Python Essentials 1: Module 3

Boolean Values, Conditional Execution, Loops, Lists and List Processing, Logical and Bitwise Operations

In this module, you will cover the following topics:

the Boolean data type;
relational operators;
making decisions in Python (if, if-else, if-elif,else)
how to repeat code execution using loops (while, for)
how to perform logic and bitwise operations in Python;
lists in Python (constructing, indexing, and slicing; content manipulation)
how to sort a list using bubble-sort algorithms;
multidimensional lists and their applications.

(part 2)

Looping your code with while

Do you agree with the statement presented below?

```
while there is something to do
   do it
```

Note that this record also declares that if there is nothing to do, nothing at all will happen.

In general, in Python, a loop can be represented as follows:

```
while conditional_expression:
    instruction
```

If you notice some similarities to the if instruction, that's quite all right. Indeed, the syntactic difference is only one: you use the word while instead of the word if.

The semantic difference is more important: when the condition is met, if performs its statements only once; while repeats the execution as long as the condition evaluates to True.

Note: all the rules regarding indentation are applicable here, too. We'll show you this soon.

Look at the algorithm below:

```
while conditional_expression:
    instruction_one
    instruction_two
    instruction_three
    :
    :
    instruction_n
```

It is now important to remember that:

if you want to execute more than one statement inside one while, you must (as with if) indent all the instructions in the same way;
an instruction or set of instructions executed inside the while loop is called the loop's body;
if the condition is False (equal to zero) as early as when it is tested for the first time, the body is not executed even once (note the analogy of not having to do anything if there is nothing to do);
the body should be able to change the condition's value, because if the condition is True at the beginning, the body might run continuously to infinity - notice that doing a thing usually decreases the number of things to do).

An infinite loop
An infinite loop, also called an endless loop, is a sequence of instructions in a program which repeat indefinitely (loop endlessly.)

Here's an example of a loop that is not able to finish its execution:

```
while True:
    print("I'm stuck inside a loop.")
```

This loop will infinitely print "I'm stuck inside a loop." on the screen.

NOTE

If you want to get the best learning experience from seeing how an infinite loop behaves, launch IDLE, create a New File, copy-paste the above code, save your file, and run the program. What you will see is the never-ending sequence of "I'm stuck inside a loop." strings printed to the Python console window. To terminate your program, just press Ctrl-C (or Ctrl-Break on some computers). This will cause the so-called KeyboardInterrupt exception and let your program get out of the loop. We'll talk about it later in the course.

Let's go back to the sketch of the algorithm we showed you recently. We're going to show you how to use this newly learned loop to find the largest number from a large set of entered data.

Analyze the program carefully. See where the loop starts (line 8). Locate the loop's body and find out how the body is exited:

```python
# Store the current largest number here.
largest_number = -999999999

# Input the first value.
number = int(input("Enter a number or type -1 to stop: "))

# If the number is not equal to -1, continue.
while number != -1:
    # Is number larger than largest_number?
    if number > largest_number:
        # Yes, update largest_number.
        largest_number = number
    # Input the next number.
    number = int(input("Enter a number or type -1 to stop: "))

# Print the largest number.
print("The largest number is:", largest_number)
```

Check how this code implements the algorithm we showed you earlier.

The while loop: more examples

Let's look at another example employing the while loop. Follow the comments to find out the idea and the solution.

```
# A program that reads a sequence of numbers
# and counts how many numbers are even and how many are odd.
# The program terminates when zero is entered.

odd_numbers = 0
even_numbers = 0

# Read the first number.
number = int(input("Enter a number or type 0 to stop: "))

# 0 terminates execution.
while number != 0:
    # Check if the number is odd.
    if number % 2 == 1:
        # Increase the odd_numbers counter.
        odd_numbers += 1
    else:
        # Increase the even_numbers counter.
        even_numbers += 1
    # Read the next number.
    number = int(input("Enter a number or type 0 to stop: "))

# Print results.
print("Odd numbers count:", odd_numbers)
print("Even numbers count:", even_numbers)
```

Certain expressions can be simplified without changing the program's behavior.

Try to recall how Python interprets the truth of a condition, and note that these two forms are equivalent:

`while number != 0:` and `while number:`.

The condition that checks if a number is odd can be coded in these equivalent forms, too:

`if number % 2 == 1:` and `if number % 2:`.

Using a counter variable to exit a loop
Look at the snippet below:

```
counter = 5
while counter != 0:
    print("Inside the loop.", counter)
    counter -= 1
print("Outside the loop.", counter)
```

This code is intended to print the string "Inside the loop." and the value stored in the counter variable during a given loop exactly five times. Once the condition has not been met (the counter variable has reached 0), the loop is exited, and the message "Outside the loop." as well as the value stored in counter is printed.

But there's one thing that can be written more compactly - the condition of the while loop.

Can you see the difference?

```
counter = 5
while counter:
    print("Inside the loop.", counter)
    counter -= 1
print("Outside the loop.", counter)
```

Is it more compact than previously? A bit. Is it more legible? That's disputable.

REMEMBER

Don't feel obliged to code your programs in a way that is always the shortest and the most compact. Readability may be a more important factor. Keep your code ready for a new programmer.

LAB

Estimated time
15 minutes

Level of difficulty
Easy

Objectives
Familiarize the student with:

using the while loop;
reflecting real-life situations in computer code.
Scenario
A junior magician has picked a secret number. He has hidden it in a variable named secret_number. He wants everyone who run his program to play the Guess the secret number game, and guess what number he has picked for them. Those who don't guess the number will be stuck in an endless loop forever! Unfortunately, he does not know how to complete the code.

Your task is to help the magician complete the code in the editor in such a way so that the code:

will ask the user to enter an integer number;
will use a while loop;
will check whether the number entered by the user is the same as the number picked by the magician. If the number chosen by the user is different than the magician's secret number, the user should see the message "Ha ha! You're stuck in my loop!" and be prompted to enter a number again. If the number entered by the user matches the number picked by the magician, the number should be printed to the screen, and the magician should say the following words: "Well done, muggle! You are free now."
The magician is counting on you! Don't disappoint him.

code:

```
secret_number = 777

print(
"""
+================================+
| Welcome to my game, muggle!    |
| Enter an integer number        |
| and guess what number I've     |
| picked for you.                |
| So, what is the secret number? |
+================================+
""")
```

EXTRA INFO

By the way, look at the print() function. The way we've used it here is called multi-line printing. You can use triple quotes to print strings on multiple lines in order to make text easier to read, or create a special text-based design. Experiment with it.

Looping your code with for

Another kind of loop available in Python comes from the observation that sometimes it's more important to count the "turns" of the loop than to check the conditions.

Imagine that a loop's body needs to be executed exactly one hundred times. If you would like to use the while loop to do it, it may look like this:

```
i = 0
while i < 100:
    # do_something()
    i += 1
```

It would be nice if somebody could do this boring counting for you. Is that possible?

Of course it is - there's a special loop for these kinds of tasks, and it is named for.

Actually, the for loop is designed to do more complicated tasks - it can "browse" large collections of data item by item. We'll show you how to do that soon, but right now we're going to present a simpler variant of its application.

Take a look at the snippet:

```
for i in range(100):
    # do_something()
    pass
```

There are some new elements. Let us tell you about them:

the for keyword opens the for loop; note - there's no condition after it; you don't have to think about conditions, as they're checked internally, without any intervention;
any variable after the for keyword is the control variable of the loop; it counts the loop's turns, and does it automatically;
the in keyword introduces a syntax element describing the range of possible values being assigned to the control variable;
the range() function (this is a very special function) is responsible for generating all the desired values of the control variable; in our example, the function will create (we can even say that it will feed the loop with) subsequent values from the following set: 0, 1, 2 .. 97, 98, 99; note: in this case, the range() function starts its job from 0 and finishes it one step (one integer number) before the value of its argument;
note the pass keyword inside the loop body - it does nothing at all; it's an empty instruction - we put it here because the for loop's syntax demands at least one instruction inside the body (by the way - if, elif, else and while express the same thing)
Our next examples will be a bit more modest in the number of loop repetitions.

Take a look at the snippet below. Can you predict its output?

```
for i in range(10):
    print("The value of i is currently", i)
```

Run the code to check if you were right.

Note:

the loop has been executed ten times (it's the range() function's argument)
the last control variable's value is 9 (not 10, as it starts from 0, not from 1)
The range() function invocation may be equipped with two arguments, not just one:

```
for i in range(2, 8):
    print("The value of i is currently", i)
```

In this case, the first argument determines the initial (first) value of the control variable.

The last argument shows the first value the control variable will not be assigned.

Note: the range() function accepts only integers as its arguments, and generates sequences of integers.

Can you guess the output of the program? Run it to check if you were right now, too.

The first value shown is 2 (taken from the range()'s first argument.)

The last is 7 (although the range()'s second argument is 8).

More about the for loop and the range() function with three arguments

The range() function may also accept three arguments - take a look at the code in the editor.

The third argument is an increment - it's a value added to control the variable at every loop turn (as you may suspect, the default value of the increment is 1).

Can you tell us how many lines will appear in the console and what values they will contain?

Run the program to find out if you were right.

```
for i in range(2, 8, 3):
    print("The value of i is currently", i)
```

You should be able to see the following lines in the console window:

```
The value of i is currently 2
The value of i is currently 5
output
```

Do you know why? The first argument passed to the range() function tells us what the starting number of the sequence is (hence 2 in the output). The second argument tells the function where to stop the sequence (the function generates numbers up to the number indicated by the second argument, but does not include it). Finally, the third argument indicates the step, which actually means the difference between each number in the sequence of numbers generated by the function.

2 (starting number) → 5 (2 increment by 3 equals 5 - the number is within the range from 2 to 8) → 8 (5 increment by 3 equals 8 - the number is not within the range from 2 to 8, because the stop parameter is not included in the sequence of numbers generated by the function.)

Note: if the set generated by the range() function is empty, the loop won't execute its body at all.

Just like here - there will be no output:

```
for i in range(1, 1):
    print("The value of i is currently", i)
```

Note: the set generated by the range() has to be sorted in ascending order. There's no way to force the range() to create a set in a different form when the range() function accepts exactly two arguments. This means that the range()'s second argument must be greater than the first.

Thus, there will be no output here, either:

```
for i in range(2, 1):
    print("The value of i is currently", i)
```

Let's have a look at a short program whose task is to write some of the first powers of two:

```
power = 1
for expo in range(16):
    print("2 to the power of", expo, "is", power)
    power *= 2
```

The expo variable is used as a control variable for the loop, and indicates the current value of the exponent. The exponentiation itself is replaced by multiplying by two. Since 20 is equal to 1, then 2 × 1 is equal to 21, 2 × 21 is equal to 22, and so on. What is the greatest exponent for which our program still prints the result?

Run the code and check if the output matches your expectations.

LAB

Estimated time
5-15 minutes

Level of difficulty
Very easy

Objectives
Familiarize the student with:

using the for loop;
reflecting real-life situations in computer code.
Scenario
Do you know what Mississippi is? Well, it's the name of one of the states and rivers in the United States. The Mississippi River is about 2,340 miles long, which makes it the second longest river in the United States (the longest being the Missouri River). It's so long that a single drop of water needs 90 days to travel its entire length!

The word Mississippi is also used for a slightly different purpose: to count mississippily.

If you're not familiar with the phrase, we're here to explain to you what it means: it's used to count seconds.

The idea behind it is that adding the word Mississippi to a number when counting seconds aloud makes them sound closer to clock-time, and therefore "one Mississippi, two Mississippi, three Mississippi" will take approximately an actual three seconds of time! It's often used by children playing hide-and-seek to make sure the seeker does an honest count.

Your task is very simple here: write a program that uses a for loop to "count mississippily" to five. Having counted to five, the program should print to the screen the final message "Ready or not, here I come!"

Use the skeleton we've provided in the editor.

EXTRA INFO

Note that the code in the editor contains two elements which may not be fully clear to you at this moment: the import time statement, and the sleep() method. We're going to talk about them soon.

For the time being, we'd just like you to know that we've imported the time module and used the sleep() method to suspend the execution of each subsequent print() function inside the for loop for one second, so that the message outputted to the console resembles an actual counting. Don't worry - you'll soon learn more about modules and methods.

Expected output
```
1 Mississippi
2 Mississippi
3 Mississippi
4 Mississippi
5 Mississippi
```

The break and continue statements

So far, we've treated the body of the loop as an indivisible and inseparable sequence of instructions that are performed completely at every turn of the loop. However, as developer, you could be faced with the following choices:

it appears that it's unnecessary to continue the loop as a whole; you should refrain from further execution of the loop's body and go further;
it appears that you need to start the next turn of the loop without completing the execution of the current turn.
Python provides two special instructions for the implementation of both these tasks. Let's say for the sake of accuracy that their existence in the language is not necessary - an experienced programmer is able to code any algorithm without these instructions. Such additions, which don't improve the language's expressive power, but only simplify the developer's work, are sometimes called syntactic candy, or syntactic sugar.

These two instructions are:

break - exits the loop immediately, and unconditionally ends the loop's operation; the program begins to execute the nearest instruction after the loop's body;
continue - behaves as if the program has suddenly reached the end of the body; the next turn is started and the condition expression is tested immediately.
Both these words are keywords.

Now we'll show you two simple examples to illustrate how the two instructions work. Look at the code in the editor. Run the program and analyze the output. Modify the code and experiment.

```
# break - example

print("The break instruction:")
for i in range(1, 6):
    if i == 3:
        break
    print("Inside the loop.", i)
print("Outside the loop.")


# continue - example

print("\nThe continue instruction:")
for i in range(1, 6):
    if i == 3:
        continue
    print("Inside the loop.", i)
print("Outside the loop.")
```

The break and continue statements: more examples

Let's return to our program that recognizes the largest among the entered numbers. We'll convert it twice, using the break and continue instructions.

Analyze the code, and judge whether and how you would use either of them.

The break variant goes here:

```
largest_number = -99999999
counter = 0

while True:
    number = int(input("Enter a number or type -1 to end program: "))
    if number == -1:
        break
    counter += 1
    if number > largest_number:
        largest_number = number

if counter != 0:
    print("The largest number is", largest_number)
else:
    print("You haven't entered any number.")
```

Run it, test it, and experiment with it.

And now the continue variant:

```
largest_number = -99999999
counter = 0

number = int(input("Enter a number or type -1 to end program: "))

while number != -1:
    if number == -1:
        continue
    counter += 1

    if number > largest_number:
        largest_number = number
    number = int(input("Enter a number or type -1 to end program: "))

if counter:
    print("The largest number is", largest_number)
else:
    print("You haven't entered any number.")
```

Look carefully, the user enters the first number before the program enters the while loop. The subsequent number is entered when the program is already in the loop.

Again - run the program, test it, and experiment with it.

LAB

Estimated time
10-20 minutes

Level of difficulty
Easy

Objectives
Familiarize the student with:

using the break statement in loops;
reflecting real-life situations in computer code.
Scenario
The break statement is used to exit/terminate a loop.

Design a program that uses a while loop and continuously asks the user to enter a word unless the user enters "chupacabra" as the secret exit word, in which case the message "You've successfully left the loop." should be printed to the screen, and the loop should terminate.

Don't print any of the words entered by the user. Use the concept of conditional execution and the break statement.

LAB

Estimated time
10-20 minutes

Level of difficulty
Easy

Objectives
Familiarize the student with:

using the continue statement in loops;
reflecting real-life situations in computer code.
Scenario
The continue statement is used to skip the current block and move ahead to the next iteration, without executing the statements inside the loop.

It can be used with both the while and for loops.

Your task here is very special: you must design a vowel eater! Write a program that uses:

a for loop;
the concept of conditional execution (if-elif-else)
the continue statement.
Your program must:

ask the user to enter a word;
use user_word = user_word.upper() to convert the word entered by the user to upper case; we'll talk about the so-called string methods and the upper() method very soon - don't worry;
use conditional execution and the continue statement to "eat" the following vowels A, E, I, O, U from the inputted word;
print the uneaten letters to the screen, each one of them on a separate line.
Test your program with the data we've provided for you.

Test data
Sample input: Gregory
Expected output:
G
R
G
R
Y
Sample input: abstemious

Expected output:
B
S
T
M
S
Sample input: IOUEA

Expected output:

LAB

Estimated time
5-15 minutes

Level of difficulty
Easy

Objectives
Familiarize the student with:

using the continue statement in loops;
modifying and upgrading the existing code;
reflecting real-life situations in computer code.
Scenario
Your task here is even more special than before: you must redesign the (ugly) vowel eater from the previous lab (3.1.2.10) and create a better, upgraded (pretty) vowel eater! Write a program that uses:

a for loop;
the concept of conditional execution (if-elif-else)
the continue statement.
Your program must:

ask the user to enter a word;
use user_word = user_word.upper() to convert the word entered by the user to upper case; we'll talk about the so-called string methods and the upper() method very soon - don't worry;
use conditional execution and the continue statement to "eat" the following vowels A, E, I, O, U from the inputted word;
assign the uneaten letters to the word_without_vowels variable and print the variable to the screen.
Look at the code in the editor. We've created word_without_vowels and assigned an empty string to it. Use concatenation operation to ask Python to combine selected letters into a longer string during subsequent loop turns, and assign it to the word_without_vowels variable.

Test your program with the data we've provided for you.

Test data
Sample input: Gregory

Expected output:

GRGRY
Sample input: abstemious

Expected output:

BSTMS
Sample input: IOUEA

Expected output:

The while loop and the else branch

Both loops, while and for, have one interesting (and rarely used) feature.

We'll show you how it works - try to judge for yourself if it's usable and whether you can live without it or not.

In other words, try to convince yourself if the feature is valuable and useful, or is just syntactic sugar.

Take a look at the snippet in the editor. There's something strange at the end - the else keyword.

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

As you may have suspected, loops may have the else branch too, like ifs.

The loop's else branch is always executed once, regardless of whether the loop has entered its body or not.

Can you guess the output? Run the program to check if you were right.

Modify the snippet a bit so that the loop has no chance to execute its body even once:

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

The while's condition is False at the beginning - can you see it?

Run and test the program, and check whether the else branch has been executed or not.

The for loop and the else branch

for loops behave a bit differently - take a look at the snippet in the editor and run it.

The output may be a bit surprising.

The i variable retains its last value.

Modify the code a bit to carry out one more experiment.

```
i = 111
for i in range(2, 1):
    print(i)
else:
    print("else:", i)
```

Can you guess the output?

The loop's body won't be executed here at all. Note: we've assigned the i variable before the loop.

Run the program and check its output.

```
for i in range(5):
    print(i)
else:
    print("else:", i)
```

When the loop's body isn't executed, the control variable retains the value it had before the loop.

Note: if the control variable doesn't exist before the loop starts, it won't exist when the execution reaches the else branch.

How do you feel about this variant of else?

Now we're going to tell you about some other kinds of variables. Our current variables can only store one value at a time, but there are variables that can do much more - they can store as many values as you want.

LAB

Estimated time
20-30 minutes

Level of difficulty
Medium

Objectives
Familiarize the student with:

using the while loop;
finding the proper implementation of verbally defined rules;
reflecting real-life situations in computer code.
Scenario
Listen to this story: a boy and his father, a computer programmer, are playing with wooden blocks. They are building a pyramid.

Their pyramid is a bit weird, as it is actually a pyramid-shaped wall - it's flat. The pyramid is stacked according to one simple principle: each lower layer contains one block more than the layer above.

The figure illustrates the rule used by the builders:

Your task is to write a program which reads the number of blocks the builders have, and outputs the height of the pyramid that can be built using these blocks.

Note: the height is measured by the number of fully completed layers - if the builders don't have a sufficient number of blocks and cannot complete the next layer, they finish their work immediately.

Test your code using the data we've provided.

Test Data

Sample input: 6

Expected output: The height of the pyramid: 3

Sample input: 20

Expected output: The height of the pyramid: 5

Sample input: 1000

Expected output: The height of the pyramid: 44

Sample input: 2

Expected output: The height of the pyramid: 1

LAB

Estimated time
20 minutes

Level of difficulty
Medium

Objectives
Familiarize the student with:

using the while loop;
converting verbally defined loops into actual Python code.
Scenario
In 1937, a German mathematician named Lothar Collatz formulated an intriguing hypothesis (it still remains unproven) which can be described in the following way:

take any non-negative and non-zero integer number and name it c0;
if it's even, evaluate a new c0 as c0 ÷ 2;
otherwise, if it's odd, evaluate a new c0 as 3 × c0 + 1;
if c0 ≠ 1, skip to point 2.
The hypothesis says that regardless of the initial value of c0, it will always go to 1.

Of course, it's an extremely complex task to use a computer in order to prove the hypothesis for any natural number (it may even require artificial intelligence), but you can use Python to check some individual numbers. Maybe you'll even find the one which would disprove the hypothesis.

Write a program which reads one natural number and executes the above steps as long as c0 remains different from 1. We also want you to count the steps needed to achieve the goal. Your code should output all the intermediate values of c0, too.

Hint: the most important part of the problem is how to transform Collatz's idea into a while loop - this is the key to success.

Test your code using the data we've provided.

Test Data

Sample input: 15

Expected output:

```
46
23
70
35
106
53
160
80
40
20
```

```
10
5
16
8
4
2
1
steps = 17
```

Sample input: 16

Expected output:

```
8
4
2
1
steps = 4
```

Sample input: 1023

Expected output:

```
3070
1535
4606
2303
6910
3455
10366
5183
15550
7775
23326
11663
34990
17495
52486
26243
78730
39365
118096
59048
29524
14762
7381
22144
11072
5536
2768
1384
692
346
173
520
260
130
```

```
65
196
98
49
148
74
37
112
56
28
14
7
22
11
34
17
52
26
13
40
20
10
5
16
8
4
2
1
steps = 62
```

Key takeaways

1. There are two types of loops in Python: while and for:

the while loop executes a statement or a set of statements as long as a specified boolean condition is true, e.g.:

```
# Example 1
while True:
    print("Stuck in an infinite loop.")

# Example 2
counter = 5
while counter > 2:
    print(counter)
    counter -= 1
```

the for loop executes a set of statements many times; it's used to iterate over a sequence (e.g., a list, a dictionary, a tuple, or a set - you will learn about them soon) or other objects that are iterable (e.g., strings). You can use the for loop to iterate over a sequence of numbers using the built-in range function. Look at the examples below:

```
# Example 1
word = "Python"
for letter in word:
    print(letter, end="*")

# Example 2
for i in range(1, 10):
    if i % 2 == 0:
        print(i)
```

2. You can use the break and continue statements to change the flow of a loop:

You use break to exit a loop, e.g.:

```
text = "OpenEDG Python Institute"
for letter in text:
    if letter == "P":
        break
    print(letter, end="")
```

You use continue to skip the current iteration, and continue with the next iteration, e.g.:

```
text = "pyxpyxpyx"
for letter in text:
    if letter == "x":
        continue
    print(letter, end="")
```

3. The while and for loops can also have an else clause in Python. The else clause executes after the loop finishes its execution as long as it has not been terminated by break, e.g.:

```
n = 0

while n != 3:
    print(n)
    n += 1
else:
    print(n, "else")

print()

for i in range(0, 3):
    print(i)
else:
    print(i, "else")
```

4. The range() function generates a sequence of numbers. It accepts integers and returns range objects. The syntax of range() looks as follows: range(start, stop, step), where:

start is an optional parameter specifying the starting number of the sequence (0 by default)
stop is an optional parameter specifying the end of the sequence generated (it is not included),
and step is an optional parameter specifying the difference between the numbers in the sequence (1 by default.)
Example code:

```
for i in range(3):
    print(i, end=" ")  # Outputs: 0 1 2

for i in range(6, 1, -2):
    print(i, end=" ")  # Outputs: 6, 4, 2
```

Key takeaways: continued

Exercise 1

Create a for loop that counts from 0 to 10, and prints odd numbers to the screen. Use the skeleton below:

```
for i in range(1, 11):
    # Line of code.
        # Line of code.
```

Check

Exercise 2

Create a while loop that counts from 0 to 10, and prints odd numbers to the screen. Use the skeleton below:

```
x = 1
while x < 11:
    # Line of code.
        # Line of code.
    # Line of code.
```

Check

Exercise 3

Create a program with a for loop and a break statement. The program should iterate over characters in an email address, exit the loop when it reaches the @ symbol, and print the part before @ on one line. Use the skeleton below:

```
for ch in "john.smith@pythoninstitute.org":
    if ch == "@":
        # Line of code.
    # Line of code.
```

Check

Exercise 4

Create a program with a for loop and a continue statement. The program should iterate over a string of digits, replace each 0 with x, and print the modified string to the screen. Use the skeleton below:

```
for digit in "0165031806510":
    if digit == "0":
        # Line of code.
        # Line of code.
    # Line of code.
```

Check

Exercise 5

What is the output of the following code?

```
n = 3

while n > 0:
    print(n + 1)
    n -= 1
else:
    print(n)
```

Check
Exercise 6

What is the output of the following code?

```
n = range(4)

for num in n:
    print(num - 1)
else:
    print(num)
```

Check
Exercise 7

What is the output of the following code?

```
for i in range(0, 6, 3):
    print(i)
```