TASK

Control Structures - While Loop

Visit our website

Introduction

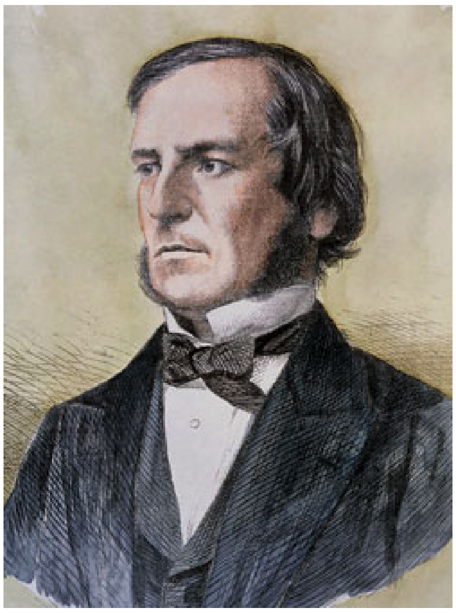
WELCOME TO THE CONTROL STRUCTURES - WHILE LOOP TASK!

In this task, you will be exposed to *loop statements* to understand how they can be utilised in reducing lengthy code, preventing coding errors, and paving the way toward code reusability. This task begins with the *while loop*, which is the simplest loop of those you will learn.

A note from the
Hyperion Team

By now, you should be familiar with the Boolean data type, and the vital role Boolean algebra plays in computer programming. Why is it called a Boolean datatype? It is named after George Boole (1815 - 1864) an English mathematician who helped establish modern symbolic logic and devised Boolean algebra. Boole is best known as the author of The Laws of Thought (1854), which contains Boolean algebra.

Boole was born in Lincoln, Lincolnshire, England. His father, a struggling shoemaker, encouraged him to take an interest in mathematics. Boole went to an elementary school and trade school for a short time, but he mostly educated himself. By the age of fifteen, Boole began teaching. At age nineteen, he set up a school in Lincoln. Boole was appointed as a professor of mathematics at Queen's College, Cork in 1849, despite not holding any university degree.



George Boole

Source: ***[The Grace Library]***

WHAT IS A WHILE LOOP?

A *while loop* is the most general form of loop statement. It takes a code block and keeps executing it while a given condition stays True. The *while statement* repeats its action until this controlling condition becomes False. In other words, the statements indented in the loop repeatedly execute "while" the condition is True (hence the name). The *while statement* begins with the keyword *while*, followed by a boolean expression. The expression is tested before beginning each iteration or repetition. If the test is True, then the program passes control to the indented statements in the loop body; if False, control passes to the first statement after the body.

Syntax:

```
while condition:
   indented statement(s)
```

The following code shows a while statement which sums successive even integers 2 + 4 + 6 + 8 + … until the total is greater than 250. An update statement increments **i** by 2 so that it becomes the next even integer. This is an event-controlled loop (as opposed to counter-controlled loops, like the *for loop*) because iterations continue until some non-counter-related condition (event) stops the process.

```
sum1 = 0
i = 2                        # initial even integer for the sum
while sum1 <= 250:
    sum1 += i
    i += 2                   # update statement, shorthand for i = i + 2
    print(sum1)
```

Run the **example.py** file to see the output of the above program.

GET INTO THE LOOP OF THINGS

Loops are handy tools that enable programmers to do repetitive tasks with minimal effort. To count from 1 to 10, we could write the following program:

```
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
print(7)
print(8)
print(9)
print(10)
```

The task will be completed correctly. The numbers 1 to 10 will be printed, but there are a few problems with this solution:

- **Efficiency:** repeatedly coding the same statements takes a lot of time.
- **Flexibility:** what if we wanted to change the start number or end number? We would have to go through and change them, adding extra lines of code where they're needed.
- **Scalability:** 10 repetitions are trivial, but what if we wanted 100 or even 1000 repetitions? The number of lines of code needed would be overwhelming and very tedious for a large number of iterations.
- **Maintenance:** where there is a large amount of code, one is more likely to make a mistake.
- **Feature:** the number of tasks is fixed and doesn't change at each execution.

Using loops, we can solve all these problems. Once you get your head around them, they will be invaluable in solving many problems in programming.

Consider the following code:

```
i=0
while i < 10:
    i += 1                # shorthand for i = i + 1
    print(i)
```

If we run the program, the same result is produced, but looking at the code, we immediately see the advantages of loops. Instead of executing 10 different lines of code, line 4 executes 10 times - 10 lines of code have been reduced to just 4. This is achieved through the assignment statement used to update the variable `i` in line 3. This adds 1 to the variable `i` in each iteration until `i == 10`. At this point, the logical test (`i<10`) will fail because `i` is no longer less than 10 but equal to 10.

You may change the number 10 to any number you like in order to repeat incrementing the variable and printing it out as many times as you like. Try it yourself!

INFINITE LOOPS

A *while loop* runs the risk of running forever if the condition never becomes False. A loop that never ends is called an infinite loop. Creating an infinite loop is not desirable, to say the least! Make sure that your loop condition eventually becomes False and that your loop is exited.

In setting up any loop, the following three steps are usually used:

1. Declare a counter/control variable. The code above does this using `i = 0`. This creates a variable called `i` that contains the value `0`.
2. Increase the counter/control variable in the loop. In the loop above, this is done with the instruction `i+=1` which increases `i` by one with each pass of the loop.
3. Specify a condition to control when the loop ends. The condition of the for loop above is `i < 10`. This loop will carry on executing as long as `i` is less than `10`. This loop will, therefore, execute ten times.

BREAK STATEMENTS

Sometimes, a very specific condition arises in a loop, and it may be worth adding a check into the loop and exiting if that condition is met. In Python (and most other languages), there exists something called a *break* statement. This statement basically says that the code must exit the loop, even if the condition of the loop hasn't been met. Break statements can only exist in loops: your code won't run if there is a break statement that isn't in a loop.

Consider the following code:

```
my_number = 0

while my_number < 100:
    my_number += 1
    if my_number == 23:
        break
```

This creates a variable called `my_number` and adds 1 to it at each iteration. Normally, the loop would finish when `my_number` reaches 100. However, there is a special condition: when it reaches 23, it will break out of the loop.

CONTINUE STATEMENTS

Like break statements, *continue* statements are typically used when a special condition is met. However, while a break statement exits a loop, the continue statement simply takes you to the next iteration of the loop.

Examine the following code:

```
my_number = 0

while my_number < 100:
    my_number += 1
    if my_number > 23:
        continue
    print(my_number)
```

This is similar to the code for the break statement. However, although `my_number` will reach the value of 100, the code will stop printing after it reaches the value 23.

Instructions

Read and run the accompanying **example files** provided before doing the practical task to become more comfortable with the concepts covered in this task.

Practical Task 1

Follow these steps:

- Create a file called **while.py**.
- Write a program that continually asks the user to enter a number.
- When the user enters "`-1`", the program should stop requesting the user to enter a number,
- The program must then calculate the average of the numbers entered, excluding the `-1`.

- Make use of the *while loop* repetition structure to implement the program.

Rate us

Share your thoughts

Hyperion strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.