Chapter 3: Control flow

In the context of computer programming, control flow refers to the sequence in which each line of code is executed. The script in the previous chapter had a straightforward sequence: each line of code was executed once and one after the other until the final line was reached. Most of the time, this sequence will not be as straightforward.

Consider the design of a user login system, where you must verify a user's password against a stored one. If they match, you display a message that the user has been granted access to the system. If they do not match you display a message saying the login credentials are incorrect. These two messages require distinct sets of code to generate them. Which one is executed depends on the outcome of the password comparison. In this situation, we will use a control flow mechanism called an *if-statement*. This statement evaluates a logical expression and allows you to specify distinct sets of code to be executed, depending on the outcome of the evaluation.

In other situations, some lines of code will need to be interpreted not once but several times. This procedure is commonly known as looping, and Python has two mechanisms to implement this. A first mechanism is used when the decision to repeat a block of code can be defined by a logical expression. For instance, we might design a demographics program that requires participants to enter their age as a number (i.e., 16 and not sixteen). In that case, we could get the user input, check if it is an actual number, and repeat these steps if the participant's input is invalid. Stated differently, if the participant's input is invalid, we want to repeat a block of code. This mechanism is called a *while-loop.*

In the example above it is not possible to know precisely in advance how many times a block of code will need to be repeated (because that depends on the input that the user provides). However, there are scenarios where you do have prior knowledge of the exact number of repetitions needed. For example, if you want to step out the edges of a square, you could write a procedure to move forward a fixed distance, followed by a 90° turn. By repeating this procedure precisely four times you have stepped out a square. Another example can be found in psychological experiments, where there is a predefined list of parameters specifying how a stimulus hould appear in each trial. For each item in this list, trial-specific code is executed, involving actions like displaying the stimulus with the given parameters and collecting responses. Python addresses these situations with for-loops.

The if-statement

To explain the different control flow mechanisms, we will use a combination of *pseudocode* and *flow charts*. Pseudocode resembles actual Python code with some parts that still need to be replaced with actual code. A flow chart is a visual representation that represents your code as a collection of rectangular blocks. Diamond-shaped blocks stand for parts in your code where decisions are made

about what code will be executed next, and they can be used to visualize how execution in your program will progress.

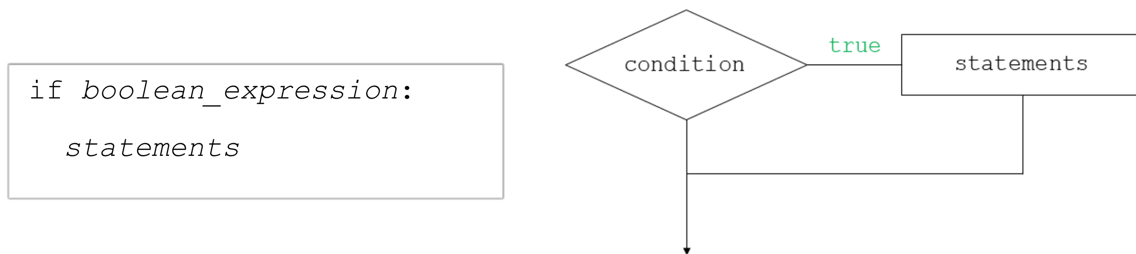

Figure 1 Pseudocode (left) and control flow diagram (right) for the if-statement

The if-statement, depicted in Figure 1, consists of two parts: the if statement's *header* and its body. In the header, we begin with the "if" keyword, followed by a boolean expression, and conclude with a colon. Following the header, the next line of code is indented by a certain number of spaces to signify its inclusion in the body of the if-statement. The body will only be executed if the boolean expression in the header evaluates to True. It can have multiple statements, under the condition that each statement is also indented by the same number of spaces. Most code editors will recognize that you are writing an if-statement and will take care of the indentation for you.

Listing 1 gives a demonstration of an if-statement in a program that evaluates a reaction time variable. The header of the if-statement has a boolean expression comparing the variable to a constant value. The body has a print statement that will be executed if the value for the reaction time is smaller than 200. Before running this code on your computer, try to predict the output. What do you think will be printed here?

```
1.  # Define a variable
2.  reaction_time = 200
3.
4.  print("We will now compare the value of reaction_time to 200")
5.
6.  # Evaluate the if-statement
7.  if reaction_time < 200:
8.      print("Well done")
9.
10. # First line of code after the if-statement
11. print("This will always be executed")
```

Listing 1: demonstration of an if-statement.

Note how the program has `print` statements both before, within, and after the if-statement. The first and last statements should always be executed, and the middle depends on the value of the reaction time variable. Even when output is not needed in your program, it can sometimes still be useful to add similar print statements that map to the control-flow of your program. Sometimes you might have an error in your code because you assume that some code in some if-statement is

executed, but by placing a print statement within that if-statement you discover that nothing is printed at all and therefore the if-statement is, in fact, not executed. In that sense, these print statements can aid in debugging your code and guide you toward potential mistakes you are making.

The version of the if-statement that we just showed only has a body with code to be executed when the value of the Boolean expression is True. More often, we want to execute an alternative set of instructions in case the condition is not true as well. For this we can add an `else` clause to the if statement. The indentation of the else syntax must align with the if-statement header (together they form an if-else statement). The body of the `else` clause is again indicated using an indentation to the right.

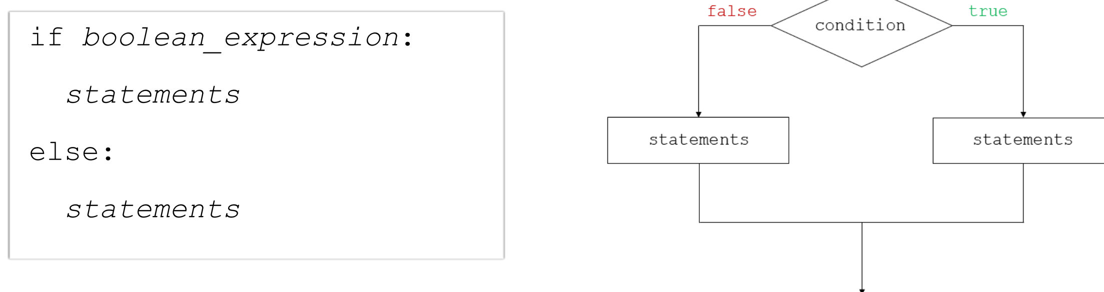

Figure 2: Pseudocode and flow chart of an if…else statement

In Listing 2, we have applied this structure by extending the code example from Listing 1. If the reaction time is smaller than 200 ms, we still get the same message. If that is not the case, we get a message that the participant is responding too fast. Again, try to change the value of the reaction time to see how it affects the output of the program.

```
1.  # Define a variable
2.  reaction_time = 200
3.
4.  print("We will now compare the value of rt to 200")
5.
6.  # Evalute the if-statement
7.  if reaction_time < 200:
8.      print("Well done")
9.  else:
10.     print("Too fast")
11.
12. # First line of code after the if-statement
13. print("This will always be executed")
```

Listing 2 demonstration of an if…else statement

We can extend this structure even further by adding an `elif` statement after the body of the first if-statement. Now we can use another boolean expression and a corresponding body of code that will run when this expression evaluates to true (Figure 3).

```
if boolean_expression:
    statements
elif boolean_expression:
    statements
else:
    statements
```

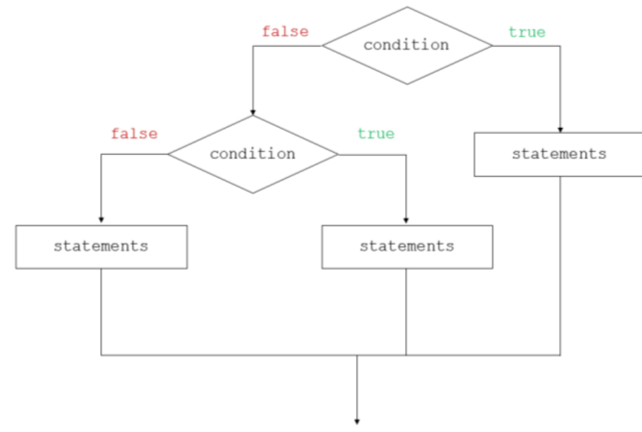

Figure 3 Pseudocode and flow chart of an if…elif…else statement

It is important to recognize that the mechanism that evaluates the expressions in an if-elif-else statement will evaluate the body of code that corresponds to the first statement that evaluates to True, and it will skip any remaining statements. This means that there can be a situation where both the `if` and the `elif` will have a true expression. But since the if-statement comes first, only that corresponding code will be executed.

A demonstration of this structure is shown in Listing 3. We first check if the reaction time is smaller than 200 ms. If that is not the case, we check if the reaction time is larger than 200 ms. Finally, the else clause captures the cases where the reaction time is exactly equal to 200 ms.

```
1.  # Define a variable
2.  rt = 200
3.
4.  print("We will now compare the value of rt to 200")
5.
6.  # Evaluate the if-statement
7.  if rt < 200:
8.      print("Too slow")
9.  elif rt > 2000:
10.     print("Too fast")
11. else:
12.     print("Well done")
13.
14. # First line of code after the if-statement
15. print("This will always be executed")
```

Listing 3: Demonstration of an if … elif … else statement

Nested if statements

There are no constraints on the code that you enter in the body of an if-statement. This means that you can have another if-statement within an if-statement, a structure that is called a “nested if-statement". Listing 4 demonstrates this idea in a program that gives feedback on the reaction time, but only if comments have been enabled through a `display_comments` variable.

```
1.  # Use this variable to set if we display comments
```

```
2.  display_comments = True
3.
4.  # Define a reaction time
5.  rt = 300
6.
7.  if rt < 300:
8.     if display_comments == True:
9.        print("Too slow!")
10.
11.    print("I might have displayed a comment")
12.
13.
14. print("Back in the main code")
```

Listing 4: Example of a nested if-statement

Although nested statements are occasionally useful, you should be careful that you do not exaggerate with the level of your nested statements (meaning how many if within if within if … statements you have). Having too many nested statements makes code more complex to read and more difficult to debug and should therefore be avoided.

Equivalence of code

It will often be the case that code can be written in different ways, while still producing the same output. To illustrate this, we have rewritten the example from listing 3 below. Instead of using an `if-elif-else` structure, we have now used three `if` statements.

```
1.  # Define a variable
2.  rt = 200
3.
4.  print("We will now compare the value of rt to 200")
5.
6.  # Multiple if-statements instead of if...elif...else
7.  if rt < 200:
8.     print("Too slow")
9.  if rt > 2000:
10.    print("Too fast")
11. if rt >= 200 and rt <= 2000:
12.    print("Well done")
13.
14. print("This will always be executed")
```

Listing 5: This code is equivalent to the code presented in listing 3.

While the output is the same in both listings, there is a difference in the way this code is executed. Because we have three separate if-statements, each Boolean expression will always be evaluated independently of the truth value of the other expressions. In contrast, in an `if…elif…else` statement, each Boolean expression will be evaluated *until* the first one that is true is met. You could also argue that they are psychologically different (not that this matters for the computer) in the sense that an `if-elif-else` statement forms a code entity that relates to a specific operation, namely evaluating the reaction time. Therefore, using separate if-statements makes less sense here.

As your code grows in complexity, it's easy to imagine that the range of possible ways to write it also expands accordingly. This can pose a potential distraction for novice programmers who might begin to question whether their approach is the "correct" one. When you are taking your first steps into programming, your primary concern should be if the code does its intended task. If that is the case, it's a success. Improving code quality and structure can always come later. This is a common practice called code refactoring. The core idea is that you modify an existing ode to enhance its organization and readability while maintaining its original functionality.

The while-statement

If we need to repeat a block of code, and if the decision to repeat the code depends on a boolean expression, we can use a while statement. If the boolean expression evaluates to true, the body of the while loop is executed. As with statements, the body of a while statement is indicated using indentation. The typical procedure is that we start with an initialization step that makes sure that the boolean expression evaluates to true at least the first time it is evaluated. The code in the while statement then takes care of updating the variables that are used in this boolean expression so that at some point the expression can also evaluate to false and the while loop exits.

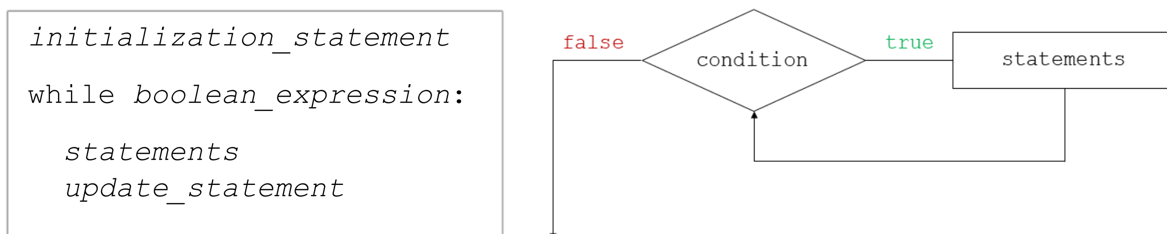

Figure 4: Pseudocode and flow chart of a while statement

```
1.  # Specify initial condition
2.  correct_password = False
3.
4.  # Evaluate the condition
5.  while correct_password == False:
6.      user_password = input("Please enter the password: ")
7.
8.      # Evaluate if the condition can be updated
9.      if user_password == "SECRET":
10.         correct_password = True
11.
12. # Here we are outside the while loop
13. print("While loop has ended")
```

Listing 6: A password verifier

In listing 6 we have a program that repeatedly asks a user to provide a password until we have the correct one. We start with an initialization step and set the value of the `correct_password` variable to False. That way, the Boolean expression in the while statement (which comes immediately

after the initialization), will be True at least once (i.e., it is True that the value of `correct_password` is False).

In the body, we first retrieve the user input. This input is then compared to the correct password. If there is a match, we update the value of the `correct_password` variable. When we reach the end of the body of the while-statement, code execution moves back to the header in which the boolean expression is evaluated again. If we had a match, this expression is no longer true and code execution moves to the first line after the body of the while-loop. If not, the code is repeated.

It is possible to forget to write a proper update step, for example when we would not have included the if-statement in the example above. In that case, the while-loop will repeat indefinitely, a situation called an infinite while loop. If this happens, you must end your program manually. One way is to use a Task Manager (on Windows). The Spyder IDE also has a red button in the IPython console window. Pressing this button interrupts the execution of your script so that you can fix this mistake and run it again.

Altering the flow of the while loop

There are two statements that can be used to alter the normal flow of events in a while loop. The *continue* statement can be used as an instruction to skip any remaining code in the while-loop and jump back to the header where the boolean expression is evaluated again.

```
1.  # Initialization
2.  counter = 0
3.
4.  # while-loop
5.  while counter <= 5:
6.      counter = counter + 1
7.
8.      if counter == 3:
9.          continue
10.
11.     print(counter)
12.
13. print("Done")
```

Listing 7: demonstration of the continue statement

In Listing 7 we have a program that increments a numerical variable until it exceeds the number 5. All the numbers are printed, except for the number 3. This is because when the counter variable reaches that value, it leads to the execution of the continue statement. This forces execution back to the start of the while-loop, therefore skipping the print statement.

A *break* statement can be used to break out of the while loop completely. As with the continue statement, it will skip any remaining statements in the while-body. But it will also skip the head of the while-statement and move code execution to the first line of code after the while-loop.

```
1. # Initialization
2. counter = 0
3.
4. # while-loop
5. while counter <= 5:
6.     counter = counter + 1
7.
8.     if counter == 3:
9.         break
10.
11.    print(counter)
12.
13. print("Done")
```

Listing 8: Demonstration of using a break statement to end a while loop irrespective of its condition

This is demonstrated in listing 8, where we have changed the continue statement into a break statement. Now, when the counter variable becomes equal to 3, the break statement will move code execution to line 13 (the first line outside the while loop), so only the numbers 1 and 2 will be printed.