TASK

Iteration

Visit our website

Introduction

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 towards code reusability. This task begins with the *while loop*, which is arguably the simplest loop of those you will learn.

We'll then move on to the *for loop* structure to understand how it can be employed to iterate over sequences, executing a block of code a predetermined number of times determined by the sequence's length.

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.

```
# Initialise total sum to 0
total_sum = 0

# Start with the first even integer
even_integer = 2

# Loop until the total sum exceeds 250
while total_sum <= 250:
    # Add the current even integer to the total sum
    total_sum += even_integer

    # Move to the next even integer
    even_integer += 2

    # Print the current total sum
    print(total_sum)
```

Run the **while_example1.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 one to ten, 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 one to ten 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:** ten repetitions are trivial, but what if we wanted 100 or even 1,000 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 ten different lines of code, line four executes ten times – ten lines of code have been reduced to just four. This is achieved through the assignment statement used to update the variable `i` in line three. 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 ten but is equal to ten.

You may change the number ten 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!

In setting up a loop, here are a few helpful steps that are often 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 *while loop* above is `i < 10`. This loop will carry on executing as long as `i` is less than `10`. This loop will, therefore, execute 10 times.

However, it is important to note that these steps do not apply to **all** types of loops. For instance, in *for loops*, you don't need to explicitly declare a counter or specify a condition to end the loop.

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.

Using a while loop with boolean data types

Here's a simple example:

```
# Initialise the control variable to True
keep_running = True

# Start the loop
while keep_running:
    print("The loop is running.")
    # Set the control variable to False to exit the loop
    keep_running = False
print("The loop has ended.")
```

In this example:

- The `while` loop continues to execute as long as `keep_running` is `True`.
- Inside the loop, `keep_running` is set to `False` to ensure the loop exits after the first iteration.

TERMINATING INFINITE LOOPS

When faced with an infinite loop during programming, it is important to know how to terminate it to avoid freezing your IDE or terminal. Here is how you can do it **using keyboard interrupt:**

- **Windows:** Press **Ctrl+C**
- **Mac:** Press **Ctrl+C**
- **Linux:** Press **Ctrl+C**

Pressing **Ctrl+C** on your keyboard will send an interrupt signal to the running program, which effectively stops it. This is a helpful shortcut that every programmer should be familiar with to interrupt infinite loops.

Steps to follow with terminating infinite loops:

- **Identify an infinite loop:** If your program is running indefinitely and not responding as expected, move on to the next step.
    - **Use Ctrl+C:** In your terminal or the integrated terminal of your IDE, press **Ctrl+C** to stop the program.
    - **Update the code:** Once the program has stopped, review the loop conditions and logic to ensure that the loop can exit as expected.

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
# Loop until my_number is less than 100
while my_number < 100:
    my_number += 1

# If my_number equals 23, exit the loop
    if my_number == 23:
        break
```

This creates a variable called `my_number` and adds one to it at each iteration. Normally, the loop would finish when `my_number` reaches 100. However, we have added 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

# Loop until my_number is less than 100
while my_number < 100:
    my_number += 1

    # Skip further processing if my_number exceeds 23
    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.

Alright, having grasped the fundamentals of *while loops*, let's transition into the exploration of *for loops* and how they can make your code more efficient and concise.

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. Therefore, a *for loop* is somewhat 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 can be 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 one to ten (i.e. either one, two, three, four, five, six, seven, eight, or nine), the indented code in the body of the loop will execute. The `range(1, 10)` specifies that `i = 1` in the first iteration of the loop, so one will be printed in the first iteration of the code example below. Then the code will run again, this time with `i = 2`, and two will be printed out, etc., until `i = 10`. At this point, `i` is no longer in the `range (1,10)` (remember that the end index is excluded), 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 one to nine. Again, note the indentation and the colon, similar to when you use 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 six, seven, eight, and nine because numbers less than or equal to five 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!

```
# Define a list of numbers
num_list = [1, 2, 3, 4, 5]

# Initialise a flag to indicate if the number is found
found = False

# Prompt the user to input a number from 1 to 10
num = int(input("Input a number from 1 to 10 and find out if it's in our
list:"))

# Check if the input number is within the valid range
if num < 1 or num > 10:
    # Inform the user if the number is out of range
    print("Number out of range")
else:
    # Iterate through the list to check if the number is present
    for number in num_list:
        if num == number:
            # Set the flag to True if the number is found and exit the loop
            found = True
            break
```

```
    # Print the result indicating whether the number is in the list or not
    print(f'List contains {num}: {found}')
```

Did you know?

Using a *break statement* to exit a loop has some important applications when 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_1 in sequence:
   for iterating_var_2 in sequence:
      statements(s)
   statements(s)
```

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

```
while condition_1:
   while condition_2:
      statement(s)
   statement(s)
```

You can put any type of loop inside of any other kind of loop. The syntax for a nested ***while loop* in a *for loop***, for example, is as follows:

```
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:

```
# Outer loop, iterating over the range 1 to 5
for x in range(1, 6):
    # Inner loop
    for y in range(1, 6):
        # Print the multiplication result of x and y
        print(f"{x} * {y} = {x*y}")
    print("")
```

When the above code is executed, it produces the 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 its logic 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 five 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 five iterations. This is represented by the trace table, where **y** cycles from one to five 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!

Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

Instructions

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

Auto-graded 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. Please be aware that `0` is not a valid input.
    - Hint: think about how you might **exit** the loop if `-1` is entered.
- The program must then calculate the average of the **valid** numbers entered, excluding the `-1` and `0`.
- Use a while loop to achieve the continuous prompting and number collection.

Auto-graded task 1

Follow these steps:

- Create a new Python file in this folder called **pattern.py**.
- Write code to output the arrow pattern shown below, using an *if-else statement* in combination with a *for loop*
    - You are **allowed** to use more than one for loop. But use only one for loop if you wish to challenge yourself):

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

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.