ICS3U: Conditional Iteration

Looping Based On Conditions With `while`

A `for` loop executes code for a fixed number of times, based on the number of elements in a sequence (such as `range`, or a string). Sometimes we want to repeat code based on a condition instead. For example, we may want code to run as long as a value does not exceed a certain threshold. Since we do not know how many iterations this might take, a `for` loop is not appropriate. Instead, we can use another type of look that allows for **conditional iteration**. This type of loop uses the `while` keyword.

The general format of a while loop is as follows.

```
while conditions:
   # CODE TO EXECUTE
```

The program below generates random integers as long as the current value is less than 8. If it is, then the condition `num < 8` will be `True`, and the loop will continue to iterate. As soon as `num` is 8 or greater, the conditoin `num < 8` will be `False` and the loop will terminate. Try running it several times, and you will see that the output always ends with a number that is 8 or greater.

```
Import random
num = random.randint(1, 10)
print(num)
while num < 8:
   num = random.randint(1, 10)
   print(num)
```

Unlike a `for` loop, a `while` loop does not keep track of counting in an associated variable. Instead, we are responsible for manipulating the values of any variables ourselves. For example, consider the for loop below that displays the values 1 through 5 to the screen.

```
for count in range(1, 6):
   print(count)
```

The variable `count`, defined in the header of the `for` loop, is automatically incremented by 1 after every iteration. To replicate the same thing using a while loop, we can do something like below.

```
count = 1
while count <= 5:
   print(count)
   count += 1
```

In this loop, the condition being checked is `count <= 5`. As long as the value of `count` is 5 or less, the condition will be `True` and the loop will run. Otherwise, it will be `False` and the loop will stop. The variable `count` will never change its value unless we explicitly change it ourselves. So, it is necessary to include the statement `count += 1` inside of the loop, or count would never increase. This can lead to some interesting behaviour, which is covered in the next section of this lesson.

Just like `for` loops, it is possible to terminate a `while` loop early by using `break`. This causes program execution to resume on the next line immediately following the loop. The code below reads values from the user, and determines the average of all positive values. If the user enters a negative value, then the condition `number < 0` is `True`, and the `break` statement inside of the `if` block is executed, terminating the loop.

```
count = 0
total = 0
number = 0
while number >= 0:
   number = int(input("Please enter a number: "))
   if number < 0:
      break
   else:
      count += 1
      total += number
if count != 0:
   average = total / count
   print("The average of the positive values is", average)
else:
   print("No positive values entered.")
```

Multiple conditions can be specified in the header of a `while` loop using logical operators. For example, the following code will loop until the user enters either a capital or lowercase A. Try running it with several letters to test it.

```
letter = input("Please enter an A: ")
while letter != "A" or letter != "a":
   letter = input("That's not an A. Try again: ")
print("Thanks for the A.")
```

Did you discover the logical error in the code? It certainly does *not* behave as expected. No matter what letter is entered, the code *always* displays "That's not an A. Try again: ". This is because the wrong logical operator was used. Consider the case where `letter` has a value of 'A'. The first condition checked is `letter != "A"`, which is `False`; however, the second condition is `letter != "a"`, which is `True` – a capital 'A' is not the same as a lowercase 'a'. Since we have used or, then the combination of comparisons has a value of `True`, and the `while` loop continues to run. Instead of `or`, we must use `and` instead, as in the following *correct* code.

```
letter = input("Please enter an A: ")
while letter != "A" and letter != "a":
   letter = input("That's not an A. Try again: ")
print("Thanks for the A.")
```

Infinite Loops

Let's return to the `while` loop that counts from 1 to 5, but this time, we will remove the final line where we update the value of `count`. The new program looks somehing like this.

```
count = 1
while count <= 5:
   print(count)
```

When we run the program, the output looks something like this.

```
1
1
1
1
1
…
```

The interpreter produces an endless stream of 1s, which can only be stopped by cancelling program execution. This is an example of an **infinite loop**. Since the condition `count <= 5` is *always* `True`, the loop never stops. In most cases, this is not what we want to happen, but in certain circumstances, it can be useful to create an infinite loop, and specify exit conditions inside of the loop using `break`. There are many ways to create an infinite loop, but the easiest is probably to use the statement `while True`. Since `True` always evaluates to `True`, and cannot change its value, the loop will run forever unless it is explicitly commmanded to stop (or crashes due to an error).

One such instance is when there are too many conditions that should cause a loop to terminate to comfortably list in the `while` header. For example, assume that there are 12 separate conditions that could cause a loop to stop. We could use a `while` loop like this:

```
while condition1 or condition2 or condition3 ... or condition11 or condition12:
   # CODE TO EXECUTE
```

This is fairly awkward, and difficult to read as a single line. Instead, we could structure the code into an organized `if` block, each of which contains a `break` statement to terminate the loop.

```
while True:
   if condition1:
      print("Exiting because of condition1.")
      break
   elif condition2:
      print("Exiting because of condition2.")
      break
   …
   else:
      print("Exiting because of condition12.")
      break
```

The downside to using an infinite loop in this way is that it is not always clear what the conditions are that will cause the loop to terminate. Instead, these may be buried deep inside of the loop, and might require the reader to search for them. A `while` loop that specifies the conditions in its header makes it easy to see immediately when the loop will stop.