Hyperiondev

TASK

Control Structures - For Loop

Visit our website

Introduction

WELCOME TO THE CONTROL STRUCTURES - FOR LOOP TASK!

In this task, you will be exposed to loop statements to understand how they can be used in reducing lengthy code, preventing coding errors, and paving the way towards code reusability. The looping structure you'll learn about today is the *for* loop, which is essentially a different variation of the *while* loop.

A note from the
Hyperion Team

Python is a high-level language, meaning it is closer to human languages than machine languages. Therefore, it is easier to understand and write. It is also more or less independent of a particular type of computer.

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

WHAT IS A FOR LOOP?

A *for loop* is similar to a *while loop*. Either a *for loop* or a *while loop* can be used to repeat instructions. However, unlike a *while loop*, the number of repetitions in a *for loop* is known ahead of time. A *for loop* is a counter-controlled loop. It begins with a start value and counts up to an end value. A *for loop* allows for counter-controlled repetition to be written more compactly and clearly, making these loops easier to read.

In Python, a *for loop* has the following syntax:

```
for index_variable in sequence:
      statements
```

As you can see, the Python *for loop* starts with the keyword *for*, followed by a variable that will hold each of the values of the sequence as we move through it. The index variable can tell you what iteration the loop is on.

In each iteration (or repetition) of the *for loop* the code that is indented is repeated. The Python *range()* function generates a sequence of numbers, which are used to iterate through a *for loop*. The *range()* function needs two integer values, a start number and a stop number. For the function range(start index: end index), the ***start index is included*** and the ***end index is not included***.

In the *for loop* below, while the variable **`i`** (which is an integer) is in the range of 1 to 10 (i.e. either 1, 2, 3 ,4 ,5, 6 … or 9), the indented code in the body of the loop will execute. **`range(1, 10)`** specifies that **`i = 1`** in the first iteration of the loop, so 1 will be printed in the first iteration of the code example below. Then the code will run again, this time with **`i=2`**, and 2 will be printed out, etc., until **`i=10`**. Now **`i`** is not in the range (1,10), so the code will stop executing.

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

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

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

You can use an *if statement* within a *for loop*!

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

The code in the example above will only print the numbers 6, 7, 8, and 9 because numbers less than or equal to 5 are filtered out.

For a *for loop* to function properly, the following things must happen:

- **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 in Python that evaluates to either True or False. The loop test expression is evaluated before 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 statement typically uses the increment `i+=1` to update the control variable. An update statement is always 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.

A loop could also contain a *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. Have a look at the example below. Copy and paste it and try it out!

```
num_list = [1, 2, 3, 4, 5]
found = False
num = int(input("Input a number from 1 to 10 and find out if it's in
our list:"))
if num<1 or num>10:
      print("Number out of range")
else:
      for number in num_list:
                  if num == number:
                        found = True
                        break
      print(f'List contains {num}: {found}')
```

Did you know?

Using a *break statement* to exit a loop has some important applications in working with files. Imagine a program which uses a *for loop* to input data from a file. The number of iterations of the for loop will depend on the amount of data in the file. The task of reading from the file is part of the loop body, which becomes the place where the program discovers that data is exhausted. When the end-of-file condition becomes True, a *break statement* can be used to exit the loop.

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 *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.

NESTED LOOPS

A *nested loop* is simply a loop within a loop. Each time the outer loop is executed, the inner loop is executed right from the start. That is, **all the iterations of the inner loop are executed with each iteration of the outer loop**.

The syntax for a nested *for loop* in another *for loop* is as follows:

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

The syntax for a nested *while loop* in another *while loop* is as follows:

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

You can put any type of loop inside of any other kind of loop. For example, a *for loop* can be inside a *while loop* or vice versa.

```
for iterating_var in sequence:
  while condition:
      statement(s)
  statements(s)
```

The following program shows the potential of a nested loop, in this case used to output times tables:

```
for x in range(1, 6):
    for y in range(1, 6):
        print(f"{x} * {y} = {x*y}")
    print("")
```

When the above code is executed, it produces following result :

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10

3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15

4 * 1 = 4
4 * 2 = 8
4 * 3 = 12
4 * 4 = 16
```

```
4 * 5 = 20

5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
```

TRACE TABLES

As you start to write more complex code, trace tables can be a really useful way of desk-checking your code to make sure the logic of it makes sense. You can do this by creating a table where you fill in the values of the variables for each iteration. This is particularly useful when you are iterating through nested loops. For example, let's look back to the code above and create a trace table:

x	y	x*y
1	1 2 3 4 5	1 2 3 4 5
2	1 2 3 4 5	2 4 6 8 10
3	1 2 3 4 5	3 6 9 12 15
4	1 2 3 4 5	4 8 12 16 20
5	1 2 3 4 5	5 10 15 20 25

As you can see, because of the nature of nested loops, the inner loop iterates 5 times before the outer loop is iterated again. That means that **x** will remain the same value while **y** loops through all its iterations. Then, once the outer loop iterates, the inner loop restarts with another 5 iterations. This is represented by the trace table, where **y** cycles from 1-5 for the same value of **x**. Practise drawing trace tables for your upcoming tasks to assist you with the logic — they can be very helpful when coding gets tricky!

Instructions

- Read through, and run, the code examples provided. Feel free to tweak the code there to see what happens - this can be a valuable learning mechanism!

Compulsory Task 1

Follow these steps:

- Create a new Python file in this folder called **pattern.py**.
- Write code to output the star pattern shown below, using an *if-else statement* in combination with a single *for loop* (it's really easy with two but using only one takes a little more thought!):

```
*
**
***
****
*****
****
***
**
*
```

Thing(s) to look out for:

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

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.