ICS3U: Creating Simple Functions

From the beginning, we have used several of the various functions that Python has to offer, such as `print`, `input` or `range`. We have also learned how to import additional functions, such as those in the `math` and `random` modules, for use in our programs. But we are not limited to the functions defined in Python, or those in the standard modules included in most distributions. It is possible to write our own functions and reference them when necessary. Functions are created using the following syntax.

```
def function_name(arguments):
   # CODE TO EXECUTE
```

A function has two main components: the **heading**, and the **body**. The heading consists of the `def` keyword, which tells the Python interpreter that a new function is being **defined**. This is followed by the name of the function, which uniquely identifies it within a program. Function names must follow the same rules as variables – letters, numbers and underscores only. After the function name is a list of **arguments**, variables passed into the function that will be assigned values. Like `if` statements and loops, the body is indented. It contains the code that is to be executed each time the function is run.

Let's assume that we want to write a function that displays the sum of an arbitrary number of six-sided dice. The following function can be used to calculate and display the sum.

```
def sum_of_dice(num_dice):
   total = 0
   for roll in range(num_dice):
      die = random.randint(1, 6)
      print("You rolled a", die)
      total += die
   print("The sum of the", num_dice, "dice is", total)
```

The function heading contains the name of the function, `sum_of_dice`, and a single argument, `num_dice`. The body of the function creates a running `total`, and uses a loop to generate random values for a number of times equal to the value of `num_dice`.

If you try running the code above, you will find that it does not run. That is because it is necessary to **call** a function using its name in order for the code inside of it to be executed. Let's add the following lines to our program, *after the function*.

```
import random
sum_of_dice(5)
```

This is necessary because the Python interpreter must first load the function into memory before it can be called. It is also useful to define all functions toward the top of a program, so that anyone who is reading the code does not have to search through the entire program to find where they are defined.

The `import` statement is necessary because the function uses `randint`. The function call itself consists of the function's name, followed by values for each required argument. Since `sum_of_dice` takes one argument, one value (5) has been provided. After adding these two lines, try running the program. You should find that the program rolls five dice, and displays their sum.

Anytime we wanted to roll any number of dice, we can call the `sum_of_dice` function and pass it a specific value as an argument. Note that it is possible to pass an existing variable into the function as an argument as well. For instance, replace the line `sum_of_dice(5)` with the following two lines.

```
dice_to_roll = int(input("How many dice to roll? "))
sum_of_dice(dice_to_roll)
```

This will prompt the user to enter some number of dice, and this value will be passed to `sum_of_dice`. Since Python will use the current value of the variable (assuming it is the correct type), it will execute in exactly the same way as when a literal value is supplied.

Not all functions take arguments. For instance, the function below simply prints "Hello" to the screen. There is no need to pass any values to `say_hello`, since there is no use for any in the body.

```
def say_hello():
    print("Hello")
```

Even though the function does not take any arguments, it is important to include the brackets in the function call. Beginning programmers often omit them, assuming that they are not necessary. Issuing the following call produces the expected output.

```
say_hello()
```

On the other hand, the following commands seems to do nothing.

```
say_hello
```

Note that the program actually does *something*. It's just that it does not do anything *obvious*. When a function's name is referenced as above, Python determines the memory address of the function, but does not execute any code within it. This can be seen using a `print` statement.

```
>>> print(say_hello)
<function say_hello at 0x171c9e0>
```

As another example, consider the problem of finding the number of diagonals in a convex polygon, such as a square or a hexagon. A diagonal is a line that connects two vertices through the interior of the polygon. They are not adjacent – these would be edges, not diagonals. Since any two vertices in a triangle are adjacent, a triangle has zero diagonals. A rectangle has two diagonals. A pentagon (5 edges) has five, and a hexagon (6 edges) has nine. Try drawing them to see for yourself. Now, how many diagonals are in a hectogon, which has 100 edges?

To answer this question, consider the process of constructing a diagonal. First, we select a vertex of the polygon, for which there are n possibilities. Next, we must select a second vertex. There are n-3 possibilities – it must be different from the first one, and must not be one of the two adjacent vertices. Note, however, that the same diagonal can result from two different orderings. For example, we may select vertex 1 then vertex 5, *or* we may select vertex 5 then vertex 1. Thus, we have overcounted by a factor of two. Putting this all together, we arrive at the conventional formula for the number of diagonals, d, in an n-edged polygon: $d=\frac{n(n-3)}{2}$.

Here, then, is a function that calculates the number of diagonals in a polygon.

```
def count_diagonals(edges):
   diagonals = edges * (edges - 3) / 2
   print("A", edges, "edged polygon has", diagonals, "diagonals.")
```

To determine the number of diagonals in a hectogon, we can call the function as follows.

```
count_diagonals(100)
```

One of the main advantages of functions is code reuse. Once a function is created, it can be used anywhere in a program. This means that we can bundle multiple lines of code inside of a function, and then use a single line to execute it whenever we need to. For example, consider a card game where each player must draw some number of cards from the deck on their turn. This involves several actions: adding a card to the hand, removing the card from the deck, and repeating this based on the number of cards required. By bundling this into a function, this can be called multiple times throughout the course of a game, for different players, with few changes to code necessary. We will see a greater importance of functions as this course progresses.