Programming with Python

By Randal Root

Module 06

In this module, you learn about **creating scripts using Functions**. You **also** learn a little about creating **simple Classes, use the PyCharm debugger, and** **GitHub web pages**.

[Functions 2](#_Toc23763348)

[Parameters 2](#_Toc23763349)

[LAB 6-1: Working with functions 3](#_Toc23763350)

[Using Variables as Arguments 4](#_Toc23763351)

[Return values 5](#_Toc23763352)

[LAB 6-2: Returning Tuples 7](#_Toc23763353)

[Working with Arguments 8](#_Toc23763354)

[Positional vs. Named arguments 8](#_Toc23763355)

[Default Parameter Values 9](#_Toc23763356)

[Overloaded Functions 9](#_Toc23763357)

[The None Keyword 11](#_Toc23763358)

[Returning Data by Reference 12](#_Toc23763359)

[Global vs. Local Variables 14](#_Toc23763360)

[Function Document Headers (Doc Strings) 17](#_Toc23763361)

[Classes and Functions 18](#_Toc23763362)

[LAB 6-3: Creating a Class of Functions 19](#_Toc23763363)

[Using the PyCharm Debugger 20](#_Toc23763364)

[Debug mode 20](#_Toc23763365)

[Breakpoints 21](#_Toc23763366)

[Creating a GitHub Webpage 23](#_Toc23763367)

[Configuring the WebSite 24](#_Toc23763368)

[Modifying the GitHub page 26](#_Toc23763369)

[Summary 27](#_Toc23763370)

# Functions

Functions are a way of **grouping one or more statements**. In Python, you must **define** a function **before** **you** can use code to **call the function**. **Calling** the function **executes the statements in the function** (Listing 1).

# ------------------------------------------------- #  
# Title: Listing 1  
# Description: Making a calling a function  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
# Define the function  
**def ProcessSomething**():  
 print(**"I'm"**) # first statement  
 print(**"processing data"**) # second statement  
  
# Call the function  
**ProcessSomething**()

Listing 1

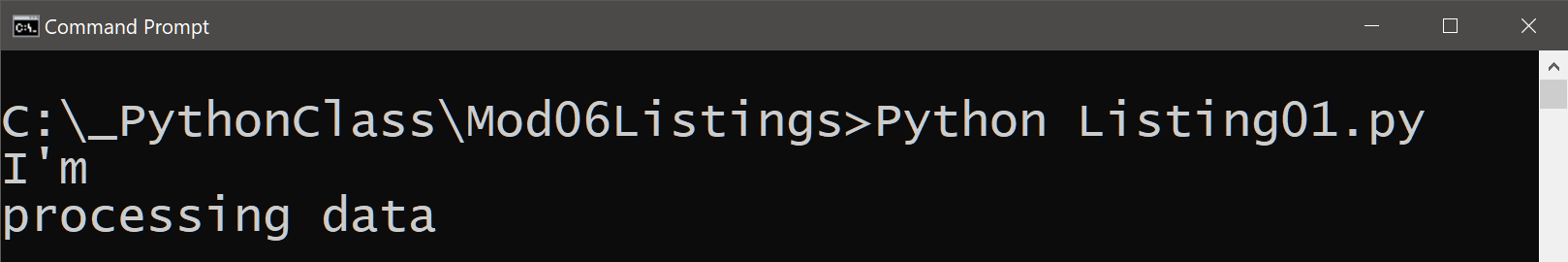


Figure 1. The results of Listing 1

## Parameters

Optionally, functions can have **parameters**. These **allow you to pass values into the function for processing**. **Values passed into parameters are officially called “arguments,”** but it's common for people to call them parameters as too (Listing 2).

# ------------------------------------------------- #  
# Title: Listing 2  
# Description: Calling a function with parameters  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
# Define the function  
**def** ProcessSomething(**parm**Message):  
 print(**"The parameter was: "** + parmMessage)  
  
# Call the function  
ProcessSomething(**"arg** ABC**"**)

Listing 2

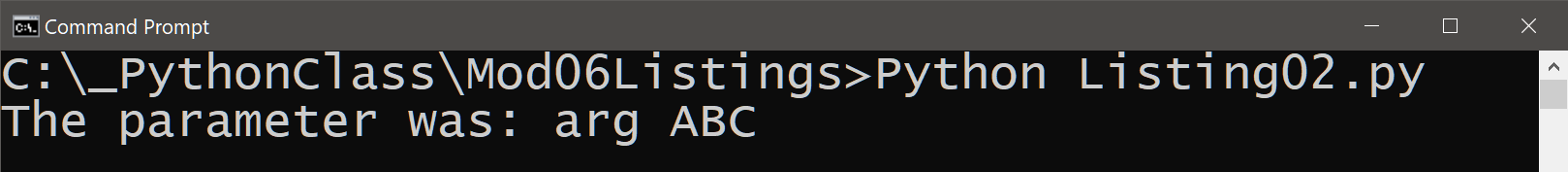


Figure 2. The results of Listing 2

There is **no practical limit on how many parameters** you can include. For example, you can easily pass two arguments to a function that calculates a sum (Listing 3).

# ------------------------------------------------- #  
# Title: Listing 3  
# Description: Calling a function with multiple parameters  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
# Define the function  
**def** AddValues(value1, value2):  
 fltAnswer = value1 + value2  
 print(**"The Sum of the values is: "** + str(fltAnswer))  
  
# Call the function  
AddValues(10, 5)

Listing 3

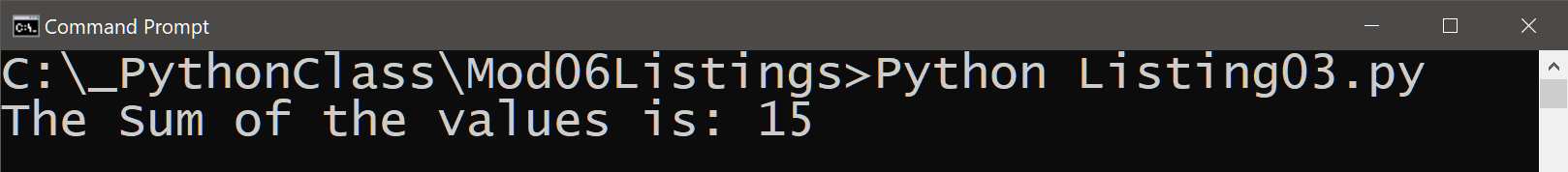


Figure 3. The results of Listing 3

**Note:** It is standard in the industry to **parameter names without a prefix**. As such, I leave them off in my function's parameters as well. I even use the common Python **"snake casing" for some examples**, just so you get used to seeing it.

# LAB 6-1: Working with functions

In this lab, you create a **function that prints the Sum, Difference, Product, and Quotient** of two numbers. Make your results look like Figure 4.

1. **Create** a new **script called Lab6-1** that uses code in Listing 3.

2. **Add** statements to **calculate** **and print** the **sum, difference, product, and quotient**.

3. **Test** the script and **write** down how the code works.

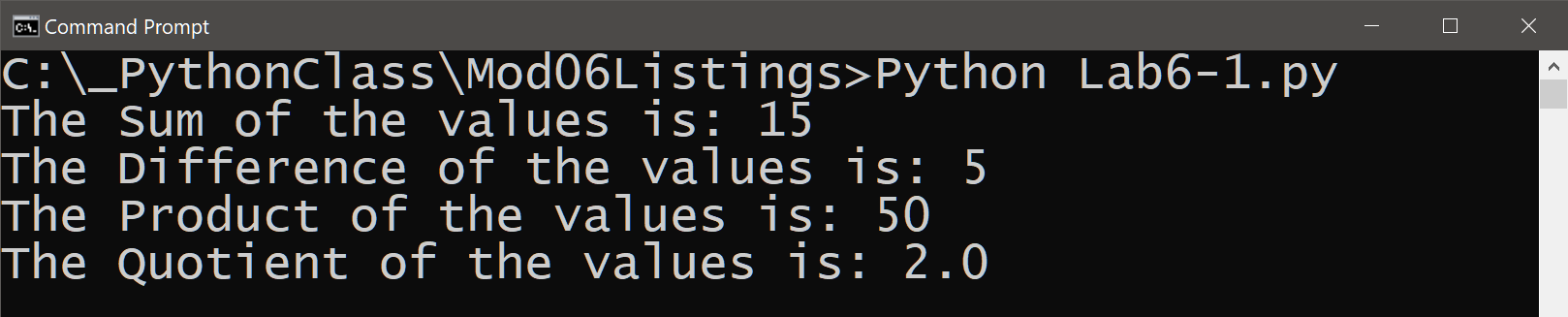


Figure 4. The results of Lab 6-1

## Using Variables as Arguments

In Listing 4, I use **variables as arguments, which is a common pattern used in programming.** Using argument variables is **useful** when you want access to these values **multiple times in a script.**

#-------------------------------------------------#  
# Title: Listing 04  
# Description: Using variables as arguments  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
#-------------------------------------------------#

# -- data code -- #  
fltV1 = **None** # first argument  
fltV2 = **None** # second argument  
  
# -- processing code -- #  
**def** AddValues(value1, value2):  
 fltAnswer = value1 + value2  
 print(fltAnswer);  
  
# -- presentation (I/0) code -- #  
fltV1 = float(input(**"Enter value 1: "**))  
fltV2 = float(input(**"Enter value 2: "**))  
print(**"The Sum of %.2f and %.2f"** % (fltV1, fltV2))  
print(**"is: "**, end=**''**)  
AddValues(fltV1, fltV2)

Listing 5

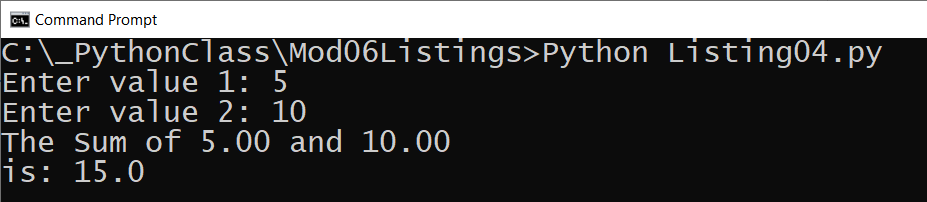


Figure 6. The results of Listing 5 using values 5 and 10

## Return values

Functions can **return one or more values.** You **capture returning values** **of a function in variables**.

**def MyFunction**():  
 **return 'data'**v1 = **MyFunction**()

**"Return values" make a function act as an expression**. **Evaluating a function as an expression** means that you **use the results of a function immediately without placing the result in a variable**.

print(**MyFunction**())

**Capturing the results in a variable** allows you to **use** the variable of **results multiple times without having to call the function again, but using it as an expression does not (Listing 5)**.

# ------------------------------------------------- #  
# Title: Listing 05  
# Description: Using a function's return value  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
# -- processing code -- #  
**def AddValues**(value1, value2):  
 fltAnswer = value1 + value2  
 **return** fltAnswer  
  
# Call the function and capture the results  
**fltResults** = **AddValues**(10, 5)  
print(**"The Sum of the values is: "** + str(**fltResults**))  
  
# Or call the function and uses as an expression  
print(**"The Sum of the values is: "** + str(**AddValues**(10, 5)))

Listing 5

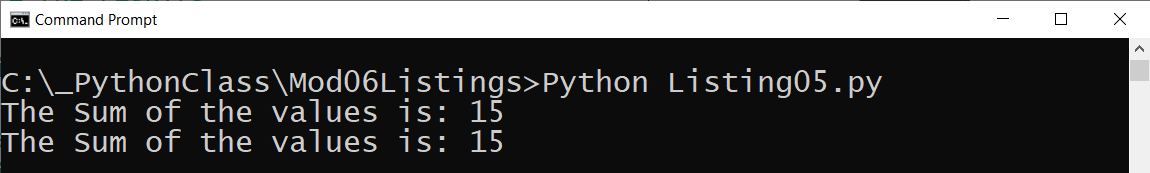


Figure 7. The results of Listing 5 using values 5 and 10

**Important:** Notice that there is **no presentation code in the function when using the return option!** Functions with return values make code **easier to divide into three layers of concern**; data, processing, and presentation.

### Returning Multiple Values

**Return values can be** a single item of data or **multiple items**. **If you return multiple values, you need to bundle them into a collection** and return that collection. In **Python,** simplify this process using the **tuple packing and unpacking feature (Listing 6)**.

#-------------------------------------------------#  
# Title: Listing 06  
# Description: Returning multiple values as a tuple  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
#-------------------------------------------------#  
  
# -- data code -- #  
fltV1 = **None** # first argument  
fltV2 = **None** # second argument  
fltR1 = **None** # first result of processing  
fltR2 = **None** # second result of processing  
fltR3 = **None** # third result of processing  
  
# -- processing code -- #  
**def** AddValues(value1, value2):  
 fltAnswer = value1 + value2  
 **return** value1, value2, fltAnswer # pack tuple  
  
# -- presentation (I/0) code -- #  
fltV1 = float(input(**"Enter value 1: "**))  
fltV2 = float(input(**"Enter value 2: "**))  
fltR1, fltR2, fltR3 = AddValues(fltV1, fltV2) # unpack tuple  
print("The Sum of %.2f and %.2f is %.2f" % (fltR1, fltR2, fltR3))

Listing 6

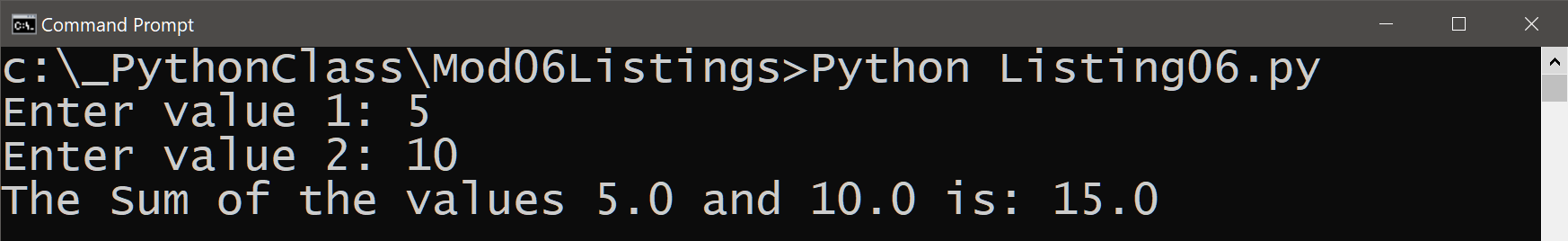


Figure 7. The results of Listing 6 using values 5 and 10

The **packing and unpacking feature only applies when using Python tuples,** but **other languages** do not let have this option. Instead, you would use code similar to Listing 7, which shows an example of **using a List object instead** of a tuplewhile still producing the **same results as those shown in Figure 7**.

#-------------------------------------------------#  
# Title: Listing 07  
# Description: Returning multiple values as a list  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
#-------------------------------------------------#  
  
# -- data code -- #  
fltV1 = **None** # first argument  
fltV2 = **None** # second argument  
lstResults = **None** # list of results for processing  
  
# -- processing code -- #  
**def** AddValues(value1, value2):  
 fltAnswer = value1 + value2  
 **return** [value1, value2, fltAnswer] # create a list  
  
# -- presentation (I/0) code -- #  
fltV1 = float(input(**"Enter value 1: "**))  
fltV2 = float(input(**"Enter value 2: "**))  
lstResults = AddValues(fltV1, fltV2) # capture the list  
print("The Sum of %.2f and %.2f is %.2f" %

(lstResults[0], lstResults[1], lstResults[2]))

Listing 7

Still, when programming with **Python, you often use tuples to return multiple values**.

# LAB 6-2: Returning Tuples

In this lab, you create a **function that returns the Sum, Difference, Product, and Quotient** of two numbers as a tuple. Make your results look like Figure 8.

1. **Create a new script called Lab6-2.**

2. **Add** code to **capture values** from a user and **return a tuple** of results that **include the sum, difference, product, and quotient of the two values entered by the user**.

3. **Print** the results to the user, as shown in Figure 8.

3. **Test** the script and **write** down how the code works.

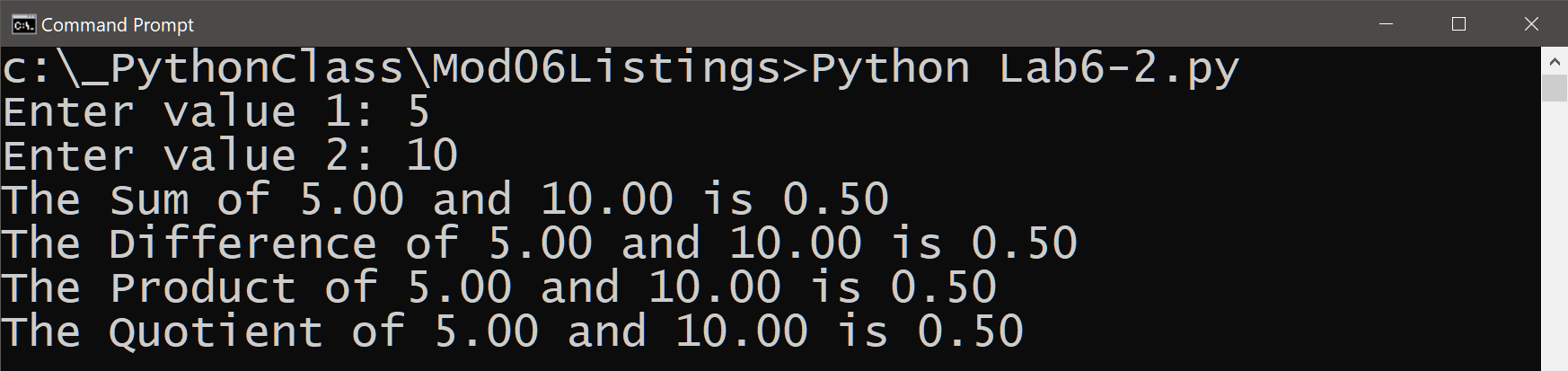


Figure 8. The results of Lab 6-2 using values 5 and 10

# Working with Arguments

Arguments are used to make a function perform different actions or return different results. **Most languages support many options for arguments that make them even more useful,** and Python is no exception.

## Positional vs. Named arguments

When you call a function, you **can include the name of the parameter and fill it explicitly with your arguments (Listing 8)**.

#-------------------------------------------------#  
# Title: Listing 08  
# Description: Using explict parameter names  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
#-------------------------------------------------#  
  
# -- data code -- #  
fltV1 = **None** # first argument  
fltV2 = **None** # second argument  
fltR1 = **None** # first result of processing  
fltR2 = **None** # second result of processing  
fltR3 = **None** # third result of processing  
  
# -- processing code -- #  
**def** AddValues(value1, value2):  
 fltAnswer = value1 + value2  
 **return** value1, value2, fltAnswer

# -- presentation (I/0) code -- #  
fltV1 = float(input(**"Enter value 1: "**))  
fltV2 = float(input(**"Enter value 2: "**))  
fltR1, fltR2, fltR3 = AddValues(value1 = fltV1, value2 = fltV2)  
print(**"The Sum of %.2f and %.2f is %.2f"** % (fltR1, fltR2, fltR3))

Listing 8

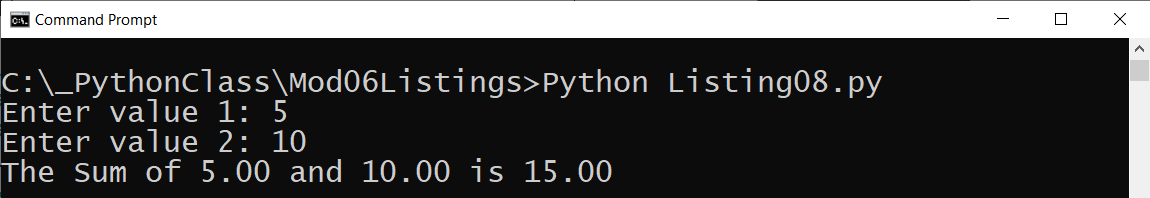


Figure 9. The results of Listing 8 using values 5 and 10

## Default Parameter Values

You can set default values for a parameter. When you do, **not supplying an argument for the parameter forces the function to use the parameter's default value (Listing 9)**.

#-------------------------------------------------#  
# Title: Listing 09  
# Description: Using default parameter values  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
#-------------------------------------------------#  
  
# -- data code -- #  
fltV1 = **None** # first argument  
fltV2 = **None** # second argument  
fltR1 = **None** # first result of processing  
fltR2 = **None** # second result of processing  
fltR3 = **None** # third result of processing  
  
# -- processing code -- #  
**def** AddValues(value1 = 0, value2 = 0):  
 fltAnswer = value1 + value2  
 **return** value1, value2, fltAnswer  
  
# -- presentation (I/0) code -- #  
fltV1 = float(input(**"Enter value 1: "**))  
fltV2 = float(input(**"Enter value 2: "**))  
fltR1, fltR2, fltR3 = AddValues(value2 = fltV2)  
print(**"The Sum of %.2f and %.2f is %.2f"** % (fltR1, fltR2, fltR3))

Listing 9

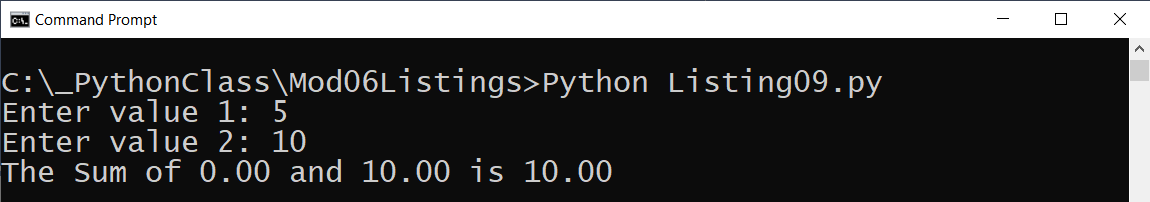


Figure 10. The results of Listing 9 using values 5 and 10

***Important****: Notice how even though I asked the user to enter "value 1," it was never used, and as such,* ***the script does not display what the user expects!***

## Overloaded Functions

**Many modern languages** allow you to **create multiple versions of a function**. Each version uses a **different number of parameters or parameters with different data types**.

MyFunction(): *# No parameters* **return 'data'**MyFunction(p1:int): *# One int parameter* **return 'data'**MyFunction(p1:float): *# One float parameter* **return 'data'**MyFunction(p1:float, p2:float): *# two float parameters* **return 'data'**print(MyFunction())  
print(MyFunction(5))  
print(MyFunction(5.0))  
print(MyFunction(5.0, 10.0))

**Python cannot** distinguish between versions based on data types due to its automatic data typing feature, nor does it allow you to have two functions with the same name. **Instead, Python uses default values to accomplish something similar (Listing 10)**.

#-------------------------------------------------#  
# Title: Listing 10  
# Description: Python's version of function overloading  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
#-------------------------------------------------#  
  
# -- processing code -- #  
**def** CalcValues(value1 = 0, value2 = 0, operation = **'+'**):  
 **if** operation.lower() == **'+'**: fltAnswer = value1 + value2  
 **elif** operation.lower() == **'-'**: fltAnswer = abs(value1 - value2)  
 **elif** operation.lower() == **'\*'**: fltAnswer = value1 \* value2  
 **elif** operation.lower() == **'/'**: fltAnswer = value1 / value2  
 **else**: fltAnswer = **"Error"  
 return** value1, operation, value2, fltAnswer  
  
# -- presentation (I/0) code -- #  
print(CalcValues())  
print(CalcValues(5,10))  
print(CalcValues(5,10,**'\*'**))

Listing 10

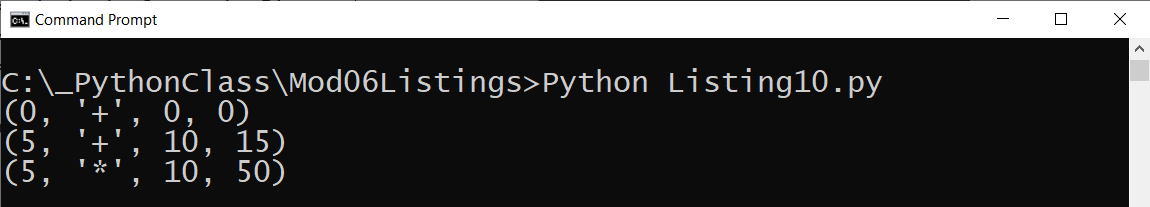


Figure 11. The results of Listing 10

## The None Keyword

In Python, **"None" is a special data type** whose only value is "None." None is frequently used to **indicate the absence** **of** a parameter **value**, as shown in Listing 11.

You often see **None used as a default argument**. When you do **check** for the default by **using** the **"is" operator** instead of the "==" operator.

#-------------------------------------------------#  
# Title: Listing 11  
# Description: Python's version of function overloading  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
#-------------------------------------------------#  
  
# -- processing code -- #  
**def** CalcValues(value1 = **None**, value2 = **None**, operation = **None**):  
 **if** value1 is **None or** value2 is **None or** operation is **None**:  
 fltAnswer = **"Error: Missing Arguments"  
 elif** operation.lower() == **'+'**: fltAnswer = value1 + value2  
 **elif** operation.lower() == **'-'**: fltAnswer = abs(value1 - value2)  
 **elif** operation.lower() == **'\*'**: fltAnswer = value1 \* value2  
 **elif** operation.lower() == **'/'**: fltAnswer = value1 / value2  
 **else**: fltAnswer = **"Error"  
 return** value1, operation, value2, fltAnswer  
  
# -- presentation (I/0) code -- #  
print(CalcValues())  
print(CalcValues(5, 10))  
print(CalcValues(5, 10, **'\*'**))

Listing 11

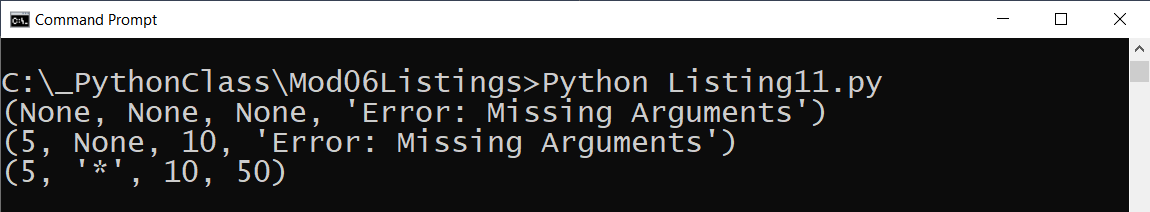


Figure 12. The results of Listing 11

***Important****: Both Listings 10 and 11 place* ***error messages a return value, and this is not considered a best practice!*** *We look at how to improve error handling using the* ***Try-Except in the next module!***

# Returning Data by Reference

**In Python, some variables act like "value types" and others as "reference types." The code in listing 12 was discussed in module 02.** Review it and **see if it makes more sense** to you at this point in the course.

# Simple data

x = 1

y = x

x = 3

print(y) # Print 1, as if only the **value** was passed!

# Complex data

x = [1, 2] # This is a Python List

y = x

x[0] = 3

print(**y**) # Prints [3,2], as expected of a **reference**!

# Simple data

x = "Bob"

y = x

x = "Robert"

print(y) # Print "Bob", as if only the **value** was passed!

# Complex data

x =["Bob", "Sue"]

y = x

x[0] = "Robert"

print(**y**) # Prints ['Robert', Sue], as expected of a **reference**!

Listing 12

**In many modern languages, you can choose to pass arguments into a function as a reference to an address in memory or a value**. You would **indicate your choice using keywords** next to the parameter's name.

MyFunction(p1:float by **val**, p2:float by **ref**):   
 **return 'data'**

**Python** **automates** this functionality **based on data types**. Simple arguments like **Strings, Integers, and Floats** pass to a function as a **value**. Complex arguments like **Lists** and **Dictionaries** pass to a function as a **reference**. Oddly, **Tuples**, although complex, also pass to a function as a **value**. Listing 12 shows an example using a List and Dictionary object as a reference argument.

#-------------------------------------------------#  
# Title: Listing 13  
# Description: Python's version of by val and by ref  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
#-------------------------------------------------#  
  
# -- data code -- #  
v1 = 5  
v2 = 10  
lstData = [0, 0, 0, 0]  
dicData = {**"sum"**: 0, **"diff"**: 0, **"prod"**: 0, **"quot"**: 0}  
  
# -- processing code -- #  
**def** CalcListValues(value1, value2, all\_answers):  
 all\_answers[0] = value1 + value2  
 all\_answers[1] = value1 - value2  
 all\_answers[2] = value1 \* value2  
 all\_answers[3] = value1 / value2  
  
**def** CalcDictionaryValues(value1, value2, all\_answers):  
 all\_answers[**"sum"**] = value1 + value2  
 all\_answers[**"diff"**] = value1 - value2  
 all\_answers[**"prod"**] = value1 \* value2  
 all\_answers[**"quot"**] = value1 / value2  
  
# -- presentation (I/0) code -- #  
CalcListValues(value1 = v1, value2 = v2, all\_answers = lstData)  
print(lstData)  
  
CalcDictionaryValues(value1 = v1, value2 = v2, all\_answers = dicData)  
print(dicData)

Listing 13

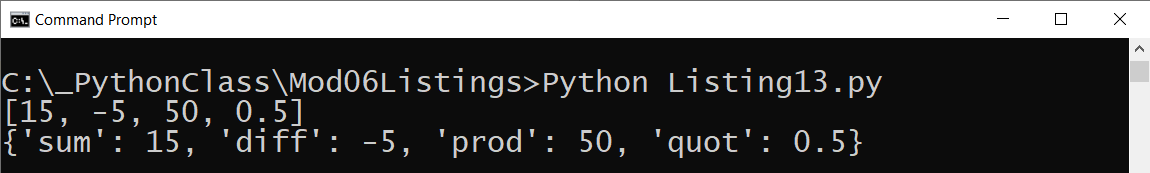


Figure 13. The results of Listing 13

***Note:*** *This code feels* ***more complicated than using a return value to access the results*** *of the function, but it does* ***allow a programmer to use the return values to indicate the status of function in advanced programming scenarios****.*

# Global vs. Local Variables

Variables in a script may be local or global. **Variables declared in a function are considered local to the** **containing function** and cannot be accessed outside of that function. **Variables declared in a "body" of the script are considered global to the containing script** and can be used anywhere in the script.

**Any code inside of the same function can "see" the local variable because that variable would be "inside of its scope!".**

**def** MyFunction():  
 v1 = 15 *# local* print(v1) *# This works!*

**Any code outside of the function cannot "see" the local variable because that variable would be "outside of its scope!"**

**def** MyFunction():  
 v1 = 15 *# local* print(v1) *# This works!*  
  
MyFunction()  
print(v1) *# This causes a "NameError: name 'v1' is not defined"*

Some developers use global variables to access values inside of a function. **Listing 14 shows how a global variable** **could be used to access function results** alongside a **local variable** being used to return results. Using the **return option is preferred.** Note that the code uses **the keyword "global" to indicate a variable is global since the default is local.**

#-------------------------------------------------#  
# Title: Listing 14  
# Description: Global variables  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
#-------------------------------------------------#  
  
# -- data code -- #  
# Note: Variables declared in the body of the script are "Global"  
v1 = 10 # first argument  
v2 = 5 # second argument  
gAnswer = **None** # result of processing  
  
# -- processing code -- #  
**def** AddValues(value1, value2):  
 **global** gAnswer # This refers to the "global" variable  
 gAnswer = value1 + value2  
 answer = value1 + value2 # This is a "local" variable!  
 **return** answer  
  
# -- presentation (I/0) code -- #  
AddValues(v1, v2)  
print(**'Global = '**, gAnswer)  
print(**'Local = '**, AddValues(v1, v2))

Listing 14

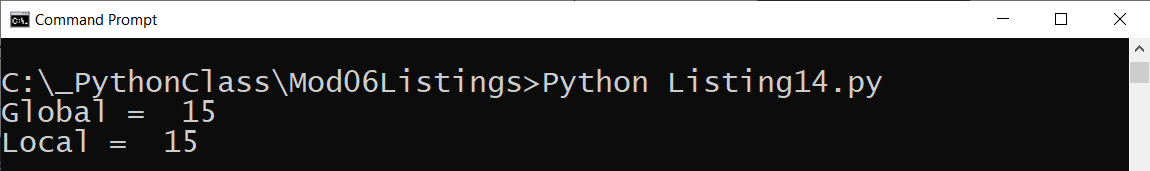


Figure 14. The results of Listing 14

### Shadowing a Global Variable

If you do use global variables within a function, **be careful** to use the keyword "global," or your local variable **will "shadow" the global** one **whenever** **you assign a value to a variable with the same name**!

#-------------------------------------------------#  
# Title: Listing 15  
# Description: Global variable Shadowing  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
#-------------------------------------------------#  
  
# -- data code -- #  
v1 = 10 # first argument  
v2 = 5 # second argument  
answer = **None** # result of processing???  
  
# -- processing code -- #  
**def** AddValues(value1, value2):  
 answer = value1 + value2 # answer "shadowed" the global variable  
  
# -- presentation (I/0) code -- #  
AddValues(v1, v2)  
print(**'Global = '**, answer)

Listing 15

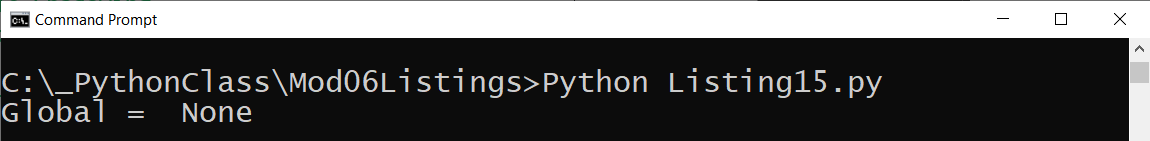
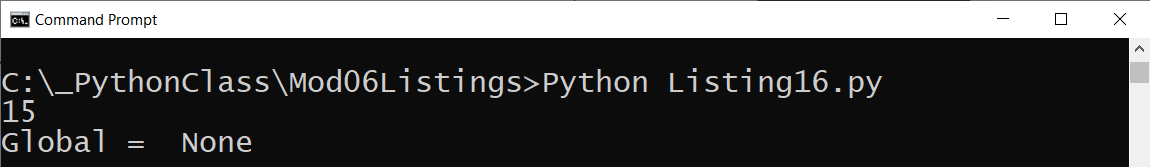


Figure 15. The results of Listing 15

Conveniently, **when you read a variable** in a function with the same name as a global variable, the **shadowing does not happen automatically**. Listing 16 shows an example.

#-------------------------------------------------#  
# Title: Listing 16  
# Description: Global variable implicit reference  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
#-------------------------------------------------#  
  
# -- data code -- #  
# Note: Variables declared in the body of the script are "Global"  
v1 = 10   
v2 = 5   
answer = **None**   
  
# -- processing code -- #  
**def** AddValues():  
 answer = v1 + v2 # v1 and v2 refer to the "global" variables  
 print(answer) # answer is a "local" variable  
  
# -- presentation (I/0) code -- #  
AddValues()  
print(**'Global = '**, answer)

**Listing 16**



**Figure 16. The result of Listing 16**

**Important**: **Using global variables within functions is discouraged** in programming since it breaks the concept of "Encapsulation/Abstraction," which we look at in module 8.

***Note:*** *Unlike global variable which could appear in many of the hundreds of line in the main body of the script, it is* ***common to name local variables without a prefix*** *since these variables are easier to locate within the function. This also* ***helps you not shadow a global variable by mistake!***

# Function Document Headers (Doc Strings)

It is a common practice to include a header at the beginning of a function, which is known as *docstring* in python.

def AddValues(value1, value2):

**""" This function adds two values """**

return v1 + v2

It can be helpful when **developers include additional notes in a docstring**. When they do Integrated development environments like **PyCharm can display tooltips** to show you a developer's notes (**use ctrl + q to activate** this option in PyCharm).

**def** AddValues(value1=0.0, value2=0.0):  
 **""" This function adds two values  
  
 :param value1: (float) the first number to add  
 :param value2: (float) the second number to add  
 :return: (float) sum of two numbers  
 """**   
 **return** value1 + value2

print(AddValues(5,10)) **# Use ctrl + q in PyCharm to see docstrings**

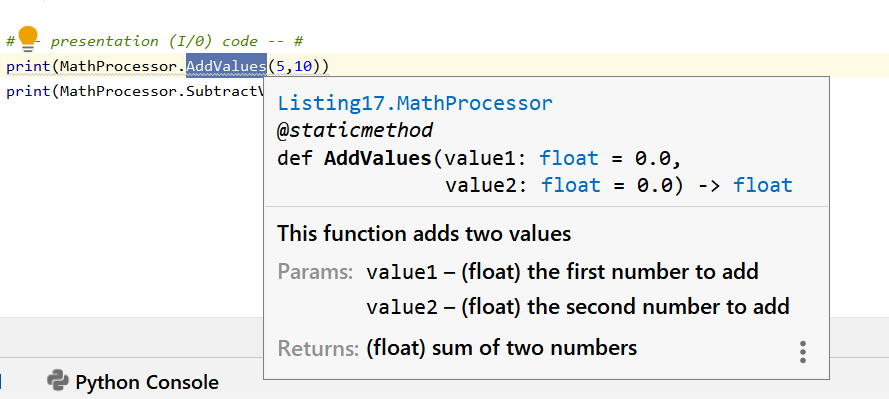


Figure 17. Showing the docstring text using PyCharm

***Note****: PyCharm includes some basic tooltip text even without a docstring.*

# Classes and Functions

**Classes** are a way of **grouping functions, variables, and constants**. **We cover** more about classes and why you might type the "@staticmethod" directive before the of the function's name **in module 08**, but for now, Listing 17 simple class of functions.

# -------------------------------------------------#  
# Title: Listing 17  
# Description: Using classes  
# ChangeLog: (Who, When, What)  
# RRoot, 01.01.2030, Created Script  
# -------------------------------------------------#  
  
# -- processing code -- #  
**class** MathProcessor():  
 """ functions for processing simple math """

@staticmethod  
 **def** AddValues(value1=0.0, value2=0.0):  
 """ This function adds two values

**:param** value1: (float) the first number to add  
 **:param** value2: (float) the second number to add  
 **:return**: (float) sum of two numbers  
 """  
 **return** float(value1 + value2)  
  
 @staticmethod  
 **def** SubtractValues(value1=0, value2=0):  
 """ This function subtracts two values

**:param** value1: (float) the first number to subtract  
 **:param** value2: (float) the second number to subtract  
 **:return**: (float) sum of two numbers  
 """  
 **return** float(value1 - value2)  
  
# -- presentation (I/0) code -- #  
print(MathProcessor.AddValues(5,10))  
print(MathProcessor.SubtractValues(5,10))

Listing 17

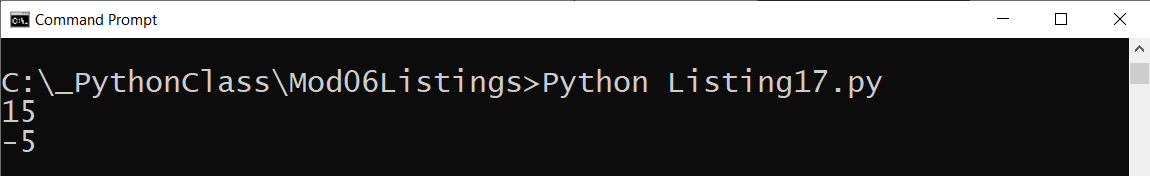


Figure 18. The results of Listing 17

# LAB 6-3: Creating a Class of Functions

In this lab, you **create a class that includes four functions, each returning either the Sum, Difference, Product, and Quotient** of two numbers as a float result. Make your results look like Figure 19.

1. **Create a new script called Lab6-3.**

2. **Add** code to create a **class called MathProcessor**

3. **Add** a **function called AddValues** to process the sum of two values and return the result as a float.

4. **Add** a **function called SubtractValues** to process the difference of two values and return the result as a float.

5. **Add** a **function called MultiplyValues** to process the product of two values and return the result as a float.

6. **Add** a **function called DivideValues** to process the quotient of two values and return the result as a float.

7. **Add** code to **capture two values** from a user.

8. **Send** **the two values to the four functions** to return the sum, difference, product, and quotient of the two values entered by the user.

9. **Print** the results to the user, as shown in Figure 19.

10. **Test** the script and **write** down how the code works.

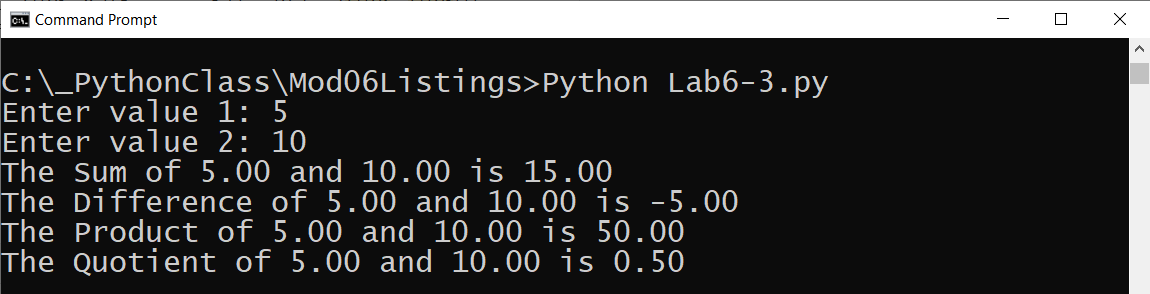


Figure 19. The results of Lab 6-3 using values 5 and 10

# Using the PyCharm Debugger

Separating code into functions and classes make your code easier to read and to debug since smaller sets of statement are easier to test. Still, developers have found that using an Integrated Development Environment (IDE) **debugging tool can make finding and fixing bugs much faster**.

It is **easy to start** using PyCharm's debugging tools, even though the IDE includes many **advanced options that can be confusing for new developers**. Let's look are the options you need to know to get started using them.

## Debug mode

Frequently I run my code in PyCharm using the "Run" option from the context menu. You access this menu by right-clicking any location within the script file.

Notice that there is a "**Debug**" option in the same context menu. Using this **option runs the code using the PyCharm debugging tools** (Figure 20).

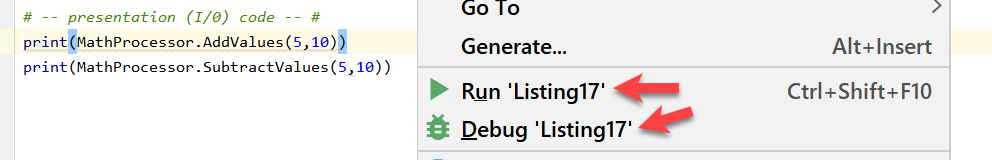


Figure 20. Running you code in debug mode

When the code starts running you will notice that the **results window looks different** in run mode than it does in debug mode. **In debug mode you can switch** between the console display or a set of debugging windows using the **Debugger | > Console tab**. This table can be hard to see, so I have highlighted in in (Figure 21).

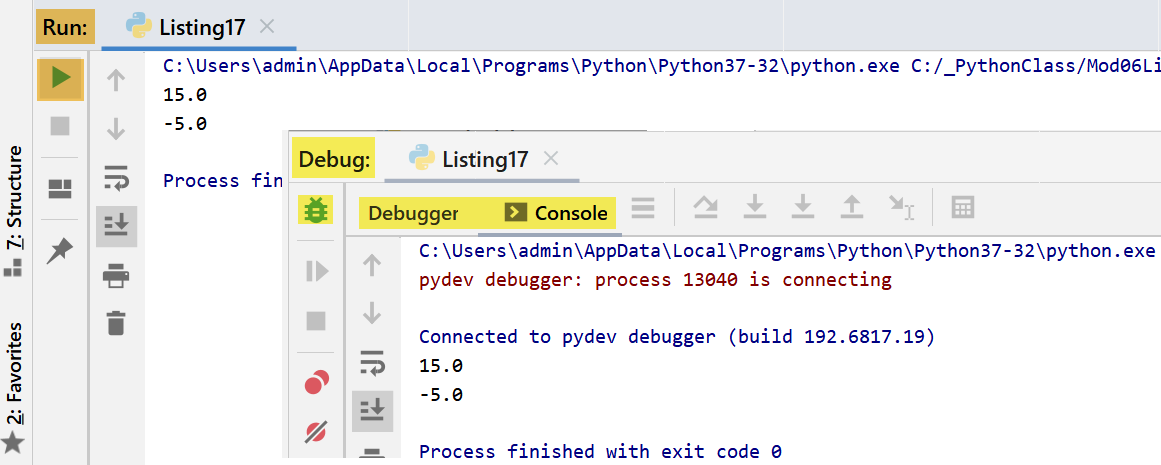


Figure 21. The different look of the run and debugging results window

## Breakpoints

To use the set of Debugging windows you need to set a "breakpoint" in your code. A **breakpoint tells the debugger when to pause running the code** **so that you can exam the information in the debugging windows**.

To set a breakpoint, locate a line of code where you would like the debugger to pause and left-click on the margin left of your code. Doing so will make a red dot appear, indicating the breakpoint has been added (Figure 22).

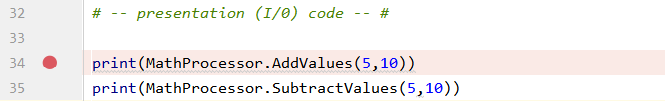


Figure 22. Setting a breakpoint

The next time you **run your code in debug mode**, the code **pauses** and lets you use the controls in debugging windows. Here is a description of the important controls noted in Figure 23:

1. The tab that lets you navigate between the debugging and console windows
2. The "Variable" window that shows which variables are being used and their values
3. The "Step Over" button that allows you to skip over seeing the code in a function when it is called.
4. The "Step Into My Code" button that allows you to step into the next line of your code, without showing you lines that the Python runtime user to run you code (Which can be confusing!)
5. The "Step Out" button that allows you to stop showing the code in a called function and return to the line of code that called it.
6. The "Stop" button that allows you to stop debugging and running the script.

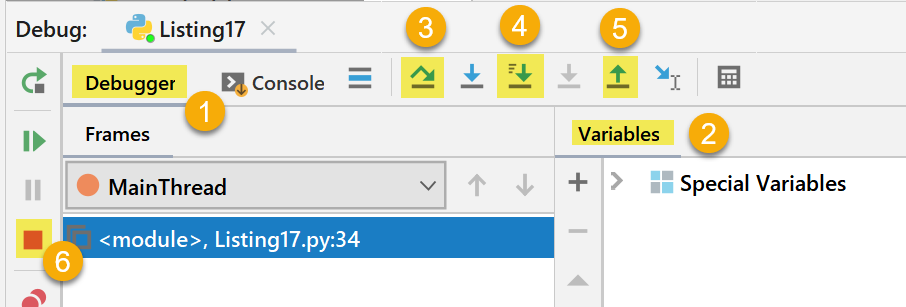


Figure 23. Commonly used controls on the debugging window

### Walking Through Code

The control you use the most is the "**Step Into My Code**" button. Using it, **allows you to "walk" through each line of code** **that performs an actions**. Code that does not perform actions, such as comments are skipped.

When a line of code is reached that contains variables, you can either hover over that variable to see what is current held in memory or look at the "Variables" window to see all the variable currently being used (Figure 24).

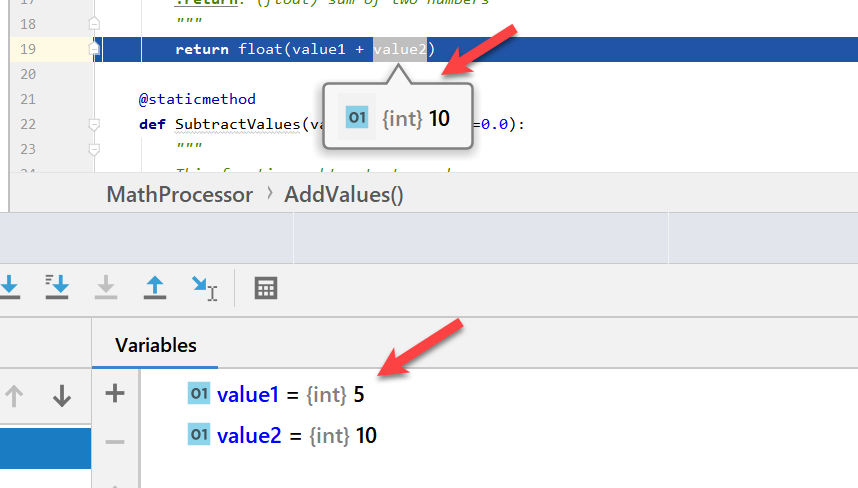


Figure 24. Viewing information about variables in PyCharm debug mode

Note that **both the value and the data type are shown** in PyCharm. This is **useful since Python automatically select a variables datatype** for you, **which can cause some hard to find bugs**!

***Important:*** *The only way to learn how to use the debugging tools is to just start using them! So, try to use the debugger at least once for each assignment going forward!*

# Creating a GitHub Webpage

In module 5, we saw how developers can upload sets of files to a GitHub website. In this module, **learn how to create simple** web pages to support and enhance a GitHub repository using "**GitHub Pages**."

To demonstrate this, let's **make a new GitHub repository called "our class name" - "Mod06,"** as shown in Figure 25.

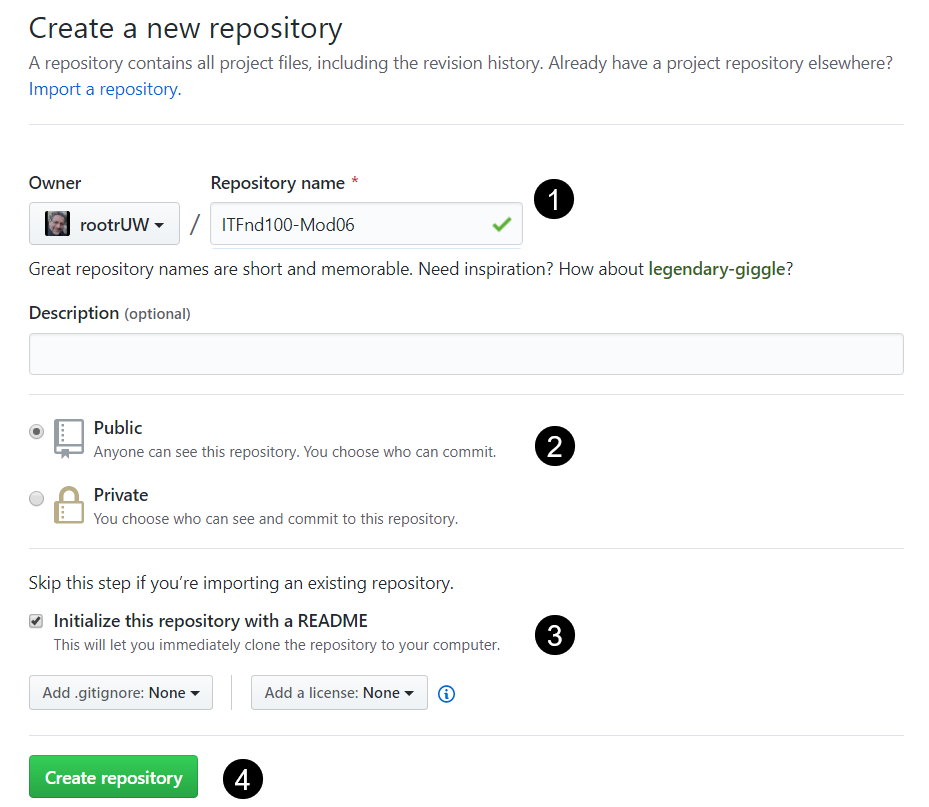


Figure 25. Creating a new GitHub repository

With the repository created, we **add a new folder**, with a file inside of it, using the "Create new file" button (Figure 26).

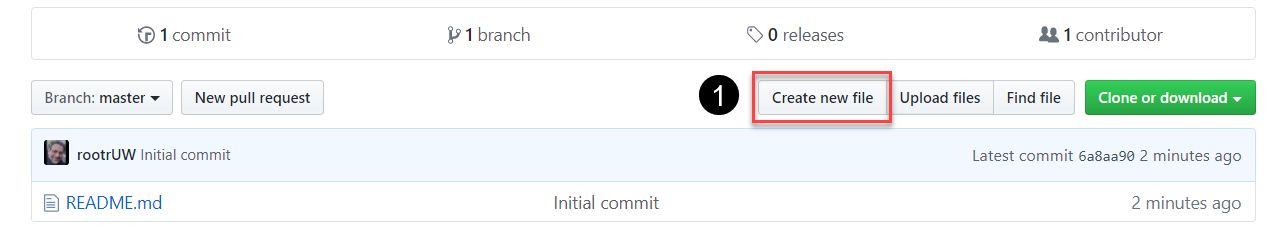


Figure 26. Creating a new folder and file using the "Create new file" button

When you **click the "Create new file" button**, a new textbox appears on the page. In the new textbox, start typing the name of the folder you want to place your web files. We want to **use the name "docs" for the folder**, so I type that name into the textbox (Figure 27.)

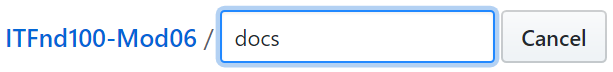


Figure 27. Adding a "publishing source" folder called "docs"

Next, **type in a forward-slash (/)**, **then** the **name of a file** you want to create in the new folder. The recommended "default" name of the first web file is "**index.md**" so we use that in Figure 28. Now, **add some simple text** to the new "index.md" page. It does not matter what you type in the page at first since it is **easy to edit it later**, but you must type in something

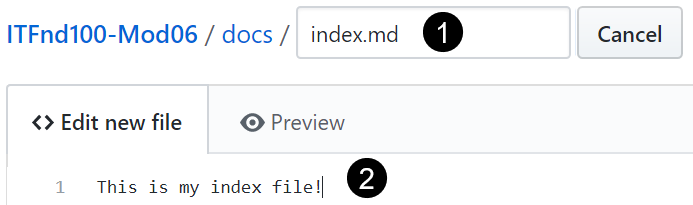


Figure 28. Adding a "default" web page file called "index.md" with simple text

Finally, scroll to the bottom of the web page and **click the "Commit new file" button** to create the new "docs" folder and "index.md" file (Figure 29).

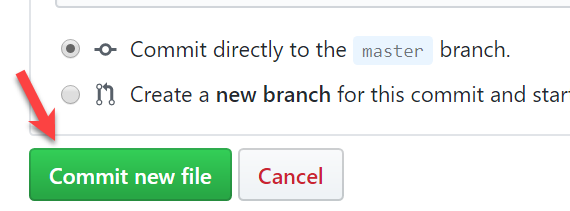


Figure 29. Using the "Commit new file" button to create the folder and file

## Configuring the Website

You need to **tell GitHub to use your new file and folder**. You do so by accessing the configuration under the **"Setting" tab** (Figure 30).



Figure 30. The Settings tab

Once on the settings tab, scroll down to the "GitHub Pages" section and **select the "master branch/docs folder" option in the "source" dropdown box** (Figure 31).

This selection starts the website generation, which **can take a few minutes to complete**. Try **navigating to your new page using the link** that appears after you make your selection (Figure 31).

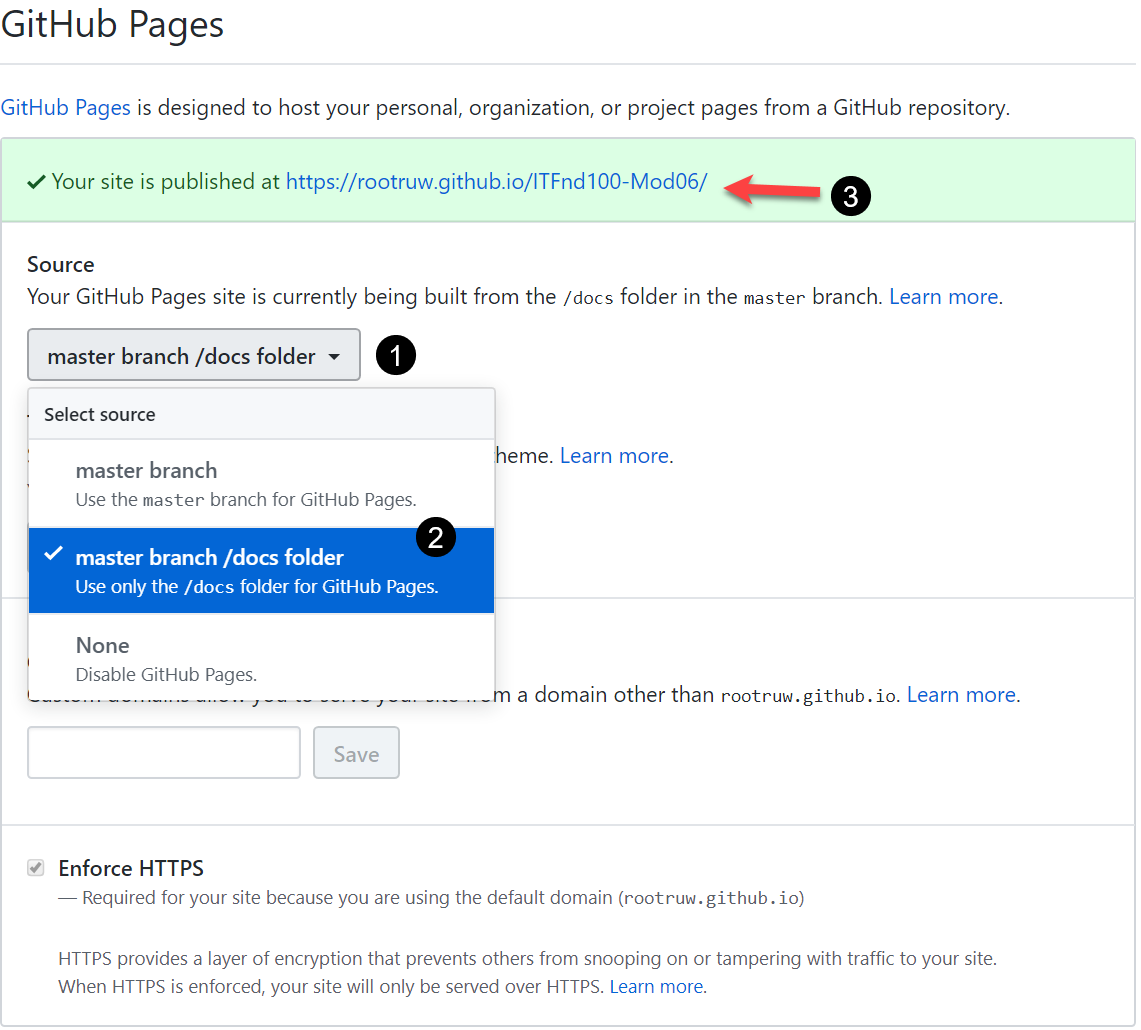


Figure 31. Configuring the GitHub Pages

Once GitHub finishes building your page, **it will look like the one in Figure 32.**

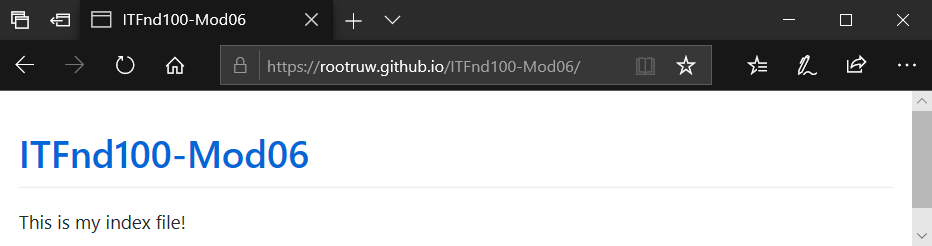


Figure 32. A simple example of a GitHub website

***Important:*** *I can take up to 20 minutes before your content shows up on the website, so be patient.*

## Modifying the GitHub page

After you create the website, you can **modify it by editing the "index.md" page**. For example, I might add the code in Listing 19 to the file to create a set of hyperlinks.

**Note:** In this language, the **square brackets hold the link's text, and parentheses hold the URL**.

# Module06 Website

---

[Google Homepage](https://www.google.com "Google's Homepage")

[Typical Assignment Document](https://github.com/rootrUW/ITFnd100-Mod06/blob/master/\_A\_Typical\_Assignment\_Document.pdf)

[GitHub Webpage Code CheatSheet](https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet)

Listing 20

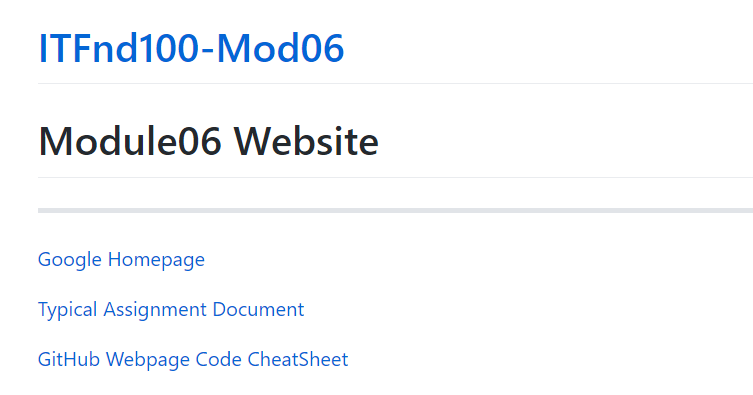


Figure 28. The result of the code in Listing 19

You can also **go back into Settings and add a Theme** to the website to greatly improve its look (Figure 29)!

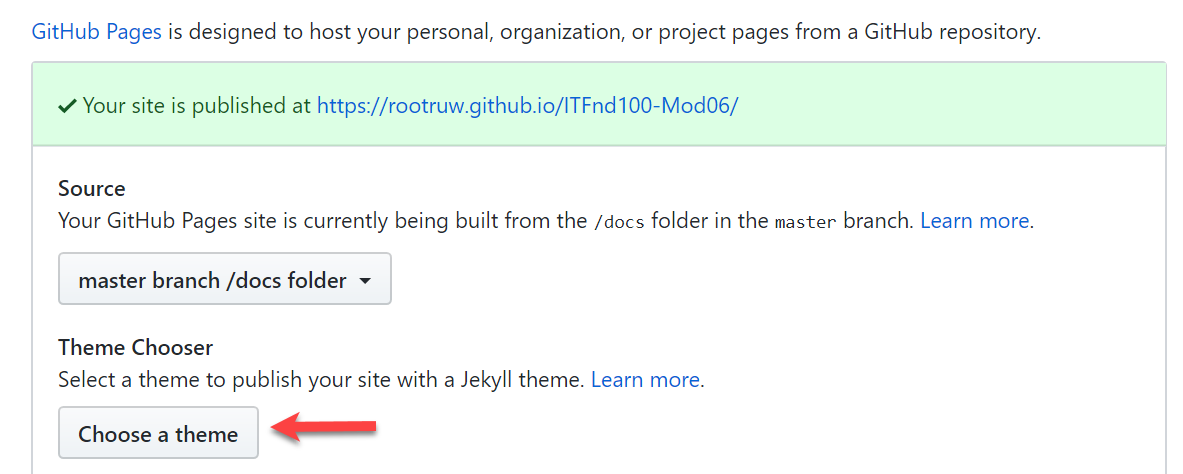


Figure 29. Choosing a GitHub pages theme

***Tip:*** *You can learn more about how to code your GitHub Pages using this URL:* [*https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet*](https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet)

# Summary

In this module, we looked at how you use functions to organize your code and classes to organize your functions. We also looked at ways you can improve your GitHub site to look more professional.

At this point, you should try and answer as many of the following questions as you can from memory, then review the subjects that you cannot. Imagine being asked these questions by a co-worker or in an interview to connect this learning with aspects of your life.

* What is a function?
* What are parameters?
* What are arguments?
* What is the difference between parameters and arguments?
* What are return values?
* What is the difference between a global and a local variable?
* How do you use functions to organize your code?
* What is the difference between a function and a class?
* How do functions help you program using the “Separations of Concerns" pattern?
* How are the debugging tools used in PyCharm?
* What is a GitHub webpage?

When you can answer all of these from memory, it is time to complete the module's assignment and move on to the next module.