Chapter 4

Loops and Iteration

1. Learning objectives

By the end of this chapter, students should be able to:

- Explain why loops are useful in programming.
- Identify situations where repetition is needed.
- Use `while` loops to repeat instructions while a condition is true.
- Develop counter-controlled loops.
- Use `for` loops to repeat instructions a specified number of times.
- Use `range()` to generate sequences of numbers.
- Use nested loops to solve simple problems involving repeated operations.
- Control loop execution using `break` and `continue`.
- Choose an appropriate loop structure for a given problem.
- Recognize and prevent common loop errors, including infinite loops and off-by-one errors.

2. Introduction

Many programming problems require the same operation to be performed several times.

For example, an engineer may need to:

- Calculate the temperature difference for ten measurements.
- Display a table of conversion values.
- Calculate the total cost of several components.
- Check a machine reading repeatedly until it reaches a specified limit.

One approach is to write the same instruction several times. However, this makes programs unnecessarily long and difficult to maintain. Python provides **loops**, which allow a group of instructions to be executed repeatedly.

Consider the following program:

```
print(5)
print(10)
print(15)
print(20)
print(25)
```

The program displays five multiples of 5. Although it works, the same type of instruction is repeated five times.

A loop provides a more efficient solution:

```
for number in range(1, 6):
    print(number * 5)
```

Both programs produce the same output:

```
5
10
15
20
25
```

The second program is shorter and easier to modify. For example, displaying the first 100 multiples of 5 requires only a small change. In this chapter, we will learn how to use loops to make programs more efficient, readable, and flexible.

3. Concepts

3.1. Loops

A loop is a programming structure that repeats a block of instructions.

Loops are useful when:

1. An operation must be repeated a fixed number of times.
2. An operation must continue until a condition is satisfied.
3. A program must process a sequence of values.
4. A calculation must be performed repeatedly using different inputs.

In Python, we can distinguish two types of loop statements **Counter-Controlled loop** in which the statements are repeated with a specified number of times and **conditional loop** where the statements are executed until reaching defined goal or conditions become false. The **`while`** statement is usually adopted for conditional loop and the counted loop is based in general on the **`for`** statement. We present in what follows the **`while`** and the **`for`** statements.

3.2. The `while` loop

A while loop repeats a block of instructions as long as a specified condition is true. The general form of the while loop is defined as illustrated bellow.

```
while condition:
    statements
```

The condition is evaluated before each iteration. If the condition is initially false, the loop body is not executed.

For example, consider a program that displays the numbers from 1 to 5.

```
number = 1

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

Output:

```
1
2
3
4
5
```

Let us examine how the program works.

Step 1 **Initialization**	`number = 1`	The variable number starts at 1.
Step 2 **Condition checking**	`while number <= 5:`	Python checks whether number is less than or equal to 5. If the condition is true, the loop body is executed.
Step 3 **Repetition**	`print(number)` `number = number + 1`	The current value is displayed, and then number is increased by 1.
Step 4 **Termination**		When number becomes 6, the condition number <= 5 is false. The loop stops.

This example illustrates three important elements of a counter-controlled while loop:

- **Initialization:** Establish the initial value.
- **Condition:** Determine whether repetition should continue.
- **Update:** Modify the variable so that the loop can eventually terminate.

If the condition is initially false, the loop body is not executed.

For example:

```
temperature = 20

while temperature > 100:
    print("Temperature is too high.")
```

There is no output because the condition is false from the beginning.

3.3. Counter-controlled loops

A counter-controlled loop repeats a block of instructions a known number of times. For example, suppose a machine performs five identical test cycles. We can use a counter to represent the current cycle.

```
cycle = 1

while cycle <= 5:
    print("Test cycle:", cycle)
    cycle = cycle + 1
```

Output:

```
Test cycle: 1
Test cycle: 2
Test cycle: 3
Test cycle: 4
Test cycle: 5
```

Counter-controlled loops are particularly useful when the number of repetitions is known. However, Python also provides a convenient alternative for this situation: the **`for`** loop.

3.4.Infinite loops

An infinite loop is a loop that does not terminate because its continuation condition remains true indefinitely, or because the program does not reach a termination condition.

Consider this example:

```
number = 1

while number <= 3:
    print(number)
```

The output begins:

```
1
1
1
1
...
```

The value of `number` never changes. Therefore, the condition `number <= 3` remains true. This is a common beginner mistake and the corrected program is:

```
number = 1

while number <= 3:
    print(number)
    number += 1
```

Now the variable increases after each iteration, and the loop terminates when `number` becomes `4`.

Before running a `while` loop, ask yourself:

1. What is the initial value?
2. What condition controls the loop?
3. Which instruction changes the relevant variable?
4. Why will the condition eventually become false?

3.5. The for Loop

In general, we use the for loop to repeat a set of statements with a known number of iterations. The general form of the for loop is given as indicated below.

```
for variable in sequence:
    statements
```

The loop `variable` receives an item from the `sequence` during each iteration. The sequence may be generated using `range()` or another iterable object, such as a string. Lists will be studied in Chapter 6. For now, we will concentrate on numerical iteration using `range()`.

3.6. The `range()` Function

The `range()` function generates a sequence of integers and it is frequently used with `for` loops. The `range()` function has three common forms.

A. `range(stop)`

```
for number in range(5):
    print(number)
```

Output:

```
0
1
2
3
4
```

The sequence begins at 0 and stops before reaching 5. Therefore, `range(5)` generates five integers: 0, 1, 2, 3, and 4 where the stop value 5 is excluded.

B. `range(start, stop)`

```
for number in range(2, 5):
    print(number)
```

Output:

```
2
3
4
```

The sequence begins at 2 and stops before 5.

C. `range(start, stop, step)`

The third argument specifies the step, which determines how the values change between iterations.

```
for number in range(0, 11, 2):
    print(number)
```

Output:

```
0
2
4
6
8
10
```

The step is 2, so the generated values increase by 2.

A negative step can be used to generate decreasing values:

```
for number in range(10, 0, -3):
    print(number)
```

Output:

```
10
7
4
1
```

Here, the sequence begins at 10 and decreases by 3. It stops before reaching 0.

3.7.Nested Loops

A nested loop is a loop placed inside another loop. Nested loops are useful when a problem involves two levels of repetition.

For example, suppose a program must display the number of the row and the number of the column in a small grid. The outer loop controls the rows, while the inner loop controls the columns.

```
for row in range(1, 3):
    for column in range(1, 4):
        print("Row:",row, "Column:",column)
```

Output:

```
Row: 1 Column: 1
Row: 1 Column: 2
Row: 1 Column: 3
Row: 2 Column: 1
Row: 2 Column: 2
Row: 2 Column: 3
```

The outer loop runs two times, where for each iteration of the outer loop, the inner loop runs three times. Therefore, the inner statement executes 2 × 3 = 6 times. The total number of executions depends on both loops.

3.8. The break Statement

In general, we use the `break` statement to jump out of the loop. Consider the following program:

```
for number in range(1, 10):
    print(number)
    if number == 5:
        break
```

Output:

```
1
2
3
4
5
```

When `number` becomes `5`, the `break` statement terminates the loop. The number `5` is printed because the condition is checked after the `print()` statement.

3.9. The continue Statement

The `continue` statement skips the remaining instructions in the current iteration and proceeds to the next iteration.

For example, suppose we want to display the numbers from 1 to 5 except 3.

```
for number in range(1, 6):
    if number == 3:
        continue
    print(number)
```

Output:

```
1
2
4
5
```

When `number` equals `3`, Python executes `continue`. The `print()` statement is skipped for that iteration. The loop then continues with the next value.

3.10. Comparing while and for

Both `while` and `for` can repeat instructions, but they are often used in different situations. A `while` loop is appropriate when repetition depends on a condition that may change during execution. A `for` loop is often more convenient when iterating through a known range of values or processing each item in a sequence.

Consider the following comparison where the both programs produce the same output:

`while` loop:

```
number = 1

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

`for` loop:

```
for number in range(1, 6):
    print(number)
```

However, the for version is shorter because Python manages the iteration automatically. The following table summarizes the main differences.

Table 4.1 Main differences between `while` *and* `for`

Feature	while	for
Main purpose	Repeat while a condition is true	Iterate over a sequence or range
Number of iterations	May be unknown in advance	Often known or determined by the iterable
Initialization	Often required for counter-controlled loops	Usually not required for the iteration variable
Updating the iteration variable	Often performed manually	Managed automatically by the loop
Risk of infinite loops	High if the condition never becomes false	Lower for ordinary finite ranges
Common applications	Repeating until a condition changes	Counting, tables, and processing sequences

Neither loop is universally better than the other. The choice depends on the problem.

Rule of thumb:

- Use `for` when iterating over a known range or sequence.
- Use `while` when repetition must continue until a condition changes.

4. Syntax Reference

This section summarizes the main loop structures introduced in the chapter.

`while`

```
while condition:
    statements
```

The statements execute repeatedly while the condition is true.

Counter-Controlled `while`

```
counter = initial_value

while condition:
    statements
    counter += increment
```

The counter must be updated appropriately so that the loop can terminate.

`for`

```
for variable in sequence:
    statements
```

The loop executes once for each item in the sequence.

`range()`

```
range(stop)
range(start, stop)
range(start, stop, step)
```

The stop value is excluded in all three forms.

Nested Loops

```
for outer_variable in outer_sequence:
    for inner_variable in inner_sequence:
        statements
```

The inner loop completes its iterations for each iteration of the outer loop.

`break`

```
for variable in sequence:
    if condition:
        break
```

The nearest enclosing loop terminates immediately.

`continue`

```
for variable in sequence:
    if condition:
        continue
    statements
```

The remaining statements in the current iteration are skipped.

5. Worked Examples

Example 1 - Calculating the sum of integers

This program calculates the sum of all integers from 1 to 10.

```
total = 0
   for number in range(1, 11):
       total += number

   print("Sum:", total)
```

Output:

```
Sum: 55
```

Example 2 - Square of numbers

```
for itr in range(1, 4):
    sq = itr**2;
    print("The square of", itr , "is" , sq )
```

Output:

```
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
```

Example 3 - Password

```
password = ""
while password != "1234":
    password = input("Enter password: ")

print("Welcome!")
```

Possible output:

```
Enter password: 1235
Enter password: 4321
Enter password: 2589
Enter password: 1234
Welcome!
```

Example 3 - Repeated user input

The following program asks the user to enter a positive number. The program continues asking until the user enters a positive value.

```
number = float(input("Enter a positive number: "))
while number <= 0:
    print("The number must be positive.")
    number = float(input("Enter a positive number: "))

print("Accepted value:", number)
```

Possible execution

```
Enter a positive number: -50
The number must be positive.
Enter a positive number: 0
The number must be positive.
Enter a positive number: 14.5
Accepted value: 12.5
```

6. Common Mistakes

Beginners frequently encounter problems when writing loops. Understanding these errors helps prevent unexpected behavior.

1. **Forgetting to update a while loop counter**

Incorrect:

```
number = 1

while number <= 5:
    print(number)
```

The value never changes, so the condition remains true.

Correct:

```
number = 1

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

2. Using the wrong stop value in `range()`

Consider:

```
for number in range(1, 5):
    print(number)
```

Output:

```
1
2
3
4
```

The number 5 is not included. Always remember that the stop value is excluded. To display the integers from 1 to 5, we write:

```
for number in range(1, 6):
    print(number)
```

3. Incorrect Indentation

Incorrect:

```
for number in range(1, 4):
print(number)
```

Python expects the loop body to be indented. Indentation identifies the statements belonging to the loop.

Correct:

```
for number in range(1, 4):
    print(number)
```

4. Initializing an accumulator incorrectly

Incorrect:

```
total = 10

for number in range(1, 4):
    total = total + number

print(total)
```

The program displays 16 because the initial value is 10. If the purpose is to calculate the sum of 1, 2, and 3, the accumulator should begin at zero.

Correct:

```
total = 0

for number in range(1, 4):
    total = total + number

print(total)
```

Output:

```
6
```

An accumulator should be initialized according to the calculation being performed.

5. Confusing `break` and `continue`

The `break` statement terminates the nearest enclosing loop.

The `continue` statement skips the remaining statements in the current iteration and proceeds to the next iteration.

For example:

```
for number in range(1, 6):
    if number == 3:
        continue

    print(number)
```

Output:

```
1
2
4
5
```

The loop continues after skipping 3.

If `continue` were replaced by `break`, the output would be:

```
1
2
```

6. Incorrect indentation in nested loops

Consider the following program:

```
for row in range(1, 4):
    for column in range(1, 3):
        print(row, column)
```

The `print()` statement belongs to the inner loop. Therefore, it executes for every column in every row. If the indentation changes, the program's behavior may change as well. When using nested loops, carefully identify which loop contains each statement.

7. Exercises

Exercise 1

Determine the output of the following program without running it.

```
for number in range(2, 15, -4):
    print(number)
```

Exercise 2

Write a program that displays the integers from 1 to 20 using a for loop.

Exercise 3

Write a program that displays the integers from 10 down to 1, followed by the message `"Finished!"`.

Exercise 4

Write a program that calculates the sum of all even integers between 2 and 100,

Exercise 5

Compute $S = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{N}$ for a user-specified N.

Exercise 6

Write a loop that checks whether a given number n is prime.

Exercise 7

Write a program that displays the integers from 1 to 30 but skips all multiples of 4.

Exercise 9

A positive integer is called a perfect number if it equals the sum of its positive divisors excluding itself. For example: 6 = 1 + 2 + 3. Write a program that asks the user to enter a positive integer and determines whether it is a perfect number.

Exercise 10

Menu-driven program: Display a menu:

```
1. Add two numbers
2. Multiply two numbers
3. Exit
```

Use a while loop statement to handle user choices until they select Exit.

Exercise 11

Area calculator: Display a menu:

```
1. Circle
2. Rectangle
3. Triangle
```

Ask the user to choose an option and input the necessary dimensions, calculate and display the area.

Exercise 12

BMI classification: Ask for height (m) and weight (kg), compute *BMI = weight / height²*, and classify using:

- Underweight: < 18.5
- Normal: 18.5 – 24.9
- Overweight: 25 – 29.9
- Obese: ≥ 30

Exercise 13

Guess the number game: Generate a random integer between 1 and 100. Ask the user to guess the number and display: "Too high" or "Too low" until they find it. Count the number of attempts.

8. Chapter Summary

- Loops allow a program to repeat instructions without writing the same statements multiple times. Python provides two principal loop structures.
  - **`while` loop:** Repeats while a condition remains true.
  - **`for` loop:** Iterates over a sequence or a range of values.
- The `range()` function generates integer sequences and supports three common forms:

```
range(stop)
range(start, stop)
range(start, stop, step)
```

  In every form, the stop value is excluded.
- Counter-controlled loops use a variable to track the number of iterations. Accumulators store running totals or other cumulative results.
- Nested loops allow one loop to execute inside another, making them useful for grids, tables, and patterns.
- Two statements can control loop execution:
  - `break` terminates the nearest enclosing loop.
  - `continue` skips the remaining statements in the current iteration.
- When selecting a loop, consider the problem requirements. A for loop is often suitable when iterating through a known range or sequence, whereas a while

loop is useful when repetition depends on a condition that may change during execution.

- Finally, always check the loop condition, initialization, updates, and indentation to avoid common errors.

9. Review Questions

Answer the following questions to check your understanding.

1. What is a loop, and why is it useful in programming?
2. What is the difference between a while loop and a for loop?
3. What happens when the condition of a while loop is false before its first iteration?
4. What is an infinite loop? Give an example of how one can occur.
5. What is the purpose of the `range()` function?
6. Explain the difference between `range(5)` and `range(1, 5)`.
7. What values are generated by `range(3, 15, 4)`?
8. What is a nested loop, and when might it be useful?
9. Explain the difference between `break` and `continue`.
10. Which loop would you choose to request user input repeatedly until a valid value is entered? Explain your choice.
11. Which loop structure would you normally use to display a fixed number of measurements? Explain your choice.

10.Key Terms

Term	Meaning
Loop	A programming structure that repeats a block of code.
Iteration	One execution of the body of a loop.
`for` loop	A loop used to iterate over a sequence of values or items.
`while` loop	A loop that repeats as long as a specified condition is "True".
`range()` function	A built-in function that generates a sequence of integers, commonly used with "for" loops.
Loop variable	A variable that takes different values during loop execution.

Counter	A variable used to count the number of occurrences or repetitions.
Accumulator	A variable used to collect a running total or combine values across iterations.
Nested loop	A loop placed inside another loop.
Infinite loop	A loop that continues indefinitely because its termination condition is never reached.
`break` Statement	A statement that immediately terminates the nearest enclosing loop.
`continue` Statement	A statement that skips the remaining instructions in the current iteration and proceeds to the next iteration.
Loop control	The use of conditions and statements to determine how and when a loop continues or terminates.
Termination condition	A condition or event that causes a loop to stop.
Iteration count	The number of times a loop body executes.

11. References

1. **Python Software Foundation.** *Python 3 Documentation — The Python Tutorial and Python User Guide.*
2. **Severance, C. R.** *Python for Everybody: Exploring Data Using Python 3.*
3. **Downey, A. B.** *Think Python. How to Think Like a Computer Scientist. 2nd Edition,*
4. **Sweigart, A.** *Automate the Boring Stuff with Python.*