**Chapter 5 : Functions**

You have used a number of Python functions (print, input, etc) so far in this course. These were all **built-in functions**, created by Guido van Rossum when Python was invented. Chapter 5 is all about writing your own **custom functions** to perform coding tasks.

**What are custom functions?**

* Custom functions are **named blocks of code** designed to do **one specific job**.
* Functions allow you to **write code once** that can then be run whenever you need to accomplish the **same task**.
* Functions can **take in the information they need**, and **return** (give back) the information they generate.
* Using functions makes your programs **easier to write, read, test, and fix**.
* Python professionals write custom functions every day, and they **save them** so they are available for future applications with similar requirements.

**Defining a Function**

* A function definition begins with a **single line** of code.
* The line consist of the keyword **def** followed by a **name** for the function.
* The line finishes with a set of **parentheses** and **a colon**.
* Beneath this first line is the **body** of the function.
* The Python statements in the body are **indented** by one level (a tab).
* The body statements accomplish the job of the function.

def say\_hello():

print(‘Hello’)

**Calling a Function**

* In coding lingo, executing or running a function is termed **calling** the function.
* You call a function by its name, without the colon.

say\_hello() # calls the say\_hello function

**Passing data to a Function**

* Some functions need one or more data items called **parameters**.
* Parameters are listed in the parentheses separated **by commas** when a function is defined.
* The data that is **passed into** a function **when it is called** are known as **arguments**.
* Arguments are listed inside the parentheses separated **by commas** when it is called.

**Examples**

def say\_hello\_name(name): # name is a **parameter**

print(f‘Hello, {name}’)

say\_hello\_name(‘Benny’) # literal string Benny is an **argument**

user = ‘Guido’

say\_hello\_name(user) # variable user is an **argument**

def product(num1, num2): # two **parameters**, num1 and num2

prod = num1 \* num2

print(f’The product is {prod}’)

n1 = 10

n2 = 5

product(n1, n2) # n1 and n2 are the **arguments**

**Two Types of Functions**

1. **Value-Returning Type** [**Very important and often not understood. Read closely.**]

* A function can be coded to “return a value” with a **return statement**.
* The return statement **stops, or ends**, the function.

def total(var1, var2): # two **parameters**, var1 and var2

return var1 + var2 # returns the total, **stops the function**

* When a function returns a value, the calling line must **provide a means to store it**.
* Often, the returned value is **assigned to a variable**.

num1 = 14

num2 = 16

result = total(num1, num2) # the returned value is **assigned to result**

# the returned value can then be used in subsequent code

print(f’{num1} plus {num2} is {result}’)

* The returned value can also be **embedded** as an **argument to another function**.

print(f’{num1} plus {num2} is {**total(num1,num2)**}’)

* This latter method is **okay only if** **you don’t need the returned value** **later in the program**.
* Functions can return **ints, floats, strings, booleans** and other types not yet studied.

**NOTE**: You have already been using some builtin functions that return a value.

name = input(‘Enter you name ‘)  
**input() is a value-returning function**. Whatever you enter is **returned as a string** into the **variable name**.

num = int(input(‘Enter your age ‘))

The **int() function** converts a string to an int **and returns it** into the **variable num**.

1. **Void Type**

* These are functions **that** **don’t return a value**.
* Void functions can be coded for various tasks, such as generating printed output.
* The **product** function shown above on this page is an example of a void function.
* A void function can contain a **naked return statement** to immediately end the function.

def prt\_total(num1, num2): # a void function

total = num1 \* num2

print(f’{num1} plus {num2} is {total}’)

* Call void functions like Python statements, since there is **no returned value to store**.

prt\_total(num1, num2) # calling the void prt\_total function

**Variable Scope, Globals**

* The **scope** of a variable is the part of a program in which the **variable is visible**.
* Declaring a variable **inside** a function **limits its scope to that function**.
* A variable declared in this way is termed a **local variable** (local to that function).
* A variable declared in a file but **outside** of any function is a **global variable.**
* **Avoid global variables**. They can cause problems. **Global constants are okay**.

**Modules**

* A module is a **separate file** that **contains a number of Python function definitions**.
* The functions in a module follow a **certain theme**.
* When you installed Python, you got the most commonly needed modules (about **200**) but there are thousands more that you can install.
* The **random module**  and  **math module** are introduced in this chapter. Both of these are used frequently in this course.
* Gain access to the functions in a module in your programs with the **import** statement.
* To use a function from an imported moduke, precede it with the **module name and a period**.

import random

# next line **returns** a random integer **10-20, inclusive**, to num

num = random.randint(10,20)

import math

num = 10

# next line **returns** the square root of 10 to variable root

root = math.sqrt(num)

**Custom Modules**

* Make your own custom modules by storing your **function definitions** in a file.
* The file should be **named with a .py filename extension**.
* However, when you import your own module, **drop the .py** extension.
* Be sure to put your custom module file in the **same folder as your program**.

A Python file **can execute in two ways**,each assigning a different value to a special variable called **\_\_name\_\_** (two underscores before and after main).

1. As the main program or script when you run the file in IDLE.
   * This assigns **“\_\_main\_\_”** to the \_\_name\_\_ variable.
2. As a library file when imported with the **import** statement.
   * This assigns the **module filename** (minus .py) to the \_\_name\_\_ variable.
   * Example code is common in a library file that **will run** when the file is executed as a main program. This example code **won’t run** if the file has been imported.

**NOTE**: Due to these two methods of executing a file, example programs in the rest of the textbook will now finish with an **if statement** that tests the value of \_\_name\_\_ to check if the file is running as a program or being imported. The code will look like this:

**if \_\_name\_\_ == ‘\_\_main\_\_’:**

**main()**

Note \_\_name\_\_ and \_\_main\_\_ are called **dunders** because of the double underlines.

|  |  |
| --- | --- |
| FILE CODE | OUTPUT WHEN EXECUTED |
|  |  |
|  |  |

**Other Modules**

Python programmers are generous people. They share their code. There are **more than 200,000** other modules (packages) that you can install with the **pip utility**. You can browse the modules at <https://pypi.org/> Learn about pip here <https://pip.pypa.io/en/stable/>

A module named **pygame** can be used to create 2D games. This module was not included when you installed Python, but you can get it for Windows in a command window (Terminal window for Mac users) with this command:

**python –m pip install pygame**

Here is a **online pygame book** with several games, including the source code:

<https://inventwithpython.com/pygame/>