Hyperiondev

TASK

Beginner Control Structures — While & For Loops

Visit our website

Introduction

WELCOME TO THE CONTROL STRUCTURES - WHILE & FOR LOOP TASK!

In this task, you will be sequentially exposed to loop statements in order to understand how they can be utilised in reducing lengthy code, preventing coding errors, as well as paving the way towards code reusability.

First, you will learn about the *while* loop, which is the most general form of loop statements. The next statement you'll be exposed to is the *for* loop, which is essentially a different variation of the *while* loop.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact your mentor to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with your mentor. You can also schedule a call or get support via email.

Your mentor is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

A note from the

HyperionDev Team

Python is a high-level language meaning it is closer to human languages than machine languages and is, therefore, easier to understand and write. It is also more or less independent of a particular type of computer. But what was the first high-level language? The answer to that question is Fortran.

Fortran was invented in 1954 by IBM's John Backus. The name Fortran is derived from "Formula Translation" and the language is best suited to numeric computation and scientific computing. Fortran is still used in computationally intensive areas such as numeric weather prediction, finite element analysis, computational fluid dynamics, computational physics, crystallography and computational chemistry.

WHAT IS A *WHILE* LOOP?

A while loop is the most general form of loop statements.

The while statement repeats its action until the 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, the control passes to the first statement after the body.

Syntax:

```
while boolean expression:
    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, about which you will study soon) 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
      print(sum1)
```

Compile and run the example.py file to see the execution output of the above program.

GET INTO THE LOOP OF THINGS

Loops are a handy tool that enables programmers to do repetitive tasks with minimal effort. Say we want a program that can 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 just fine, that is, the numbers 1 to 10 will be printed in the output, but there are a few problems with this solution:

- **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 to solving many problems in programming.

Now let's consider the following code:

```
i=0
while i < 10:
    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 of this piece of code executes 10 times. Ten lines of code (from the previous example) have been reduced to just four. Furthermore, we may change the number 10 to any number we like. Try it yourself — replace the 10 with your own number.

WHAT IS A *FOR* LOOP?

The *for* statement is an alternative loop structure for a *while* statement. This means that a *for* loop is equivalent to a *while* loop and one can apply a *for* loop where a *while* loop could be applied and vice-versa.

The main difference between a *while* loop and a *for* loop is the syntax. A *for* loop allows for counter-controlled repetition to be written more compactly and clearer, therefore making it easier to read.

In the *for* loop below, the condition is that when the variable i (which is an integer) is in the range of 1 to 10 (i.e. either 1, 2, 3, 4, 5, 6, 7, 8, or 9), the indented code in the body of the *for* loop will execute.

The **`range(1, 10)`** specifies that **`i = 1`** in the first iteration of the loop. So, 1 will be printed in the first iteration of this code. Then the code will run again, this time with **`i=2`**, and 2 will be printed out, and so on until **`i=10`**. Now, **`i`** is not in the **`range (1,10)`** so the code will stop executing.

For the function **`range(start index: end index)`**, start index IS included and end index IS NOT included. (Remember we discussed this when we were introduced to variable indexing?)

`i` is known as the index variable as it can tell you what 'iteration' or repetition the loop is on. In each iteration of the *for* loop, the code indented inside the *for* loop is repeated.

```
for i in range(1, 10):
      print(i)
```

This *for* loop prints the numbers 1 to 9. Again, note the indentation and the colon, just like in the *if* statement.

You can also use an if statement within a *for* loop! See the example below:

```
for i in range (1,10):
      if i > 5:
            print(i)
```

This will only print the numbers 6, 7, 8 and 9 because numbers less than or equal to 5 are filtered out.

In order for a *for* loop to function properly, the loop needs to be initialised and a loop test must be performed.

Initialise Loop

The loop needs to use a variable as its counter variable. This variable will tell the computer how many times to execute the loop.

Loop Test

The loop test is a Boolean expression. That is, the loop test is a Python expression such that when it is evaluated, the value of the expression is a Boolean. The loop test expression is evaluated prior to any iteration of the *for* loop. If the condition is true then the program control is passed to the loop body; if false, control passes to the first statement after the loop body.

Update Statement

Update statements assign new values to the loop control variables. The statements typically use the increment `i+=1` to update the control variable. An update statement is always only executed after the body has been executed. After the update statement has been executed, control passes to the loop test to mark the beginning of the next iteration.

Break Statement

Within a loop body, a break statement causes an immediate exit from the loop to the first statement after the loop body. The break allows for an exit at any intermediate statement in the loop.

```
break
```

Using a break statement to exit a loop has limited, but important applications. Let us describe one of these situations.

A program may use a loop to input data from a file. The number of iterations depends on the amount of data in the file. The task of reading from the file is part of the loop body which thus becomes the place where the program discovers that data is exhausted. When the end-of-file condition becomes true, a break statement exits the loop.

INFINITE LOOPS

In selecting a loop construct (either *while* loop or *for* loop) to read from a file, we recognise that the test for end-of-file occurs within the loop body. The loop statement has the form of an infinite loop: one that runs forever. The assumption is that we do not know how much data is in the file. Versions of the *for* loop and the *while* loop permit a programmer to create an infinite loop.

In the infinite version of the *for* loop, each field of the loop is empty. There are no control variables and no loop test. The equivalent *while* loop uses the constant true as the logical expression.

The syntax of infinite *for* and infinite *while* loops are as follows:

```
for(range):
      loop block

while(true):
      loop block
```

Instructions

Before attempting the compulsory tasks below, make sure you've tried running *example.py* in this task folder, as well as all the programs in the Example Programs folders.

Compulsory Task 1

Follow these steps:

- Create a new file called **while.py**
- Write a program that always asks the user to enter a number.
- When the user enters the negative number -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.

Compulsory Task 2

Follow these steps:

- Duplicate your **while.py** file to **while2.py** and do the following:
    - Require the user to enter their name, with only a certain name being able to trigger the loop.
    - Print out the number of tries it took the user before inputting the correct number.
    - Add a conditional statement that will cause the user to exit the program without giving the average of the numbers entered if they enter a certain input.

Compulsory Task 3

Follow these steps:

- Save your program as **tables.py**.
- This program needs to display the multiplication table for any number.
- For example, say the user enters 6. The program must print:

The 6 times table is:
6 x 1 = 6
6 x 2 = 12
...
6 x 12 = 72

Compulsory Task 4

A simple rule to determine whether a year is a leap year is to test whether it is a multiple of 4.

- Write a program to input a year and a number of years.
- Then determine and display which of those years were or will be leap years.

```
What year do you want to start with?      1994
How many years do you want to check?      8

1994 isn't a leap year
1995 isn't a leap year
1996 is a leap year
1997 isn't a leap year
1998 isn't a leap year
1999 isn't a leap year
2000 is a leap year
2001 isn't a leap year
```

Completed the task(s)?

Ask your mentor to review your work!

Review work

Things to look out for:

1. Make sure that you have installed and setup all programs correctly. You have setup **Dropbox** correctly if you are reading this, but **Python or Notepad++** may not be installed correctly.
2. If you are not using Windows, please ask your mentor for alternative instructions.

Rate us

Share your thoughts

HyperionDev 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.