ICS3U: Counted Iteration

Generating Sequences of Integers With `range`

One thing that computers are very good at is counting. It should not come as a surprise, then, that many programs utilize this ability. One thing that we will do fairly often in our own programs is to generate sequences of integers for counting purposes. Python enables this via the `range` function.

The `range` function takes up to three arguments: the start value, the end value, and a step value. Like the `randrange` function in the `random` module, `range` does not include the ending value. To generate all of the integers from 5 to 12, for example, one must use `range(5, 13)`. To generate all of the odd numbers between 17 and 29, use `range(17, 30, 2)` where the final argument indicates that we are counting by 2. To count backward, use a negative step value, as in `range(20, 10, -1)`.

If only a single argument is given, then it is assumed to be the end value. The start value in this case is 0. So `range(6)` will generate the values 0, 1, 2, 3, 4 and 5. This is for practical reasons, as we will see later.

A `range` object is its own data type. Although it consists of integers, it cannot be used like an `int`: mathematical operations, such as addition, will not work.

```
>>> r = range(1, 10)
>>> r + 2
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'range' and 'int'
```

In order to use `range` objects in our programs, we will need to look at a few other commands first.

Checking For Inclusion With `in`

Any non-empty sequence type – whether it is a string, a tuple or a list – consists of one or more **elements**. For example, the string "Python!" contains seven elements (six letters and one punctuation character). To check if an element is included in a sequence, Python provides the `in` keyword. This is done in a manner similar to the one below.

```
>>> "x" in "wxyz"
True
>>> 7 in (3, 4, 5)
False
```

Python will always generate a boolean value when using in: `True` if the element is contained in the sequence, and `False` if it is not. As long as a variable is a sequence type, it can be used in place of a literal value.

```
string = input("Please type something: ")
if "B" in string:
   print("The string contains the letter B.")
else:
   print("There is no letter B in the string.")
```

Although the previous examples have checked for the presence of a single character, there is no such limitation when using `in`, as shown below.

```
>>> "elf" in "shelf"
True
>>> "pig" in "ping"
False
```

Although all of the characters in “pig” are contained in “ping” (even in the same order) in the previous example, the characters are non-consecutive, and so `in` generates a value of `False`.

A common mistake is to try to use in with a non-sequence data type, such as an integer.

```
>>> num = 123
>>> 2 in num
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: argument of type 'int' is not iterable
```

Since a `range` contains a sequence of integers, one can check if a particular value is included using `in`.

```
>>> integers = range(5, 12)
>>> 8 in integers
True
```

Counted Iteration With `for`

Besides processing commands sequentially and making decisions, another important aspect of programming is repeating code. In is not efficient for us to write the same, or very similar, commands over and over again. Consider the following code:

```
import random
total = 0
roll = random.randint(1, 6)
print("You rolled a", roll)
total += roll
roll = random.randint(1, 6)
print("You rolled a", roll)
total += roll
roll = random.randint(1, 6)
print("You rolled a", roll)
total += roll
print("The total of the rolls is", total)
```

The program simulates rolling three dice, and adds the values of the rolls. Many of the lines are identical. It would be great if it was possible to simply tell the program to roll a die three times, adding its value each time. Fortunately, this is fairly easy to do in Python.

In programming terms, a **loop** is a section of code that repeats itself. Python allows for two main types of loops. The first is *counted* looping, where a loop iterates over a fixed number of values. The second, which we will explore later, is *conditional* looping, where a loop iterates while one or more conditions are `True`.

Here is the general structure of a counted loop.

```
for variable in sequence:
   # CODE TO RUN
```

To begin a counted loop we start by using the `for` keyword, followed by a variable name. This variable will take on various values as the loop executes. These values are found `in` some specified sequence. In essence, the loop will run "for" every element in a sequence.

Remember that `range` will generate a sequence of integers. As such, it is extremely common to use `range` to create an iterative loop. The following code will display a message five times.

```
for count in range(5):
   print("Hello")
```

Recall that when `range` has only one argument, the start value is 0 and the end value is not included. Thus, the `range` object will contain the values 0, 1, 2, 3 and 4. On the first run through the loop, `count` is assigned the first value in the sequence, 0. The code inside of the indented block is executed and, upon reaching the last command, returns to the beginning of the loop. On each subsequent iteration, `count` is reassigned the next value in the sequence. Since there is a total of five values in the sequence, the loop will execute a total of five times.

In many cases, the value of the loop variable itself is used as output or as part of a calculation. The following example displays the value of $n!$ for all values of n between 1 and 20. Note how n is referenced inside of the `print` statement, just like any other variable would be.

```
import math
for n in range(1, 21):
   print(n, "! = ", math.factorial(n), sep="")
```

And here is an example that simulates a rocket launch. Note the negative step value to count backward.

```
for count in range(10, 0, -1):
   print(count)
print("Liftoff!")
```

To return to the dice example earlier, we can rewrite the code using a for loop instead.

```
import random
total = 0
for count in range(3):
   roll = random.randint(1, 6)
   print("You rolled a", roll)
   total += roll
print("The total of the rolls is", total)
```

This is not only shorter and cleaner, but it is easier to change the code if necessary. For example, if we wanted to roll the die 30 times instead, we would not need to add 81 additional lines of code to accommodate the rolling, displaying and updating of the total.

It is not necessary to use `range` if a `for` loop will iterate over another sequence type, such as a string. The code below reads a string from the user and displays each character on a separate line.

```
word = input("Enter a word: ")
for letter in word:
   print(letter)
print("DONE")
```

Try running the code and entering the word "Yes". On the first iteration, `letter` will be assigned the value 'Y', which is then displayed to the screen. On the next two iterations, `letter` is assigned the values 'e' and 's' respectively. When all values in the string have been exhausted, the loop terminates and program execution moves on to the next line, which displays "DONE".

One of the main issues that beginning programmers have with loops is determining what code will go inside of a loop, and what code must remain outside. There is a logical error in the code below. Run it several times and see if you can identify what is happening.

```
for count in range(5):
   total = 0
   num = int(input("Enter a number: "))
   total += num
print("The total of the values you entered is", total)
```

The program will always display the last value entered by the user. This happens because the assignment total = 0 occurs inside of the loop. Each time the loop is executed, total is reset to zero. We want this action to occur only once, *before* the loop starts. Moving it outside of the loop, as in the code below, fixes the error. In general, anything that you want to run more than once should be inside of a loop, whereas things that should only be done once must be outside.

```
total = 0
for count in range(5):
   num = int(input("Enter a number: "))
   total += num
print("The total of the values you entered is", total)
```