ICS3U: Controlling Iteration

Terminating `for` Loops Early Using `break`

Consider the program below, which displays the squares of the values 0 through 6.

```
for val in range(7):
   sqr = val**2
   print(val, "squared is", sqr)
```

When run, the program outputs the following.

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
```

Suppose we want the loop to stop early, if the value of a square exceeds a certain value. Python provides the `break` statement to do this. By placing a `break` statement inside of an `if` block, we can cause a loop to terminate when a condition is met. The following code checks if the value of the square is greater than 20 and, if so, stops the loop from running any further.

```
for val in range(7):
   sqr = val**2
   if sqr > 20:
      break
   print(val, "squared is", sqr)
```

The modified program outputs the following.

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
```

While `range(7)` would normally cause the loop to execute 7 times, the condition `sqr > 20` is `True` when `val` has a value of 5 (since $5^2 = 25$). This causes the interpreter to execute the `break` statement inside of the `if` block, and the loop terminates immediately.

Python programmers often use `break` statements to stop a loop when further processing is no longer necessary. For example, the following code checks whether a given string consists solely of letters. If it does, then a message is displayed. If non-alphabetic characters are encountered, an alternate message is displayed.

```
string = input("Enter a string of characters: ")
letters = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
all_letters = True
for char in string:
   if char not in letters:
      all_letters = False
if all_letters:
   print("All letters.")
else:
   print("Not all letters.")
```

After reading a string from the user, the program creates a variable that holds a boolean data type. This type of variable is called a **flag**, and is common in many computer programs. A flag is "set" to either `True` or `False` when a certain condition is met. Later, the value of the flag can be checked to determine if the condition was met or not. In this case, we assume that the string is composed entirely of letters, so we set all_letters to `True`.

After this is done, the loop checks every character in the string to see if it is contained in the set of lowercase and uppercase letters in `letters`. If at any time a character is not in `letters`, `all_letters` is set to `False`. Eventually the last character is processed, and the loop ends. The `if` block after the loop displays the state of the string.

The code below does the same thing, but it stops checking characters once the first non-letter character is found. Note the `break` statement after all_letters is set to `False`.

```
string = input("Enter a string of characters: ")
letters = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
all_letters = True
for char in string:
   if char not in letters:
      all_letters = False
      break
if all_letters:
   print("All letters.")
else:
   print("Not all letters.")
```

If the user enters 80 characters, and the third character is a number or a space, then it makes no sense to continue checking the remaining 77 characters. The loop can safely stop once the flag is set.

If you run both of the above programs, you will probably not notice much of a difference. This is because 80 characters is not really that long of a string, so processing times for both programs are very fast. To get an idea of how much more efficient a program can be, consider the program below that determines if a number is prime or not.

```
num = int(input("Enter a value greater than 1: "))
prime = True
for val in range(2, num):
    if num % val == 0:
        prime = False
if prime:
    print(num, "is prime.")
else:
    print(num, "is not prime.")
```

After reading a value, `num`, from the user, the program sets the flag `prime = True`, then begins to test potential factors of `num`. If `num` is evenly divisible by the value, it cannot be a prime number, so the flag is set to `False`. When the loop finishes, the program checks the flag to output the appropriate message.

The program works very well for small values, but takes an increasingly long time to run for large values of `num`. Try running the code with a value of 823 204 347 179. Note that this may take several *hours* to complete, even on a relatively fast computer. When it is done (or when you come to your senses and stop the program), try running the code below which adds a `break` statement after `prime` is set to `False`.

```
num = int(input("Enter a value greater than 1: "))
prime = True
for val in range(2, num):
    if num % val == 0:
        prime = False
        break
if prime:
    print(num, "is prime.")
else:
    print(num, "is not prime.")
```

The program finishes nearly instantly. This is because 823 204 347 179 = 241 271 × 3 411 949. When `val` has the value 241 271, then the condition `num % val == 0` is `True`. In this version of the program, the `if` statement is executed and the `break` statement inside causes the loop to terminate. This is fine, since if we find any factor other than 1 and the number itself, it cannot be prime. In the first version of the program, without a `break` statement, the loop continues until the very last value (823 204 347 178) is checked. The second version allows us to ignore over 800 billion values. That's a huge improvement!

Using `else` with `for` Loops

In addition to `if` blocks, an `else` statement can be attached to a `for` loop. Code inside of this `else` block will execute once the loop has completed its final iteration. Note that this will only happen if the loop completes on its own accord. A loop that terminates when it encounters a `break` statement will not execute the code inside of the `else` block. The code below contains an else block that will only execute if the value of `num` is 5 or less.

```
num = int(input("Enter a value: "))
for count in range(num):
   if count == 5:
      break
   print(count, end=" ")
else:
   print("DONE")
```

When `num` is 3, the program produces the following.

```
0 1 2 DONE
```

When `num` is 7, it produces the following. When `count` took on a value of 5, the condition `count == 5` was `True`, so the `if` block containing the `break` statement was executed.

```
0 1 2 3 4
```

As another example, we can rewrite the letter-checking program from earlier to use an `else` statement instead of a flag.

```
string = input("Enter a string of characters: ")
letters = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
for char in string:
   if char not in letters:
      print("Not all letters.")
      break
else:
   print("All letters.")
```

If the string contains a non-alphabetic character, then the condition `char not in letters` will be `True`, and the `break` statement will stop the loop. The `print` statement inside of the `else` block will not be run. On the other hand, if the condition is never `True` then the loop will finish naturally and the `print` statement inside of the `else` block will be displayed. Try it and see.

The `continue` Command

Sometimes we encounter a situation where we want a loop to continue iterating, but to ignore any additional commands inside of the loop. The `continue` keyword acts in a similar manner to `break`, but instead of exiting the loop completely, program flow returns immediatly to the beginning of the loop. Thus, while `break` stops a *loop*, `continue` stops an *iteration*. Try running the following code to see what happens.

```
vowels = "AEIOU"
for letter in "ABCDEFG":
   if letter in vowels:
      continue
   print(letter)
```

The program prints all letters that are not vowels, and skips the ones that are. This is because when a letter is a vowel, the condition `letter in vowels` is `True`, and the `if` block containing the `continue` statement is executed. This makes the loop restart its next iteration, ignoring the `print` statement.

Using `break`, `else` and `continue` statements is usually a choice for either improved efficiency or better readability. There are often other ways to write programs that do not involve these flow control commands, but they are available if they can be used effectively.