ICS3U: Returning Values

In the previous section, we learned how to create functions in Python such that, when they were called, they performed one or more actions and displayed the result to the screen. Consider a simple function that finds the product of two numbers.

```
def find_product(a, b):
   product = a * b
   print("The product is", product)
```

If we wanted to call this function to find the product of 3 and 7, we could write `find_product(3, 7)` and it would display `The product is 21`. While this gives us our answer, it isn't very useful. We can't really do anything with that value other than see it on the screen. It would be better if we could hold on to that value, in case we need it to do other calculations. We can achieve this by **returning** a value from the function back to the main program. Python uses the `return` keyword, followed by one or more values, to do this. Below is a reworked version of the function, and the main program.

```
def find_product(a, b):
   product = a * b
   return product

prod = find_product(3, 7)
print("The product is", prod)
```

Once the function has calculated the value of `product`, it is returned back to the main program. Note that when calling the function, we must specify a variable that will be assigned the returned value(s). In the example above, the returned value is assigned to `prod`. Without assigning this value to a variable, there would be no way for us to reference the value once the function had completed.

Sometimes it is useful to write functions that simply check if one or more criteria are met. If they are, then the overall set of conditions is `True`, and if they are not, it is `False`. For instance, below is a function that checks whether a given integer is even or odd.

```
def is_even(n):
   if n % 2 == 0:
      return True
   else:
      return False
```

Inside of the function, the `if` and `else` statements handle the cases where the integer is divisible by two or not. Depending on the condition that is met, the function will return either `True` if the integer is even, or `False` if it is odd. To use this function in a program, we can call `is_even` as our conditions in an `if` block.

```
num = int(input("Enter an integer: "))
if is_even(num):
   print(num, "is an even number.")
else:
   print(num, "is an odd number.")
```

If the user enters a value of 8, then `is_even` returns a value of `True`, since the condition `n % 2 == 0` is `True`. This in turn means that the if statement will be `True`, and the program will state that 8 is an even number. On the other hand, if the user enters a value of 5, then is_even returns a value of `False` and the `else` statement in the main program is executed instead.

When the `return` keyword is read, program execution leaves the function and *immediately* returns to the point in the program from where it was called. Any code after a `return` statement will not be executed. For instance, in the code below nothing is ever printed to the screen because the `print` statement follows the `return` statement.

```
def add_numbers(a, b):
   total = a + b
   return total
   print("The sum is", total)

sum = add_numbers(3, 5)
```

Instead, we might place the `print` statement following the call to `add_numbers`.

```
def add_numbers(a, b):
   total = a + b
   return total

total = add_numbers(3, 5)
print("The sum is", total)
```

Sometimes you will come across code that exploits this "feature", like in the snippet below.

```
def is_even(num):
   if num % 2 == 0:
      return True
   return False
```

If an integer is divisible by two, then the condition `num % 2 == 0` will be `True` and the function will return a value of `True`. If it is *not* divisible by two, however, then the condition `num % 2 == 0` will be `False`. The interpreter will move on to the next line of the function, which returns a value of `False`. No `else` statement is needed. While the code is shorter, it is arguably not as clear as it would be using an `else` statement. To improve readability, it might be better to refrain from doing this sort of thing.

A function can return more than one value by listing them, separated by commas, after the `return` statement. Here is a function that calculates the midpoint of a line segment using the formula $(x, y)=\left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}\right)$, and returns two values representing the *x*- and *y*-coordinates.

```
def midpoint(x1, y1, x2, y2):
   mid_x = (x1 + x2) / 2
   mid_y = (y1 + y2) / 2
   return mid_x, mid_y
```

The main program would look something like this.

```
x1 = int(input("Enter a value for x1: "))
x2 = int(input("Enter a value for x2: "))
y1 = int(input("Enter a value for y1: "))
y2 = int(input("Enter a value for y2: "))
mid_x, mid_y = midpoint(3, 5)
print("The midpoint is (", mid_x, ", ", mid_y, ")", sep="")
```

Notice that when we call our `midpoint` function, we must specify two variables to hold the two returned values. If we specify only one variable, the values will be stored in a tuple instead (more on this later), which will cause a run-time error with the `print` statement at the end.

At this point we may want to write a function that is responsible for obtaining a coordinate pair, (x, y), from the user instead of asking for all of the coordinates in the main program. We might write something like the following.

```
def midpoint(x1, y1, x2, y2):
   mid_x = (x1 + x2) / 2
   mid_y = (y1 + y2) / 2
   return mid_x, mid_y

def get_coordinate():
   x = int(input("Enter a value for x: "))
   y = int(input("Enter a value for y: "))
   return x, y

x1, y1 = get_coordinate()
x2, y2 = get_coordinate()
mid_x, mid_y = midpoint(3, 5)
print("The midpoint is (", mid_x, ", ", mid_y, ")", sep="")
```

Run the program to verify that it acts the same as the previous version. We will revisit it later with some additional changes.