensor within safe range")
        safe_readings += 1

print("-" * 58)
print(f"Safe readings: {safe_readings}")
print(f"Warnings: {warning_readings}")
print("=" * 58)
```

Notice the architecture: the for loop controls repetition, input brings in data, if makes a decision, continue handles an invalid reading, and counters accumulate information for the final report.

Mission Control v4.1: Monitor Until Shutdown

```
while True:
    command = input("Command [scan/shutdown]: ")

    if command == "shutdown":
```

```
        print("Mission Control shutting down.")
        break

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

    temperature = float(input("Temperature: "))

    if temperature > 80:
        print("ALERT")
    else:
        print("SAFE")
```

WHY TWO DIFFERENT LOOPS?

Use for when the number/items are known, such as five sensor positions.
Use while when the stopping event is unknown, such as an operator deciding when to shut the system down.

break, continue and pass: Side by Side

Code idea	Effect
if condition: break	Stop the entire current loop.
if condition: continue	Skip the remaining statements in this iteration.
if condition: pass	Do nothing and continue normally.

```
for n in range(1, 6):
    if n == 2:
        continue
    if n == 5:
        break
    print(n, end=" ")
```

THINK BEFORE YOU RUN

What is printed?
Hint: n=2 is skipped; n=5 stops the loop before print(5).

TRACE THE CONTROL FLOW

When you see break or continue, do not just read the next line. Ask where execution jumps.

Common Loop Mistakes

Mistake	Why it happens	Better habit
Forgetting the while update	The condition never becomes False.	Identify the state change before coding.
Wrong update direction	The loop moves away from termination.	Ask whether each update moves toward the stopping condition.
Off-by-one error	Misreading range or boundary conditions.	Trace the first and last values.
Thinking range stop is included	Assuming range(1, 6) includes 6.	Remember: stop is excluded.
Using break when continue is needed	Confusing loop exit with iteration skip.	Say the meaning aloud before choosing.
Using continue inside a while without updating state	The next iteration repeats the same state.	Ensure state changes before the loop condition is tested again.
Using pass as if it skips an iteration	pass sounds like 'pass over'.	Remember: pass does nothing.
Changing a for loop variable manually	Importing while-loop thinking into for.	Let for control its loop variable unless there is a specific reason.
Wrong indentation in nested loops	Not seeing which block belongs where.	Indent and trace the outer and inner blocks separately.

Trace Before You Run

Tracing is one of the most useful skills for quizzes and exams.

Write down the changing state at each iteration.

```
total = 0

for i in range(1, 5):
    total += i
    print(i, total)
```

Iteration	i	total before	total after
1	____	____	____
2	____	____	____
3	____	____	____

4	___	___	___

EXAM HABIT
For a loop question: identify the initial values, list the loop values, update the state one iteration at a time, then determine the final output.

Boundary and Edge-Case Testing

Loops often contain conditions, so boundary testing remains important. The best tests depend on the stopping or filtering rule.

Rule	Useful tests
while x < 10	x = 9, x = 10, and a value below 9
if n % 5 == 0	4, 5, 6
range(1, 6)	Check first value 1 and excluded stop 6
repeat until 0	Positive value, 1, then 0
find a target	Target first, target last, target absent

THINK LIKE A TESTER
Ask what happens when there are zero valid values, exactly one value, all values valid, no match, or the stopping value appears immediately.

A Useful Pattern: Read Until a Sentinel

A sentinel is a special value that means 'stop'. This pattern appears frequently in programming problems.

```
total = 0
count = 0

number = int(input("Enter a number (0 to stop): "))

while number != 0:
    total += number
    count += 1
    number = int(input("Enter a number (0 to stop): "))

print(f"Count: {count}")
print(f"Total: {total}")
```

The sentinel 0 is not part of the data being accumulated. It is a control signal.

THINK BEFORE YOU RUN

What happens if the user enters 0 immediately?

Hint: The loop body executes zero times, so count and total remain at their initial values.

Practice: Translate Problems into Loop Patterns

Problem A - Count Multiples

Write a program that asks for a positive integer n and counts how many numbers from 1 through n are divisible by 5.

```
# Your design:
# 1. What repeats?
# 2. for or while?
# 3. What variable stores the answer?
# 4. What condition identifies a multiple of 5?
```

Problem B - Sum Until Zero

Repeatedly ask the user for numbers until 0 is entered. Print the total of the non-zero numbers.

```
# Your design:
# 1. What is the sentinel?
# 2. What must be initialised before the loop?
# 3. What changes each iteration?
# 4. What makes the loop stop?
```

Problem C - Search

Ask for a target number and search through 1 to 100. Stop as soon as the target is found.

```
# Think:
# for n in range(1, 101):
#     ...
# When would break be useful?
```

THE TRANSFERABLE SKILL

The challenge is not to recognise a familiar example.

It is to identify the underlying pattern: count, accumulate, filter, search or repeat-until.

Week 1 to Week 4: The Growing Programming Model

Week	Capability	Typical building block
1	CALCULATE	operators, expressions, types
2	INTERACT	variables, input, conversion, output
3	DECIDE	Boolean expressions, if/elif/else
4	REPEAT	while, for, range, break, continue

THE BIG PICTURE

A program can now receive data, process it, make decisions about it, repeat those decisions, keep running totals or counts, search for information and stop when appropriate.

Quiz and Exam Focus

Be particularly confident with:

- Tracing the exact output of a while or for loop.
- Determining how many times a loop executes.
- Understanding range(start, stop, step) and the excluded stop value.
- Predicting reverse ranges and negative steps.
- Recognising infinite-loop logic.
- Explaining the practical difference between while and for.
- Tracing break and continue correctly.
- Distinguishing pass from continue.
- Following indentation in nested loops.
- Tracing counter and accumulator variables.
- Recognising filtering, searching, counting and summing patterns.
- Understanding largest-so-far initialisation.
- Checking boundary values and variable updates.

EXAM STRATEGY

Do not mentally 'run' a long loop all at once.

Make a small table of the important variables and update it one iteration at a time.

Self-Test: Can You Explain These?

1. Why do we need loops?

Answer: _____

2. What is the difference between while and for?

Answer: _____

3. What three ingredients are central to a basic while loop?

Answer: _____

4. Why can a while loop become infinite?

Answer: _____

5. Why is while useful for input validation?

Answer: _____

6. What does break do?

Answer: _____

7. What does continue do?

Answer: _____

8. What does pass do?

Answer: _____

9. What is the difference between break and continue?

Answer: _____

10. What values does range(5) produce?

Answer: _____

11. Is the stop value included in range?

Answer: _____

12. What does range(2, 10, 2) produce?

Answer: _____

13. How do you make range count backwards?

Answer: _____

14. What is a counter variable?

Answer: _____

15. What is an accumulator?

Answer: _____

16. How does filtering with if inside a loop work?

Answer: _____

17. How can you search for a target?

Answer: _____

18. Why is largest = values[0] often safer than largest = -1?

Answer: _____

19. How many times does an inner loop execute in a nested loop?

Answer: _____

20. When would you choose several separate if statements rather than one if/elif/else chain?

Answer: _____

21. How does Mission Control v4 differ from v3?

Answer: _____

Chapter Concept Map

Concept	Role	Example
while	Repeat while a condition is True	while fuel < 0:
for	Visit each value in a sequence	for n in values:
range()	Generate integer sequences	range(1, 6)
break	Exit the loop	if done: break
continue	Skip current iteration	if invalid: continue
pass	Do nothing / placeholder	if future_case: pass
counter	Count qualifying events	count += 1
accumulator	Build a running result	total += value
filter	Select values satisfying a condition	if n % 2 == 0:
search	Stop when a target is found	if value == target: break
largest-so-far	Track best value seen so far	if value > largest:
nested loop	Repeat an inner process for each outer iteration	for row ... for col ...

Chapter Checklist

- [ ] I can explain why loops are useful.
- [ ] I can distinguish indefinite repetition from definite iteration.
- [ ] I can write a basic while loop.
- [ ] I can identify initialisation, condition, work and update.
- [ ] I can recognise an infinite loop.
- [ ] I can use while for input validation.
- [ ] I can use while True with break when appropriate.
- [ ] I can use break correctly.
- [ ] I can use continue correctly.
- [ ] I can explain what pass does.
- [ ] I can write a for loop over a sequence.

- [ ] I can use range() with one, two and three arguments.
- [ ] I remember that range stop is excluded.
- [ ] I can use a negative step.
- [ ] I can count qualifying values.
- [ ] I can accumulate a total.
- [ ] I can filter values using if.
- [ ] I can search for a value and use break.
- [ ] I can track the largest value seen so far.
- [ ] I can trace a nested loop.
- [ ] I can combine Weeks 1 to 3 concepts inside a loop.
- [ ] I can test boundary and edge cases.
- [ ] I can explain Mission Control v4.

Summary

Week 4 changes the scale of your programs. Instead of performing one calculation, accepting one input, or making one decision, your program can now repeat useful work.

while is appropriate when repetition depends on a condition and the number of iterations is not known in advance. for is appropriate when you want to visit values in a sequence. range() gives for loops convenient integer sequences, with an important rule: the stop value is excluded.

break, continue and pass are deliberately different. break exits the loop, continue skips the remainder of the current iteration, and pass does nothing. Combining loops with if statements produces powerful patterns such as counting, summing, filtering, searching and finding the largest value.

The deeper skill is learning to reason about changing program state. A loop is correct only when its repeated work, state changes and stopping rule all make sense.

ONE SENTENCE TO REMEMBER
Iteration means revisiting a process under control: know what repeats, know what changes, and know why the loop stops.

Looking Ahead: Functions

By the end of Week 4, some of your programs may contain blocks of logic that you want to reuse. That leads naturally to functions.

```
# Week 4: repeated logic
for sensor_number in range(1, 6):
    temperature = float(input(f"Sensor {sensor_number}: "))
    if temperature > 80:
        print("WARNING")
    else:
        print("SAFE")
```

Next, you will learn how to package reusable logic into a named function.

```
def check_temperature(temperature):
    if temperature > 80:
```

```
        return "WARNING"
    return "SAFE"
```

NEXT STEP
Week 4 teaches REPEAT.
Week 5 will teach ORGANISE AND REUSE.

The goal is not just shorter code. Functions help give a program structure, reuse logic and make larger problems easier to manage.

Final Reflection

What is the biggest difference between if and while?

Your reflection: _____

When would you choose for instead of while?

Your reflection: _____

Which loop pattern do you find most useful: counting, summing, filtering, searching or largest-so-far?

Your reflection: _____

What was the most surprising thing about range()?

Your reflection: _____

Can you explain break, continue and pass without looking at your notes?

Your reflection: _____

What is one infinite-loop mistake you now know how to diagnose?

Your reflection: _____

How did Mission Control change from v3 to v4?

Your reflection: _____

What repeated logic in your programs might become a function next week?

Your reflection: _____