\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CSC121 Python Programming**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Lesson 04 Iterative Control Structures

# **Objectives**

In this learning unit, students will learn:

- How to write program to repeat the same action multiple times

- How to create a loop

- How to write indefinite loops

- How to write definite loops

# **4.1 Overview**

So far we have learned using assignment statements to perform arithmetic calculations and using if statements to perform selections. With what we have learned we are able to write some useful programs. However, every program we have written so far can handle only one case, e.g. determining whether one student is eligible to graduate, or calculating tuition for one student, etc. We can make our programs more useful if one program can process multiple cases, e.g. calculating tuition for multiple students. This lesson will show us how to do that.

# **4.2 Creating a Loop**

Imagine a program that calculates commissions a realtor earns for selling houses. The program asks for the price of the first house sold and uses it to calculate the commission (with a 3% commission rate). Then the program repeats this action again and again for other houses until the commission for every house sold has been calculated. The following shows how it looks like when you run the program:

Enter house price: 200000

Commission: 6000

Calculate commission for another house?[y/n] y

Enter house price: 250000

Commission: 7500

Calculate commission for another house?[y/n] y

Enter house price: 180000

Commission: 5400

Calculate commission for another house?[y/n] y

Enter house price: 1000000

Commission: 30000

Calculate commission for another house?[y/n] n

To make the computer do this, we need to use an **iterative control**, which is also called a **loop**. In Python, we can use a while statement to implement a loop. A while statement is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression. The syntax of while statement is very similar to the syntax of if statement, except that the keyword if is replaced by the keyword while:

**while** *condition***:**

*block of statements to execute if the condition is true*

Similar to an if statement, a while statement has a condition and a statement block. The condition is a Boolean expression. If the condition is true, the computer executes the loop’s statement block (also called **loop body**) once, and then it will loop back to the condition and test it again to see whether the condition is still true. If the condition is still true, the computer executes the loop’s statement block for another time, and then it tests the condition again. This repeats again and again until the loop’s condition is false. When the condition of the loop becomes false, the computer will no longer execute the loop’s statement block any more.

The following diagram shows the idea of a loop:

true

Condition true/false?

Statements after loop

false

Statements before loop

Statements to execute if the condition is true

In the next two sections, you will see how to use the while statement to create two types of loops: indefinite loops and definite loops.

# **4.3 Indefinite Loops**

An **indefinite loop** is a loop in which the number of times that the loop will iterate cannot be determined before the loop is executed. The realtor commission program, which will be written soon, is an example of indefinite loops. In that program, we are going to write statements in the loop body to calculate commission for one house. The number of times the loop will iterate is controlled by the user when the loop is executing. The loop executes again and again as long as the user says he wants to calculate commission for another house. The loop stops repeating only when the user wants to stop.

Let’s write the realtor commission program. We will start with the algorithm:

1. *Input price of the first house*
2. *Calculate commission = house price \* 3%*
3. *Display commission*
4. *Input whether to calculate commission for another house*
5. *While answer is yes:*

*Input price of next house*

*Calculate commission = house price \* 3%*

*Display commission*

*Input whether to calculate commission for another house*

The following is the Python code:

house\_price = float(input(**"Enter house price: "**))  
commission = house\_price \* 0.03  
print(**"Commission:"**, commission)  
again= input(**"Calculate commission for another house?[y/n] "**)  
**while** again == **'y'**:  
 house\_price = float(input(**"Enter house price: "**))  
 commission = house\_price \* 0.03  
 print(**"Commission:"**, commission)  
 again = input(**"Calculate commission for another house?[y/n] "**)

In the program, we have statements to get price, calculate and display commission for the first house. We then ask the user to enter y or n to indicate whether he wants to calculate commission for another house. We use a while statement to implement the loop. The condition is again == 'y'. That means if the user enters y the loop will iterate again, i.e., the program will calculate commission for another house. The computer will stop executing the loop when the condition is false, i.e. when the user does not answer y.

Let’s look at another example of indefinite loops. We wrote a program to calculate number of BTU needed to cool a room. In this example, we modify the program so that we can calculate BTU needed for each home in a house. The following is the algorithm:

1. *Enter room length, width and height*
2. *Calculate room volume = room length \* room width \* room height*
3. *Calculate BTU needed = room volume \* 3.5*
4. *Display BTU needed*
5. *Enter whether to calculate BTU needed for another room*
6. *While answer is yes:*

*Enter room length, width and height*

*Calculate room volume = room length \* room width \* room height*

*Calculate BTU needed = room volume \* 3.5*

*Display BTU needed*

*Enter whether to calculate BTU needed for another room*

The following is the Python code.

length = float(input(**"Enter room length: "**))  
width = float(input(**"Enter room width: "**))  
height = float(input(**"Enter room height: "**))  
volume = length \* width \* height  
btu = volume \* 3.5  
print(**"BTU needed:"**, btu)  
again = input(**"Calculate BTU needed for another room?[y/n] "**)  
**while** again == **'y'**:  
 length = float(input(**"Enter room length: "**))  
 width = float(input(**"Enter room width: "**))  
 height = float(input(**"Enter room height: "**))  
 volume = length \* width \* height  
 btu = volume \* 3.5  
 print(**"BTU needed:"**, btu)  
 again = input(**"Calculate BTU needed for another room?[y/n] "**)

Similar to the commission program, we use a loop to calculate BTU needed for other rooms after we have finished the first room. We ask the user whether he wants to calculate BTU needed for another room every time calculation for one room is finished. As long as the user enters y, the program will continue to do the same task until the user finally enters something else. The following is a sample test run:

Enter room length: 15

Enter room width: 14

Enter room height: 11

BTU needed: 8085.0

Calculate BTU needed for another room?[y/n] y

Enter room length: 21

Enter room width: 17

Enter room height: 10

BTU needed: 12495.0

Calculate BTU needed for another room?[y/n] y

Enter room length: 9

Enter room width: 7

Enter room height: 10.5

BTU needed: 2315.25

Calculate BTU needed for another room?[y/n] n

If you read the Python code of the BTU program again, you will find that the statements before the loop are exactly the same as the statements inside the loop. The reason is simple. The code we need to process the first case, which is the code before the loop, is exactly the same as the code we need to process other cases, which is the code inside the loop body. Is there a way to shorten the program by eliminating one set of statements?

The answer is yes. We can eliminate the statements before the loop and let the loop body to handle the first and later cases. We just need to find a way to ensure that the program will always execute the loop body at least once. How do we ensure that? Since the loop condition is again == **'y'**, the program will execute the loop body at least once if we set again to y right before the loop. The following is the modified Python code:

again = **'y'  
while** again == **'y'**:  
 length = float(input(**"Enter room length: "**))  
 width = float(input(**"Enter room width: "**))  
 height = float(input(**"Enter room height: "**))  
 volume = length \* width \* height  
 btu = volume \* 3.5  
 print(**"BTU needed:"**, btu)  
 again = input(**"Calculate BTU needed for another room?[y/n] "**)

Similarly, the realtor commission program can be shortened like this:

again = **'y'  
while** again == **'y'**:  
 house\_price = float(input(**"Enter house price: "**))  
 commission = house\_price \* 0.03  
 print(**"Commission:"**, commission)  
 again = input(**"Calculate commission for another house?[y/n] "**)

We have seen two examples of indefinite loops. In both examples, we ask the user whether to process another case after one case is finished. The user controls whether the loop will continue to iterate or not. Indefinite loops can be used in other situations. One example is input error checking, which is also called input validation. The program asks the user to input something. If the user enters something that has error, the program uses a loop to ask the user to re-enter the input until the input has no error in it. Let’s look at one example.

In an earlier lesson we saw a program that calculated user’s age next year:

age = float(input(**'How old are you now? '**))  
age\_next\_yr = 1 + age  
print(**'You are'**, age\_next\_yr, **'years old next year.'**)

The program performs no check on the age entered by the user. We know that age should not be a negative number. Therefore, if a negative age is entered, we know it is an input error. We can add a loop to check age:

age = float(input(**'How old are you now? '**))  
**while** age < 0:  
 print(**'Error: Age cannot be negative.'**)  
 age = float(input(**'How old are you now? '**))  
age\_next\_yr = 1 + age  
print(**'You are'**, age\_next\_yr, **'years old next year.'**)

An input error checking loop is inserted in the program right after age is entered. It checks whether age is less than 0. If it is, an error message is displayed and the user is asked to enter the age again. The following is a sample test run of the program:

How old are you now? -15

Error: Age cannot be negative.

How old are you now? 15

You are 16.0 years old next year.

Some students may ask why we need a loop there. Can’t we just use an if statement to check age?

age = float(input(**'How old are you now? '**))  
**if** age < 0: # is this good enough?  
 print(**'Error: Age cannot be negative.'**)  
 age = float(input(**'How old are you now? '**))  
age\_next\_yr = 1 + age  
print(**'You are'**, age\_next\_yr, **'years old next year.'**)

The problem of using an if statement is that it checks for input error only once. If the user enters something invalid again, the if statement won’t catch it. Please see the following test case for the program that uses if statement for error checking:

How old are you now? -15

Error: Age cannot be negative.

How old are you now? -5

You are -4.0 years old next year.

The if statement fails to catch the second error, which will be caught if a loop is used:

How old are you now? -15

Error: Age cannot be negative.

How old are you now? -5

Error: Age cannot be negative.

How old are you now? 15

You are 16.0 years old next year.

Let’s look at another example of input error checking. In an earlier lesson we saw this program:

midterm\_score = float(input(**'Enter midterm score: '**))  
final\_score = float(input(**'Enter final score: '**))  
avg\_score = (midterm\_score + final\_score)/2  
print(**'Your average score:'**, avg\_score)

There are two input values: midterm and final scores. Let’s add two loops to validate these values. These scores must be between 0 and 100. That means they are invalid if they are less than 0 or greater than 100.

midterm\_score = float(input(**'Enter midterm score: '**))  
**while** midterm\_score < 0 **or** midterm\_score > 100:  
 print(**'Score must be between 0 and 100'**)  
 midterm\_score = float(input(**'Enter midterm score: '**))

final\_score = float(input(**'Enter final score: '**))  
**while** final\_score < 0 **or** final\_score > 100:  
 print(**'Score must be between 0 and 100'**)  
 final\_score = float(input(**'Enter final score: '**))

avg\_score = (midterm\_score + final\_score)/2  
print(**'Your average score:'**, avg\_score)

The following is a sample test run of the program:

Enter midterm score: -95

Score must be between 0 and 100

Enter midterm score: 95

Enter final score: 102

Score must be between 0 and 100

Enter final score: 92

Your average score: 93.5

# **4.4 Definite Loops**

A **definite loop** is a loop in which the number of times the loop will iterate can be determined before the loop is executed.

Suppose there are 5 students in a class. They just took a test and they did not do well. The instructor has decided to give 10 extra points to every student. We need a program to calculate the new test score for each student.

Let’s analyze this problem. To calculate and display the new score for one student, we need the following steps:

*Input test score*

*Calculate new test score = test score + 10*

*Display new test score*

To calculate and display the new scores for 5 students, we need a loop to repeat these steps exactly 5 times:

*Repeat 5 times:*

*Input test score*

*Calculate new test score = test score + 10*

*Display new test score*

The logic is clear and simple. We just need to find a way to program the computer to iterate a loop exactly 5 times. How do we do that?

First of all, we need a variable to keep track of the number of iterations a loop has run. Before the loop starts executing, number of iterations is 0 because the loop has iterated 0 times so far. Every time the loop executes its loop body once, we increase this variable by 1. When this variable reaches 5, the loop has iterated 5 times already and it is time to stop.

Variables like this one that are used to count something in a program are called **counters**. Since definite loops use counters to control how many times they iterate, they are also called **counter-controlled loops**.

The logic of how a counter controls a loop can be shown in pseudocode as follows:

*Set iterations to 0*

*While iterations < preset limit:*

*Statements to process one case*

*Increase iterations by one*

Let’s apply this logic to our score problem. The pseudocode will be:

*Set iterations to 0*

*While iterations < 5:*

*Input test score*

*Calculate new test score = test score + 10*

*Display new test score*

*Increase iterations by one*

The following is the Python code:

iterations = 0  
**while** iterations < 5:  
 score = float(input(**"Enter score: "**))  
 new\_score = score + 10  
 print(**"New score:"**, new\_score)  
 iterations = iterations + 1

A counter variable iterations is created and set to 0 right before the loop. We put a statement at the end of the loop body to increase iterations by 1 every time the loop body executes. The loop condition is iterations < 5. This condition is true before iterations reaches 5. After the loop body has executed 5 times, iterations now has the value 5 stored in it. The condition is false and so the program exits the loop. The following is a sample test run:

*Enter score: 71*

*New score: 81.0*

*Enter score: 82.5*

*New score: 92.5*

*Enter score: 66*

*New score: 76.0*

*Enter score: 64*

*New score: 74.0*

*Enter score: 52*

*New score: 62.0*

To help us better understand how the counter variable controls the loop, let’s add one extra statement in the loop body to display the value of iterations right after it has increased.

iterations = 0  
**while** iterations < 5:  
 score = float(input(**"Enter score: "**))  
 new\_score = score + 10  
 print(**"New score:"**, new\_score)  
 iterations = iterations + 1  
 print(**"Number of iterations finished:"**, iterations)

This extra print statement is unnecessary, but it helps us understand the logic of a counter-controlled loop. The following is a sample test run:

*Enter score: 71*

*New score: 81.0*

*Number of iterations finished: 1*

*Enter score: 82.5*

*New score: 92.5*

*Number of iterations finished: 2*

*Enter score: 66*

*New score: 76.0*

*Number of iterations finished: 3*

*Enter score: 64*

*New score: 74.0*

*Number of iterations finished: 4*

*Enter score: 52*

*New score: 62.0*

*Number of iterations finished: 5*

From the sample output above we can clearly see how the counter variable iterations changes from 0 to 1, 2, 3, 4 and finally 5 before the loop exits.

Sometimes we do not know the exact number of times a loop needs to execute at the time we are writing a program. We can still use definite loops in situations like that if we want to. We just need to use a variable to represent this number. When the program runs, we ask the user to enter a value for this variable, i.e. to enter into the program the number of times the loop needs to iterate, before the program starts to execute the loop. Let’s look at an example.

*A fitness trainer is designing a training program for the students. The following formula is used to calculate the target heart rate during training:*

*target hart rate = (220 – age – resting heart rate) \* 0.4 + resting heart rate*

*The target heart rate tells students what heart rate they should reach when they are training. If they do not reach this target heart rate, the intensity of the training is too low. Write a program to calculate the target heart rate for each student in the class. [Note: The formula was made up for this example. Do not use it to calculate target heart rate for anybody in real life.]*

Let’s analyze this problem. It is quite similar to the score problem we saw earlier except that we do not know how many students there are. Therefore, we need to ask the user to enter this number into the program before the loop. We are going to use this input value to control the number of times the loop will iterate. The following is the pseudocode:

*Input number of students*

*Set iterations to 0*

*While iterations < number of students:*

*Input age and resting heart rate*

*Calculate target heart rate = (220 - age – resting rate) \* 0.4 + resting rate*

*Display target heart rate*

*Increase iterations by one*

The following is the Python code:

num\_students = int(input(**"How many students are there? "**))  
iteraions = 0  
**while** iteraions < num\_students:  
 age = float(input(**"Enter age: "**))  
 resting\_rate = float(input(**"Enter resting heart rate: "**))  
 target\_rate = (220 - age - resting\_rate) \* 0.4 + resting\_rate  
 print(**"Your target heart rate is"**, target\_rate)  
 iteraions = iteraions + 1

The following is a sample test run:

How many students in the training program? 3

Enter age: 47

Enter resting heart rate: 75

Your target heart rate is 114.2

Enter age: 38

Enter resting heart rate: 81

Your target heart rate is 121.4

Enter age: 52

Enter resting heart rate: 70

Your target heart rate is 109.2

Let’s look at another example. In a diving competition, each of five judges give a score between 0 and 10 to a diver. Write a program that uses a loop to receive five scores. Calculate and display the total of the five scores.

Analysis: We are going to use a loop to input one score at a time. It will be somewhat like this:

*Set iterations to 0*

*While iterations < 5:*

*Input score*

*Increase iterations by one*

The pseudocode above will input 5 scores from the user. It is incomplete because we need a way to calculate the total of the scores. The question is how to do that. The problem is that we are not storing the 5 scores in 5 variables in the memory. We cannot write an assignment statement to add up the 5 scores all at once. In fact, we are reusing the same variable to read a new score every time we execute the loop body. Every time we input a new score and store it, the previous score is overwritten by the new score. At any moment of the program execution, only one score is stored in the memory because we are reusing the same variable again and again.

To find the total, we must create a variable to store the total and set it to 0 before the loop starts iterating. Every time a new score is entered, we need to add the score to the total immediately before the score is overwritten by the new score entered in the next iteration of the loop. In other words, instead of calculating the total of all 5 scores in one step, we calculate the total gradually by adding one score to the total at a time. It is a running total in the sense that at any moment of the program execution, this variabe tells us the total of all the scores that have been entered so far. Before the loop starts, the running total is 0. After the first score is entered, it is the total of just one score. After the second score is entered, it is the total of the first two scores, and so on until after the last score is entered, it is the total of all 5 scores. The following is the complete pseudocode:

*Set iterations to 0*

*Set total to 0*

*While iterations < 5:*

*Input score*

*Calculate total = total + score*

*Increase iterations by one*

*Display total*

The following is the Python code:

iteraions = 0  
total = 0  
**while** iteraions < 5:  
 score = float(input(**"Enter a score: "**))  
 total = total + score  
 iteraions = iteraions + 1  
print(**"The total score is"**, total)

The following is a sample test run:

Enter a score: 8.9

Enter a score: 8.6

Enter a score: 9.1

Enter a score: 9.0

Enter a score: 8.9

The total score is 44.5

To help us understand the logic behind this program, let’s add an extra print statement to display the running total every time a new score is entered:

iteraions = 0  
total = 0  
**while** iteraions < 5:  
 score = float(input(**"Enter a score: "**))  
 total = total + score  
 print(**"Total of scores entered so far:"**, total)  
 iteraions = iteraions + 1  
print(**"The total score is"**, total)

The following is the sample test run:

Enter a score: 8.9

Total of scores entered so far: 8.9

Enter a score: 8.6

Total of scores entered so far: 17.5

Enter a score: 9.1

Total of scores entered so far: 26.6

Enter a score: 9.0

Total of scores entered so far: 35.6

Enter a score: 8.9

Total of scores entered so far: 44.5

The total score is 44.5

Let’s look at one more example about finding the total of a sequence of input values.

A basketball game has four quarters. Suppose we want a program to handle scores in a basketball game. The program asks the user to enter scores of both teams in each quarter. The program calculates and displays each team’s total score after each quarter. When it asks the user to enter scores, the program should display what quarter it is (quarter 1, quarter 2, quarter 3 or quarter 4) so the user will never get confused. At the end, the program announces which team has won (or it is a tie game).

Analysis: This is similar to the diving score example, except for two things. First, we need to calculate the running totals of two separate sets of scores (team A’s and team B’s). To do that, we need to use two total variables, one for each team. Second, we need to display the quarter number while we are asking for scores. The quarter number actually is just number of iterations finished plus 1. In the first iteration of the loop, number of iterations finished is 0 when we are asking for scores for first quarter. So the quarter number is number of iterations plus 1. In the second iteration of the loop, number of iterations finished is 1 when we are getting scores for the second quarter. The quarter number again is number of iterations finished plus 1. This relationship holds in every iteration. The following is the pseudocode:

*Set iterations to 0*

*Set team A total to 0*

*Set team B total to 0*

*While iterations < 4:*

*Calculate quarter = iterations + 1*

*Display quarter*

*Input team A score in this quarter*

*Input team B score in this quarter*

*Calculate team A total = team A total + team A score in this quarter*

*Calculate team B total = team B total + team B score in this quarter*

*Display team A total*

*Display team B total*

*Increase iterations by one*

*If team A total > team B total:*

*Display team A has won*

*Else if team A total < total B total:*

*Display team B has won*

*Else:*

*Display tie game.*

The following is the Python code:

iteraions = 0  
total\_A = 0  
total\_B = 0

**while** iteraions < 4:  
 quarter = iteraions + 1  
 print(**"Quarter"**, quarter)  
 score\_A = int(input(**"Enter team A score in this quarter: "**))  
 score\_B = int(input(**"Enter team B score in this quarter: "**))  
 total\_A = total\_A + score\_A  
 total\_B = total\_B + score\_B  
 print(**"Team A total after this quarter:"**, total\_A)  
 print(**"Team B total after this quarter:"**, total\_B)  
 iteraions = iteraions + 1  
 print() *# insert a blank line to separate the quarters*

**if** total\_A > total\_B:  
 print(**"Team A has won the game"**)  
**elif** total\_A < total\_B:  
 print(**"Team B has won the game"**)  
**else**:  
 print(**"It is a tie game. We need overtime."**)

The following is a sample test run:

Quarter 1

Enter team A score in this quarter: 22

Enter team B score in this quarter: 19

Team A total after this quarter: 22

Team B total after this quarter: 19

Quarter 2

Enter team A score in this quarter: 20

Enter team B score in this quarter: 27

Team A total after this quarter: 42

Team B total after this quarter: 46

Quarter 3

Enter team A score in this quarter: 25

Enter team B score in this quarter: 18

Team A total after this quarter: 67

Team B total after this quarter: 64

Quarter 4

Enter team A score in this quarter: 21

Enter team B score in this quarter: 21

Team A total after this quarter: 88

Team B total after this quarter: 85

Team A has won the game

# **4.5 Further Reading**

Please read chapter 3 section 3.4 and 3.5 of the textbook. Section 3.4 introduces the iterative control structure, including both definite and indefinite loops. It also discusses a type of bad loops: infinite loops, which are loops that keep running and never stop. Do not mix up infinite loops with indefinite loops although their names are so similar. Indefinite loops can stop when the right value is entered to make the condition false. Infinite loops, on the other hand, will never stop and require some forceful interruption of the program execution. Section 3.5 uses a calendar program to integrate and conclude the whole chapter.