Week 6 Agenda

Week 6 Functions and Modules

Chapter 8 in Python Crash Course: A Hands-on, Project-Based Introduction to Programming

Functions

Functions, are named blocks of code that are designed to do one specific job. When you want to perform a particular task that you've defined in a function, you *call* the name of the function responsible for it. If you need to perform that task multiple times throughout your program, you don't need to type all the code for the same task again and again; you just call the function dedicated to handling that task, and the call tells Python to run the code inside the function. You'll find that using functions makes your programs easier to write, read, test, and fix.

Benefits of Using Functions

- Simpler code
  - Small modules easier to read than one large one
- Code reuse
  - Can call modules many times
- Better testing
  - Test separate and isolate then fix errors
- Faster development
  - Reuse common tasks
- Easier facilitation of teamwork
  - Share the workload

Function Syntax

A *function* is a series of related steps (statements) that make up a larger task, which is often called from multiple places in a program.

Here is the generic form of a Python function:

```
def <functionName>(<optionalParameters>): # notice the parentheses and the ending colon
    <indented statement(s)> # the 'body' of the function
```

Defining Functions

●The word def is short for definition. We use this keyword since we're defining a function. Next, you supply a name for the function. A set of parentheses follows the name. The line ends with a colon (:), which is very important.

●All the statements that make up the function, called the *body* of the function, are indented from the def statement. Python relies on indenting to show a grouping of lines. The convention is to indent four spaces. (You can change this in the IDLE Preferences, but it defaults to 4, and four spaces is a broadly accepted convention.) Python's use of indenting only (with no braces) is unique and helps make Python code much more readable than most other languages.

Function Example

In this example we define a function called hello_function. This function will declare a variable called userName which is set to the input from the user. The function will then print a message to the screen.

```
def hello_function():
    userName = input('What is your name?')
    print('Hello ' + name + ', welcome to my function!')
```

Calling a Function

In order to call a function we simply reference the function name using the following syntax:

```
<functionName>(<argument1>, <argument2>, ...)
```

Calling Function Example:

```
def hello_function():
    userName = input('What is your name?')
    print('Hello ' + name + ', welcome to my function!')

hello_function() #This line calls our function defined above
```

Function Parameters

It is far more useful to allow the passing of data to functions. The data we pass to a function is referred to as a parameter.

A *parameter* is a variable (defined in the def statement of a function) that is given a value when a function starts. (It is also known as a *parameter variable.)*

When you call a function, Python must match each argument in the function call with a parameter in the function definition. The simplest way to do this is based on the order of the arguments provided. Values matched up this way are called *positional arguments*.

Function Parameter Syntax

```
def blend(<parameter 1>, <parameter 2>…):
    <indented block of code>
```

Function Parameter Example

```
def hello_function(userName):
    print('Hello ' + userName + ', welcome to my function!')

userName = input('What is your name?')
hello_function(userName) #Call our function and pass the userName as a parameter
```

Returning Data from a Function

- It is common for a function to perform an operation of some kind and return the results to the program.
- In Python, when a function wants to give a result to a caller, it uses a return statement and specifies the value to hand back. The generic form looks like this:

```
    return <returnValue>
```

Example :

```
def multiplyNumbers(x):
    return x*x

print(multiplyNumbers(3))
print(multiplyNumbers(3) + multiplyNumbers(4))
```

Returning Multiple Values

Python has a further extension of the return statement. In most other programming languages, the return statement can only return either no values or a single value. In Python, just as you can pass as many values as you want into a function, you can also return any number of values:

```
return <value1>, <value2>, <value3>, ...
```

For example, you could create a function that returns three values, like this:

```
def myFunction(parameter1, parameter2):
    #
    # Body of the function, calculates
    # values for answer1, answer2, and answer3
    return answer1, answer2, answer3 # hand back three answers to the caller
```

Then you would call myFunction with code like this:

```
variable1, variable2, variable3 = myFunction(argument1, argument2)
```

Where to Define Functions

Functions in Python are typically defined at the beginning of the program prior to the location where they are called. If Python functions are not defined before they are called, Python will throw an error.

Global Variables

If you define *global variables* (variables defined outside of any function definition), they are visible inside all of your functions. They have *global scope*. It is good programming practice to avoid defining global variables and instead to put your variables inside functions and explicitly pass them as parameters where needed. One common exception is constants: A *constant* is a name that you give a fixed data value to, by assigning a value to the name only in a single assignment statement. You can then use the name of the fixed data value in expressions later.

Global Constant Example

```
'''Illustrate a global constant being used inside functions.'''
PI = 3.14159265358979 # global constant -- only place the value of PI is set

def circleArea(radius):
    return PI*radius*radius # use value of global constant PI

def circleCircumference(radius):
    return 2*PI*radius # use value of global constant PI

def main():
    print('circle area with radius 5:', circleArea(5))
    print('circumference with radius 5:', circleCircumference(5))

main()
```

Modules

Python makes it possible to store your functions within a file separate from the main python program. These are referred to as modules. Creating modules allow you to use the same functionality in multiple programs. Modules can be imported into a program by using the `import` statement. Let's break up the program on the previous slide into two files. The first is circleCalc.py. This is our module and will contain the two functions to calculate the Area and the Circumference of a circle.

```
#circleCalc.py

PI = 3.14159265358979 # global constant -- only place the value of PI is set

def circleArea(radius):
    return PI*radius*radius # use value of global constant PI

def circleCircumference(radius):
    return 2*PI*radius # use value of global constant PI
```

Using Modules

Now we can use our module within our program. In our new program circle.py, we import our circleCalc Module. We define a main function which has our programs logic, and then we call main. Notice that we call our functions (circleArea and circleCircumferences) by using dot notation with the module name and a period prior to the function name.

```
import circleCalc

def main():
    print('circle area with radius 5:', circleCalc.circleArea(5))
    print('circumference with radius 5:', circleCalc.circleCircumference(5))

main()
```

References

The Basics of Compiled Languages, Interpreted Languages, and Just-in-Time Compiliers, https://www.upwork.com/hiring/development/the-basics-of-compiled-languages-interpreted-languages-and-just-in-time-compilers/

The End

You may close this Window and return to the course.