02 Prepare: Calling Functions

Because most useful computer programs are very large, programmers divide their programs into parts. Dividing a program into parts makes it easier to write, debug, and understand. A programmer can divide a Python program into modules, classes, and functions. In this lesson, you will learn how to call existing functions, and in the next lesson, you will learn how to write your own functions.

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson.

What Is a Function?

A function is a group of statements (computer commands) that together perform one task. Broadly speaking, there are four types of functions in Python which are:

1. Built-in functions
2. Standard library functions
3. Third-party functions
4. User-defined functions

A programmer (you) can save lots of time by using existing functions. In this lesson, you will learn how to use (call) the first two types of functions. In lesson 5, you will learn how to install third-party modules and call third-party functions. In the next lesson, you will learn how to write and call user-defined functions.

Built-in Functions

Python includes many built-in functions such as: `input`, `int`, `float`, `str`, `len`, `range`, `abs`, `round`, `list`, `dict`, `open`, and `print`. These are called built-in functions because you don't have to import any module to use them. They are simply a built-in part of the Python language. You can read about the built-in functions in the Built-in Functions section of the official Python online reference.

How to Call a Function

A programmer uses a function by calling it (also known as invoking it). To call (or invoke) a function means to write code that causes the computer to execute the code that is inside that function. Regardless of the type of function (built-in, standard, third-party, or user-defined), a function is called by writing its name followed by a set of parentheses ( ). During CSE 110 and 111, you often wrote code that called the built-in `input` and `print` functions like this:

```
name = input("Please enter your name: ")
print(f"Hello {name}")
```

```
> python example_1.py
Please enter your name: Miyuki
Hello Miyuki
```

To call a function you must know the following three things:

1. The name of the function
2. The parameters that the function takes
3. What the function does

These three pieces of information are normally available in online documentation. For example, from the online Python documentation about the `input` function, we read this:

> `input(prompt)`
> Write the *prompt* parameter to the terminal window, then read a line of user input from the terminal window, convert the input to a string, and return the input as a string.

From this short description, we know the following:

1. The name of the function is `input`.
2. The function takes one parameter named *prompt*.
3. The function writes the prompt to the terminal and then reads and returns user input from the terminal.

A parameter is a piece of data that a function needs in order to complete its task. In the online documentation for the `input` function, we see that the `input` function has one parameter named *prompt*.

An argument is the value that is passed through a parameter into a function. In other words, parameters are listed in a function's documentation, and arguments are listed in a call to a function.

To write code that calls a function, we normally do the following:

1. Type a new variable name and use the assignment operator (=) to assign a value to the variable.
2. Type the name of the function followed by a set of parentheses.
3. Between the parentheses, type arguments that the computer will pass into the function through its parameters.

For example, the following code calls the built-in `input` function and passes the string `"Please enter your name: "` as the argument for the *prompt* parameter.

```
name = input("Please enter your name: ")
```

When a function has more than one parameter and a programmer writes code to call that function, the programmer nearly always writes the arguments in the same order as the parameters. Consider the description of the built-in `round` function:

> `round(number, ndigits)`
> Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, `round` returns the nearest integer to *number*.

Now consider this Python code that gets a number from a user, rounds that number to two digits after the decimal, and then prints the rounded number.

```
1  n = float(input("Please enter a number: "))
2  r = round(n, 2)
3  print(r)
```

```
> python example_2.py
Please enter a number: 95.716
95.72
```

In the previous example,

- The code on line 1 causes the computer to call the built-in `input` function and then call the built-in `float` function.
- Line 2 causes the computer to call the built-in `round` function and pass two arguments. Notice that the order of the arguments matches the order of the parameters. Specifically, the number to be rounded (*n*) is the first argument, and the number of digits after the decimal point (2) is the second argument.
- Line 3 causes the computer to call the built-in `print` function to print the rounded number.

Optional Arguments

When calling a function or method, some arguments are optional. Again consider the description of the built-in `round` function:

> `round(number, ndigits)`
> Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is `None`, `round` returns the nearest integer to *number*.

From the description, we read that the second argument is optional. If the programmer doesn't pass a second argument, the value in the *number* parameter will be rounded to an integer.

Named Arguments

For some optional arguments, we must pass a named argument, which is an argument that is preceded by the name of its matching parameter. For example, here is an excerpt from the documentation for the `print` function:

> `print(*objects, sep=" ", end="\n", file=sys.stdout, flush=False)`
> Print objects to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file* and *flush*, if present, must be given as named arguments.

Notice from the excerpt that the `print` function can take many objects that will be printed. Optionally, it can take parameters named *sep*, *end*, *file*, and *flush* that must be named when they are used. For example, this code calls the `print` function to print three words all separated by a vertical bar (|). Notice the named arguments *sep* and *flush*.

```
x = "sun"
y = "moon"
z = "stars"
print(x, y, z, sep="|", flush=True)
```

```
> python example_3.py
sun|moon|stars
```

What Is a Module?

A Python module is a collection of related functions. The Python standard library includes many modules which have more functions, such as the `math` module—which includes the `floor`, `ceil`, and `sqrt` functions and the `random` module—which includes the `randint`, `choice`, and `shuffle` functions. Consider the description of the `sqrt` function that is in the standard `math` module:

> `math.sqrt(x)`
> Return the square root of *x*.

From this short description, we know the following:

1. The name of the containing module is `math`.
2. The name of the function is `sqrt`.
3. The function takes one parameter named *x*.
4. The function computes and returns the square root of the number that is in *x*.

To use any code that is in a module, you must import the module into your program and precede the function name with the module name. For example, if you wish to call the `math.sqrt` function, you must first import the `math` module and then type `math.` in front of `sqrt` like this:

```
import math

r = math.sqrt(71)
print(r)
```

```
> python example_4.py
8.426149773176359
```

In the above example, 71 is the argument that will be passed through the parameter *x* into the `math.sqrt` function. The `math.sqrt` function will compute the square root of 71 and return the computed value that will then be stored in the variable *r*. You can read more about the standard modules in the official documentation for the Python Standard Library.

What Is a Method?

Python is an object-oriented language and includes many classes and objects. A method is a function that belongs to a class or object. Even though classes and objects are not part of this course (CSE 111), calling a method in Python is so common and so easy that you should know how to do it. A method is a kind of function, so calling a method is similar to calling a function. The difference is that to call a method we must type the name of the object and a period (.) in front of the method name.

Consider the following example code that gets a string of text from a user and prints the number of characters in the string and prints the string in all upper case characters.

```
 1  # Get a string of text from the user.
 2  text1 = input("Enter a motivational quote: ")
 3
 4  # Call the built-in len function to get
 5  # the number of characters in the text.
 6  length = len(text1)
 7
 8  # Call the string upper method to convert
 9  # the quote to upper case characters.
10  text2 = text1.upper()
11
12  # Call the built-in print function to print
13  # the length of the text and the text in all
14  # upper case for the user to see.
15  print(length, text2)
```

```
> python example_5.py
Enter a motivational quote: Rise, take up thy bed, and walk.
32 RISE, TAKE UP THY BED, AND WALK.
```

Notice the code on line 6 calls the built-in `len` function and the code on line 10 calls the string `upper` method. Compare the function call in line 6 to the method call in line 10. To call the `len` function, we type the name of the function followed by a list of arguments inside parentheses. To call the `upper` method, we type the name of the object (`text1`) and a period, then the method name (`upper`), and then a list of arguments inside parentheses.

A method can receive arguments just like a function can. However, in the previous example at line 10, there are no arguments passed to the `upper` method, so the parentheses are empty. In order for the computer to call the `upper` method, a programmer must type the empty parentheses. In other words, if you write a line of code to call the `upper` method but don't type the empty parentheses, like this:

```
text2 = text1.upper
```

the computer will not call the `upper` method. Instead the computer will assign a reference to the `upper` method to the *text2* variable. You don't want the computer to do this because assigning a function reference won't make sense to you until you study functional programming.

Storing a Returned Value

While it's usually a good practice, you don't *have* to store the value that is returned from a function in a variable. Sometimes you will see it used directly as shown in example 7 at lines 9, 13, and 15.

```
 1  # Example 7
 2  import math
 3
 4  # Get a number from the user.
 5  number = float(input("Enter a number: "))
 6
 7  # Call the math.sqrt function and
 8  # immediately print its return value.
 9  print( math.sqrt(number) )
10
11  # Call the math.sqrt function again and
12  # use its return value in an if statement.
13  if math.sqrt(number) < 100:
14      print(f"The square root is less than 100.")
15  elif math.sqrt(number) > 100:
16      print(f"The square root is more than 100.")
17  else:
18      print(f"The square root is exactly 100.")
```

```
> python example_7.py
Enter a number: 675
25.98076211353316
The square root is less than 100.
```

Notice in example 7, there are three statements that call the `math.sqrt` function, one at line 9 to print the square root, another at line 13 to check if the square root is less than 100, and yet another at line 15 to check if the square root is greater than 100. Every time the computer calls a function, the computer will execute the code that is inside that function. In example 7, because the arguments are the same at lines 9, 13 and 15, the returned result will be the same in all three cases. So it would be faster to save the result in a variable and reuse the variable instead, as shown in example 8 at lines 9, 11, 13, and 15.

```
 1  # Example 8
 2  import math
 3
 4  # Get a number from the user.
 5  number = float(input("Enter a number: "))
 6
 7  # Call the math.sqrt function and store its
 8  # return value in a variable to use later.
 9  root = math.sqrt(number)
10
11  print(f"The square root is {root:.2f}")
12
13  if root < 100:
14      print(f"The square root is less than 100.")
15  elif root > 100:
16      print(f"The square root is more than 100.")
17  else:
18      print(f"The square root is exactly 100.")
```

```
> python example_8.py
Enter a number: 675
The square root is 25.98
The square root is less than 100.
```

Tutorial

If you are uncertain about any of the concepts in the previous section, you could reread the section. Also, you could read about the same concepts in the Python functions tutorial at w3schools.

Summary

A function is a group of statements that together perform one task. The computer will not execute the code in a function unless you write code that calls the function. In this lesson, you learned how to call built-in functions, functions that are in a module, and functions (methods) that belong to an object.

1. To call a built-in function, write code that follows this template:

   ```
   variable_name = function_name(arg1, arg2, … argN)
   ```

2. To call a function from a module, import the module and write code that follows this template:

   ```
   import module_name

   variable_name = module_name.function_name(arg1, arg2, … argN)
   ```

3. To call a method, write code that follows this template:

   ```
   variable_name = object_name.method_name(arg1, arg2, … argN)
   ```