PROGRAMMING WITH PYTHON

By Randal Root

Module 05

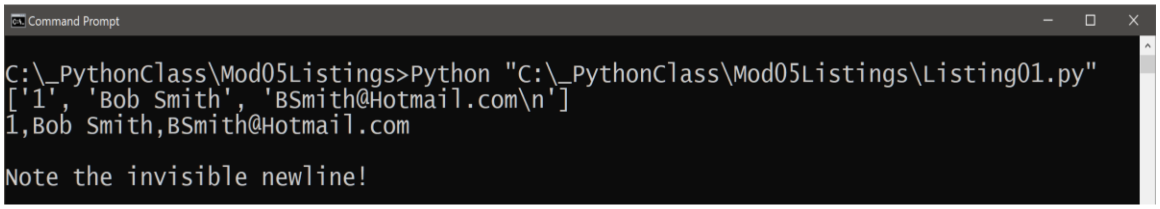
In this module, you learn about **creating scripts using Lists and Dictionaries**. You also learn some **basics** about **error handling**, **functions, script templates, and GitHub**.

Lists

Lists are a simple way to hold a collection of objects, and Lists include a lot of built-in functions to help you work with those objects. Listing 1 shows how a List holds and used to present data.

```
# ------------------------------------------------- #
# Title: Listing 1
# Description: Writing to and reading from a list
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created script
# ------------------------------------------------- #

# Declare my variables
lstRow = []

# Process the data
lstRow = ["1", "Bob Smith", "BSmith@Hotmail.com\n"]
print(lstRow)
print(lstRow[0] + ',' + lstRow[1] + ',' + lstRow[2])
print('Note the invisible newline!')
```

Listing 1



```
C:\_PythonClass\Mod05Listings>Python "C:\_PythonClass\Mod05Listings\Listing01.py"
['1', 'Bob Smith', 'BSmith@Hotmail.com\n']
1,Bob Smith,BSmith@Hotmail.com

Note the invisible newline!
```

Figure 1. The results of listing 1

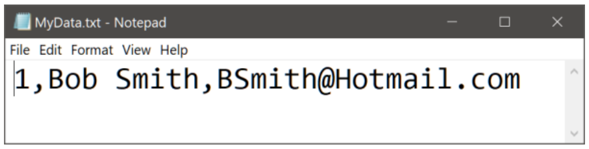
Storing List Data in a File

Lists always **hold data in a computer's memory**. Memory is cleared, and all **data is lost when an application closes**. When a developer wants a user's **data to persist** between running the application, they often store data **in a file**. The code in listing 2 demonstrates this simple process.

```
# ------------------------------------------------- #
# Title: Listing 2
# Description: Writing Data from a list to a file
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created script
# ------------------------------------------------- #

# Declare my variables
lstRow = []
strFile = 'MyData.txt'
objFile = None
```

```
# Process the data
objFile = open(strFile, "w")
lstRow = ["1", "Bob Smith", "BSmith@Hotmail.com"]
objFile.write(lstRow[0] + ',' + lstRow[1] + ',' + lstRow[2] + '\n')
objFile.close()
```

Listing 2



```
1,Bob Smith,BSmith@Hotmail.com
```

Figure 2. The results of listing 2

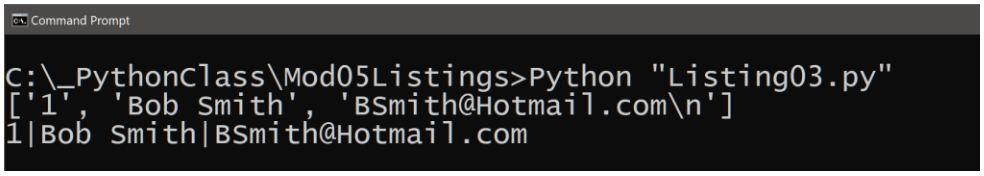
Loading List Data from a File

Data in a file **must be loaded back into memory** if the user wants to continue using this stored data. Since L**ists** are simple and have many useful functions, developers **often load text file data into them**. Listing 2 shows a simple way to do this.

Note the use of the split() function to **separate elements** of the new list based on comas found in the text file. Also note the use of the **strip() function** to **remove the unwanted carriage retur**n.

```
# -------------------------------------------------- #
# Title: Listing 3
# Description: Reading Data from a file to a list
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created script
# -------------------------------------------------- #

# Declare my variables
lstRow = []
strFile = 'MyData.txt'
objFile = None

# Process the data
objFile = open(strFile, "r")
for row in objFile:
    lstRow = row.split(",")
    print(lstRow)
    print(lstRow[0] + '|' + lstRow[1] + '|' + lstRow[2].strip())
objFile.close()
```

Listing 3



```
C:\_PythonClass\Mod05Listings>Python "Listing03.py"
['1', 'Bob Smith', 'BSmith@Hotmail.com\n']
1|Bob Smith|BSmith@Hotmail.com
```

Figure 3. The result of listing 3

Lab 5-1 Working with Files and Lists

In this lab, you **write and read home inventory data using a text file**. **Use** the data listed in table 1 and **starter code** is shown in listing 4 to perform this lab.

Item	Value
Lamp	$30
End Table	$60

Table 1

```
# ---------------------------------------------------- #
# Title: Lab 5-1
# Description: Writing and Reading Data from a file
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# <Your Name Here>,<Date>, Added read/write to file code
# ---------------------------------------------------- #

# Declare my variables
strChoice = '' # User input
lstRow = []  # list of data
strFile = 'HomeInventory.txt'  # data storage file
objFile = None  # file handle

# Get user Input
while(True):
    print("Write or Read file data, then type 'Exit' to quit!")
    strChoice = input("Choose to [W]rite or [R]ead data: ")

    # Process the data
    if (strChoice.lower() == 'exit'): break
    elif (strChoice.lower() == 'w'):
        # List to File
        print('ToDo: add code here')
    elif (strChoice.lower() == 'r'):
        # File to List
        print('ToDo: add code here')
    else:
        print('Please choose either W or R!')
```

Listing 4

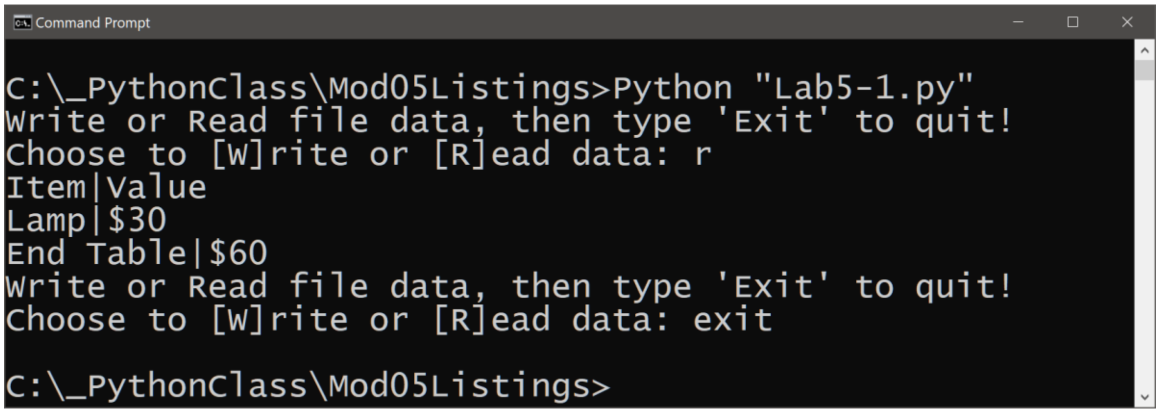
1. **Review** the code in Listing 4 and make some simple notes about what it is trying to accomplish.

2. **Copy** and **paste** the code from Listing 4 into a new script file called Lab5-1.

3. **Replace** your name and date in the script header's "changelog" section.

4. **Replace** the statements, "print('ToDo: add code here')," with code to accomplish the tasks of writing to and reading from a file.

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



```
C:\_PythonClass\Mod05Listings>Python "Lab5-1.py"
Write or Read file data, then type 'Exit' to quit!
Choose to [W]rite or [R]ead data: r
Item|Value
Lamp|$30
End Table|$60
Write or Read file data, then type 'Exit' to quit!
Choose to [W]rite or [R]ead data: exit

C:\_PythonClass\Mod05Listings>
```

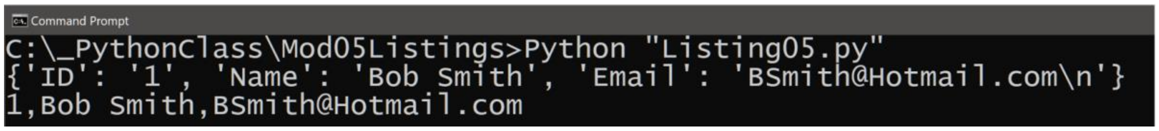
Figure 4. The desired results of listing 4

Dictionaries

We have seen how you access elements of a **List, Tuple, or String** sequence using an **index (numeric) subscript**. **Dictionaries** are very similar to these other sequences, but they replace the index subscripts with **key (character) subscripts**. You **use the braces {} operator** to indicate that you want a variable to be a dictionary.

```
# -------------------------------------------------- #
# Title: Listing 5
# Description: Reading and Writing to a dictionary
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created script
# -------------------------------------------------- #

# Declare my variables
dicRow = {}

# Process the data
dicRow = {"id":"1", "name":"Bob Smith", "email":"BSmith@Hotmail.com"}
print(dicRow)
print(dicRow["id"] + ',' + dicRow["name"] + ',' + dicRow["email"])
```

Listing 5



```
C:\_PythonClass\Mod05Listings>Python "Listing05.py"
{'ID': '1', 'Name': 'Bob Smith', 'Email': 'BSmith@Hotmail.com\n'}
1,Bob Smith,BSmith@Hotmail.com
```

Figure 5 The results of listing 5

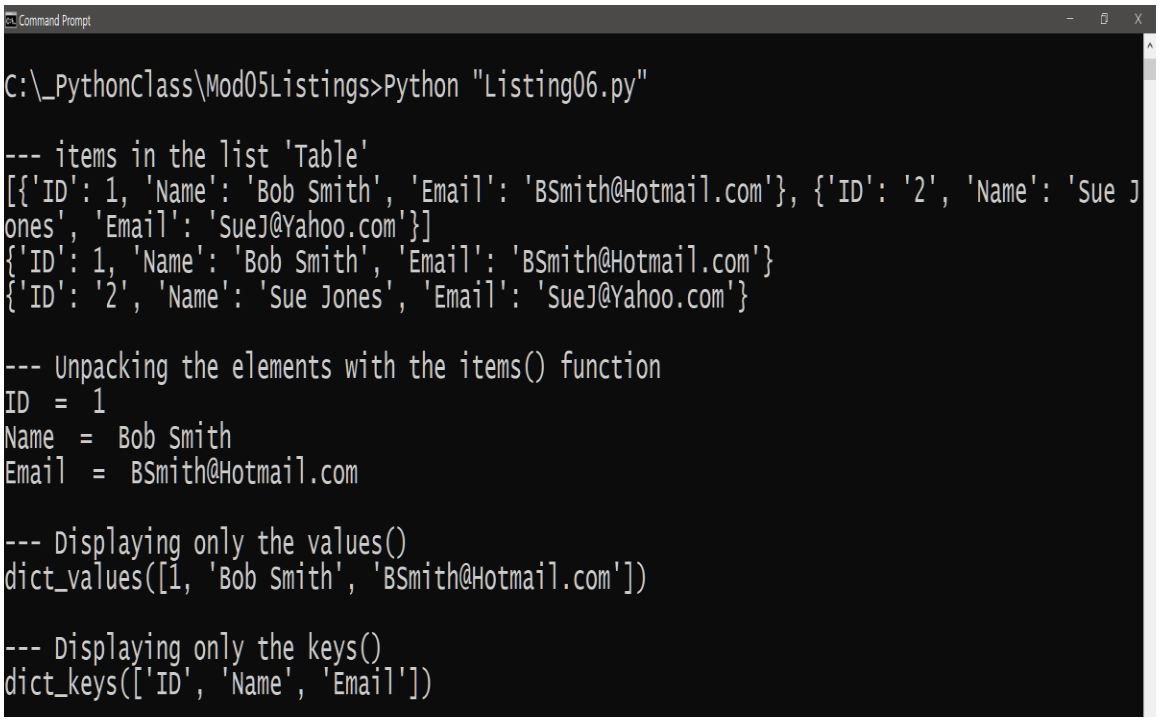
Working with Dictionaries

Dictionary keys are a lot like columns in a spreadsheet or database. As such, it is **helpful think for a dictionary as a row of data**. These “**rows” can be added to a List** to form a collection of rows, which **creates a table** like two-dimensional collection of data.

Like the Python List class, the **dictionary class has special built-in methods** that help you work with the data and keys. Listing 6 shows how the **items(), values(), and keys()** functions work.

```
# -------------------------------------------------- #
# Title: Listing 6
# Description: Working with a list of dictionary objects
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created script
# -------------------------------------------------- #

# Create a data structure
dicRow1 = {"ID":1,"Name":"Bob Smith", "Email":"BSmith@Hotmail.com"}
dicRow2 = {"ID":"2","Name":"Sue Jones", "Email":"SueJ@Yahoo.com"}
lstTable = [dicRow1, dicRow2]

# Process the data
print("\n--- items in the list 'Table'")
print(lstTable)
for objRow in lstTable:
    print(objRow)

print("\n--- Unpacking the elements with the items() function")
for myKey, myValue in dicRow1.items():
    print(myKey, " = ", myValue )

print("\n--- Displaying only the values()")
print(dicRow1.values())

print("\n--- Displaying only the keys()")
print(dicRow1.keys())
```

Listing 6



```
C:\_PythonClass\Mod05Listings>Python "Listing06.py"

--- items in the list 'Table'
[{'ID': 1, 'Name': 'Bob Smith', 'Email': 'BSmith@Hotmail.com'}, {'ID': '2', 'Name': 'Sue J
ones', 'Email': 'SueJ@Yahoo.com'}]
{'ID': 1, 'Name': 'Bob Smith', 'Email': 'BSmith@Hotmail.com'}
{'ID': '2', 'Name': 'Sue Jones', 'Email': 'SueJ@Yahoo.com'}

--- Unpacking the elements with the items() function
ID  =  1
Name  =  Bob Smith
Email  =  BSmith@Hotmail.com

--- Displaying only the values()
dict_values([1, 'Bob Smith', 'BSmith@Hotmail.com'])

--- Displaying only the keys()
dict_keys(['ID', 'Name', 'Email'])
```

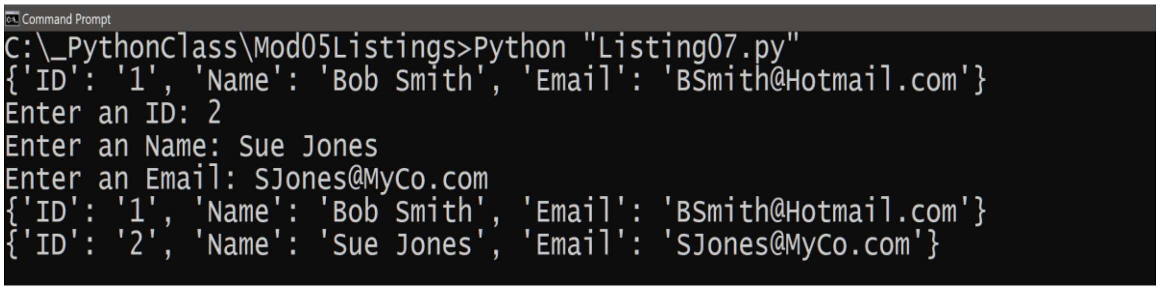
Figure 6. The results of listing 6

Just like a Python List, **you can add user's input or text file data into a Dictionary**. Listing 7 shows an **example** of **collecting a "row" of data from a user's input** and adding it to a dictionary.

```
# -------------------------------------------------- #
# Title: Listing 7
# Description: Adding user data to the "table"
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created script
# -------------------------------------------------- #

# Data
strFile = 'MyData.txt'
objFile = None
dicRow = {}
lstTable = []

# Create the table
dicRow = {"id":"1","name":"Bob Smith", "email":"BSmith@Hotmail.com"}
lstTable.append(dicRow)

# Process the data
for objRow in lstTable:
    print(objRow)
```

```
# Get User Input
strID = input("Enter an ID: ")
strName = input("Enter an Name: ")
strEmail = input("Enter an Email: ")
dicRow = {"id":strID,"name":strName, "email":strEmail}
lstTable.append(dicRow)
for objRow in lstTable:
    print(objRow)
```

Listing 7



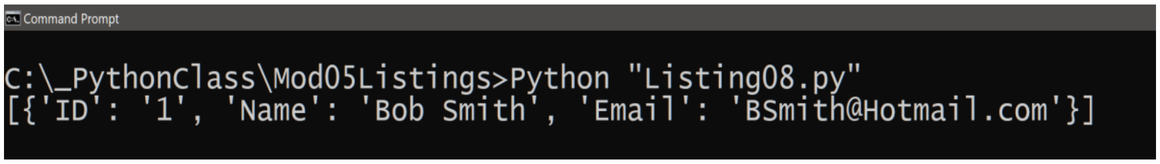
```
C:\_PythonClass\Mod05Listings>Python "Listing07.py"
{'ID': '1', 'Name': 'Bob Smith', 'Email': 'BSmith@Hotmail.com'}
Enter an ID: 2
Enter an Name: Sue Jones
Enter an Email: SJones@MyCo.com
{'ID': '1', 'Name': 'Bob Smith', 'Email': 'BSmith@Hotmail.com'}
{'ID': '2', 'Name': 'Sue Jones', 'Email': 'SJones@MyCo.com'}
```

Figure 7. The results of listing 7

Listing 8 shows an **example** of **reading a "row" of data from a text file** and adding it to a dictionary.

```
# ------------------------------------------------- #
# Title: Listing 8
# Description: Adding file data to the "table"
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created script
# ------------------------------------------------- #

# Declare my variables
strFile = 'MyData.txt'
objFile = None
dicRow = {}
lstTable = []

# Process the data
objFile = open(strFile, "r")
for row in objFile:
  lstRow = row.split(",")
  dicRow = {"id":lstRow[0],"name":lstRow[1],"email":lstRow[2].strip()}
  lstTable.append(dicRow)
objFile.close()

print(lstTable)
```

Listing 8



```
C:\_PythonClass\Mod05Listings>Python "Listing08.py"
[{'ID': '1', 'Name': 'Bob Smith', 'Email': 'BSmith@Hotmail.com'}]
```

Figure 8. The results of listing 8

LAB 5-2: Working with a Table of Dictionaries

In this lab, you modify the code from Lab 5-1 to write and read home inventory data in a text file from and to a list of dictionary objects.

1. **Review** the code in Listing 9 and make some simple notes about what it is trying to accomplish.

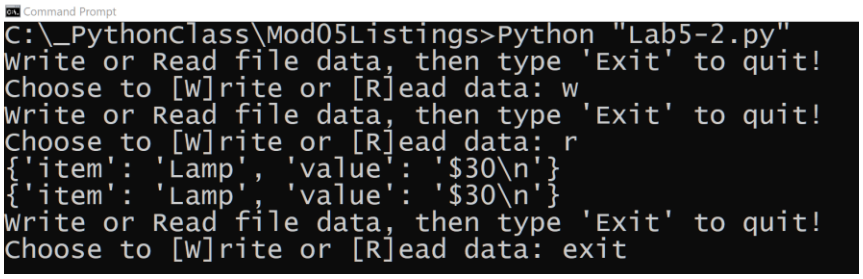
```
# ------------------------------------------------- #
# Title: Listing 9
# Description: Writing and Reading Data from a file
# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# <Your Name Here>,<Date>,Changed rows from lists to dictionaries
# ------------------------------------------------- #

# Declare my variables
strChoice = '' # User input
lstRow = []  # list of data
strFile = 'HomeInventory.txt'  # data storage file
objFile = None  # file handle

# Get user Input
while(True):
    print("Write or Read file data, then type 'Exit' to quit!")
    strChoice = input("Choose to [W]rite or [R]ead data: ")

    # Process the data
    if (strChoice.lower() == 'exit'): break
    elif (strChoice.lower() == 'w'):
        # List to File
        objFile = open(strFile, "w")
        lstRow = ["Lamp", "$30"]
        objFile.write(lstRow[0] + ',' + lstRow[1] + '\n')
        lstRow = ["End Table", "$60"]
        objFile.write(lstRow[0] + ',' + lstRow[1] + '\n')
        objFile.close()
    elif (strChoice.lower() == 'r'):
        # File to List
        objFile = open(strFile, "r")
        for row in objFile:
            lstRow = row.split(",") # Returns a list!
            print(lstRow[0] + '|' + lstRow[1].strip())
        objFile.close()
    else:
        print('Please choose either W or R!')
```
Listing 9

2. **Copy** and **paste** the code from Listing 9 into a new script file called Lab5-2.

3. **Replace** your name and date in the script header's "changelog" section.

4. **Locate and replace** the code that uses list objects to store rows of data so that it uses dictionary objects instead.

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



```
C:\_PythonClass\Mod05Listings>Python "Lab5-2.py"
Write or Read file data, then type 'Exit' to quit!
Choose to [W]rite or [R]ead data: w
Write or Read file data, then type 'Exit' to quit!
Choose to [W]rite or [R]ead data: r
{'item': 'Lamp', 'value': '$30\n'}
{'item': 'Lamp', 'value': '$30\n'}
Write or Read file data, then type 'Exit' to quit!
Choose to [W]rite or [R]ead data: exit
```

Figure 9. The desired results of listing 9

6. **Review** the results in Figure 9 and **write** down **one thing you like** about the results and **one thing that could make it better**.

Improving Your Scripts

As your code becomes complex and your scripts become large, the need to **organize and use more professional coding practices** becomes vital. Let's look at four simple techniques to improve your scripts; **the separation of concerns programming pattern, functions, script templates, and structured error handling**.

Separation of Concerns

"In computer science, separation of concerns (SoC) is a design principle for separating a computer program into distinct sections, so that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program." **https://en.wikipedia.org/wiki/Separation_of_concerns**, 2019

Most programs can be divided into three different sections; Data, Processing, and Presentation (or Input-Output). You should **practice dividing your code** into these sections as soon as you can since it takes practice before you become the process. For instance, you might start each new script by adding the set of comments shown in listing 10.

```
#-- Data --#
# Example: Declare variables and constants

#-- Processing --#
# Example: Perform tasks on data

#-- Presentation (Input/Output) --#
# Example: Get user input
```

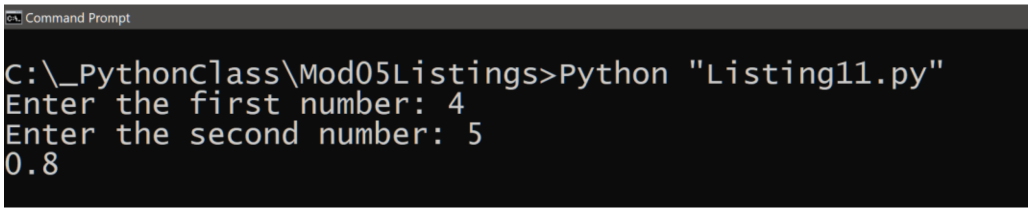
Listing 10

The **name and order of these sections change based on the language and design** of your program, but **listing 11 shows a simple example** of how this might work.

Note, the order of the executed statements, and how **the sixth statement is not is in the presentation section even though it is presentation code** and be aware that this is a typical outcome. **You may not always be able to divide your code completely** into each section, and that is **OK as long as you do so whenever it is possible**!

```
# -- Data -- #
fltN1 = 0.0  # 1
fltN2 = 0.0  # 2

# -- Presentation (Input/Output) -- #
fltN1 = float(input("Enter the first number: "))  # 3
fltN2 = float(input("Enter the second number: ")) # 4

# -- Processing -- #
fltQuot = (fltN1 / fltN2)  # 5
print(fltQuot)  # 6, presentation code, but still needed here!
```

Listing 11



```
C:\_PythonClass\Mod05Listings>Python "Listing11.py"
Enter the first number: 4
Enter the second number: 5
0.8
```

Figure 11. The results of listing 11

Functions

One thing that **helps you when sectioning your code** is functions. Functions **allow you to group a set of programming statements** and later reference them **by** a given **name**. In Python, function **must be defined in a script before they can be called**. The statements of a function run later in your program by “calling” the function.

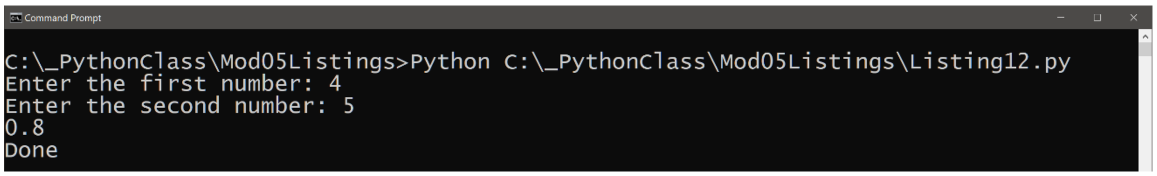
For example, we could better organize our code in listing 11 using a new function called DivideValues(). When the script starts, it **loads the function into memory but waits to run its statements**. Listing 12 shows an example of this.

Note that the **code starts running the first statements of the script; in listing 12, this sets up the variables. After the variables are created, it is loads the definition of the function** into memory and then moves on to the next statements, and then the next, and then the next.

When Python gets to the line that calls to the function it jumps to the section of code where the function is defined. It runs all the code inside the function and **then returns to the line of code directly after the function call and continues running all the lines until the end of the script**. In listing 12, I have numbered these steps with comments.

```
# -- Data -- #
fltN1 = 0.0  # 1
fltN2 = 0.0  # 2

# -- Processing -- #
def DivideValues():  # 6, this code loads but does not run yet!
    return (fltN1 / fltN2)  # 7

# -- Presentation (I/O) -- #
fltN1 = float(input("Enter the first number: "))  # 3
fltN2 = float(input("Enter the second number: "))  # 4
print(DivideValues())  # 5
print("Done")  # 8
```

Listing 12



```
C:\_PythonClass\Mod05Listings>Python C:\_PythonClass\Mod05Listings\Listing12.py
Enter the first number: 4
Enter the second number: 5
0.8
Done
```

Figure 12. The results of listing

You can see from this example that the **function makes it easier to separate the presentation and processing code**.

NOTE: We cover details about creating custom functions in module 6

Script Templates

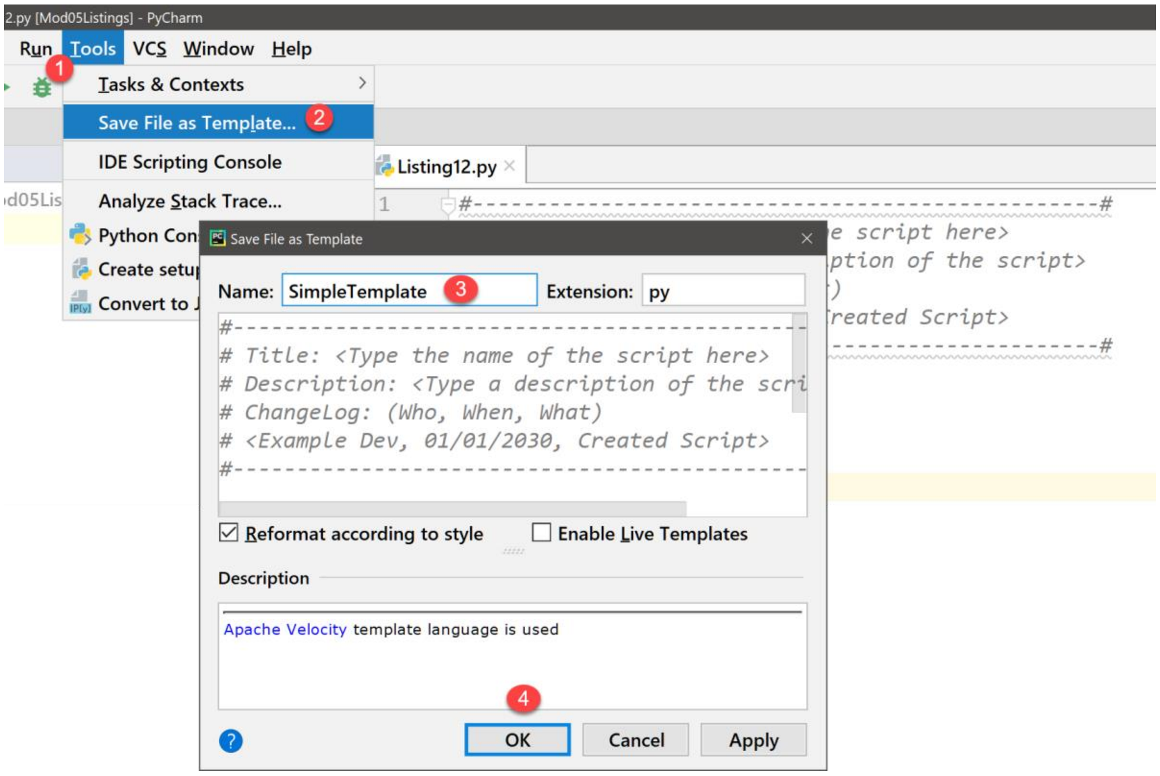
Consistency improves your scripts by making them look more professional and easier to read. Since you know to divide the code into sections of concerns, you would create a script template to help that as well. Listing 13 shows a simple example.

```
# ------------------------------------------------- #
# Title: <Type the name of the script here>
# Description: <Type a description of the script>
# ChangeLog: (Who, When, What)
# <Example Dev,01/01/2030,Created Script>
# ------------------------------------------------- #

# -- Data -- #
# -- Processing -- #
# -- Presentation (I/O) -- #
```

Listing 13

Script Templates in PyCharm

Using script templates is such a common practice that **most advanced Integrated Development Environments (IDE) allow you to save custom templates** and use them with each new file. For example, in figure 13 you can see how to add a new template in **PyCharm** by using the **Tools > Save File as Template… menu option**. This option launches the "Save File as a Template" **dialog window** where you can **name your temple and OK its creation.**



Name: SimpleTemplate Extension: py

```
#-------------------------------------------
# Title: <Type the name of the script here>
# Description: <Type a description of the scri
# ChangeLog: (Who, When, What)
# <Example Dev, 01/01/2030, Created Script>
#-------------------------------------------
```

☑ Reformat according to style ☐ Enable Live Templates

Apache Velocity template language is used

Figure 13. Adding a custom template to PyCharm

Once a template is created, you **use it by right-clicking your project icon** in the Project Explorer tree and **using the New > "Name of your Template" option** in the context menu as shown in figure 14.

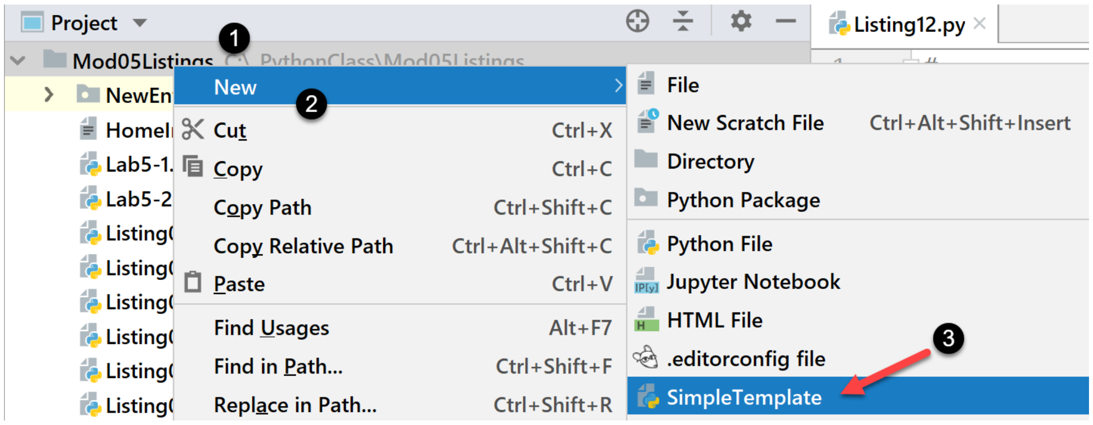


Figure 14. Using a custom template in PyCharm

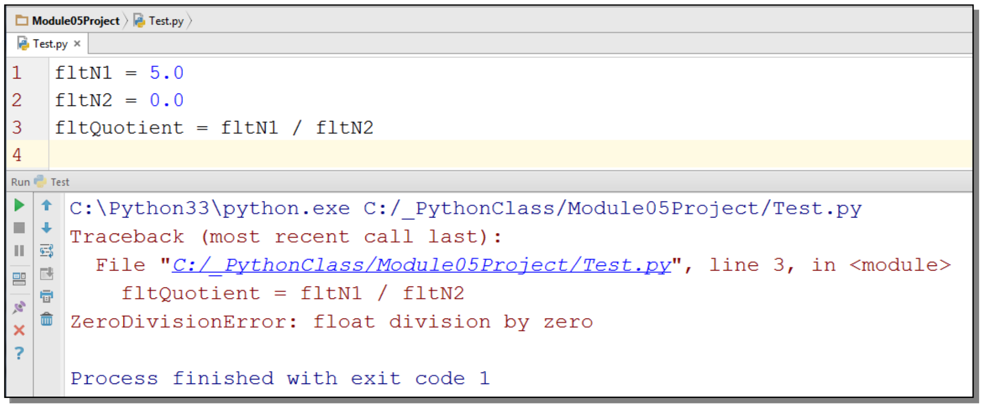
Error Handling (Try-Except)

Error Handing improves your scripts by managing errors you may not have control over in any other way. For **example**, you may ask a **user** for **input** of a number to calculate but receive a character instead. While **you cannot stop people from making mistakes you can actively plan for them** and make the process of recovering from errors less painful.

You can **trap errors in your programs using a try-except construct**. One advantage is its ability to **provide more general and user-friendly error messages**. Another advantage is that it **allows for a simple, organized way of grouping statements to be processed**.

A third advantage is that **if an error occurs in the grouped statements**, Python **automatically moves to another set of statements where you can handle the error** in your own way (instead of the way Python would normally do so).

For example, figure 15 shows how a **normal "divide by zero" error** looks. Note it's a **technical explanation of what occurred**, which is accurate, but **not user-friendly**

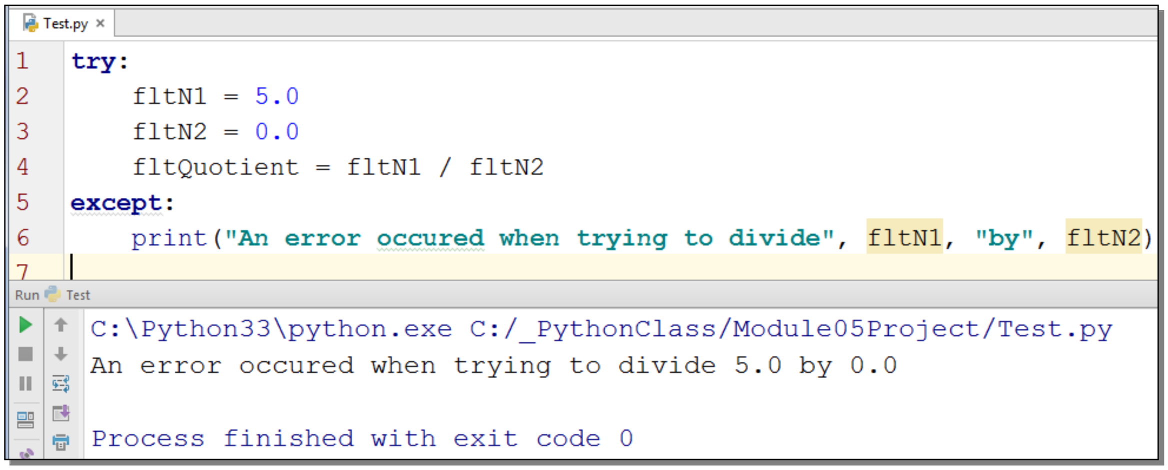


```
fltN1 = 5.0
fltN2 = 0.0
fltQuotient = fltN1 / fltN2
```

```
C:\Python33\python.exe C:/_PythonClass/Module05Project/Test.py
Traceback (most recent call last):
  File "C:/_PythonClass/Module05Project/Test.py", line 3, in <module>
    fltQuotient = fltN1 / fltN2
ZeroDivisionError: float division by zero

Process finished with exit code 1
```

Figure 15. A standard complex error message

Technical messages **work fine for developers**, but they are **too complex for end-users**. A**dding a Try-Except block to your code allows you to customize the error messages** as shown in figure 16.



```
try:
    fltN1 = 5.0
    fltN2 = 0.0
    fltQuotient = fltN1 / fltN2
except:
    print("An error occured when trying to divide", fltN1, "by", fltN2)
```

```
C:\Python33\python.exe C:/_PythonClass/Module05Project/Test.py
An error occured when trying to divide 5.0 by 0.0

Process finished with exit code 0
```

Figure 16. A simple, customized error message

Note: We cover details about using Try-Except in module 6

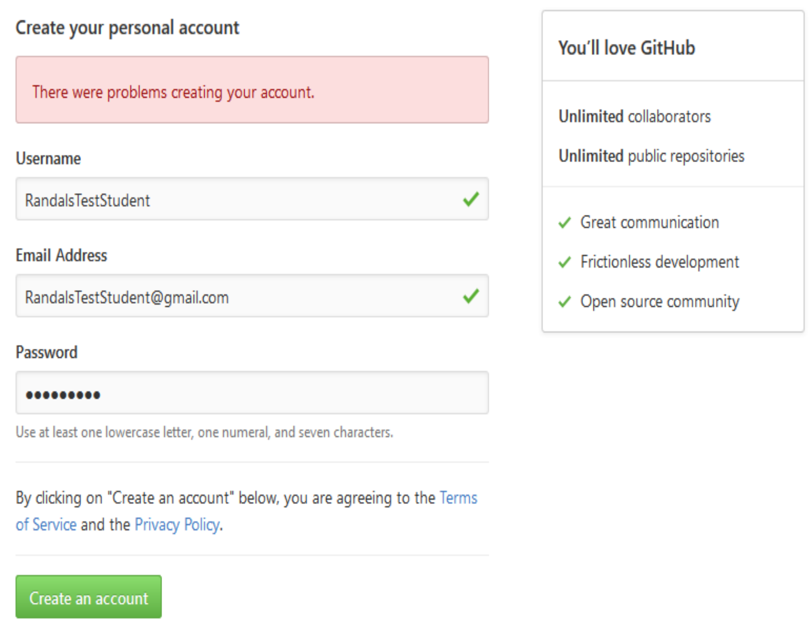
GitHub

It has always been a **good idea to make backups of your code files** and to make them available for others to access. **Traditional** this has been accomplished using a **network share on an organization's server**. While this is still a common practice, more and **more organizations have embraced storing this code on the Internet via source control software.**

One of the most popular source control software is Git, which can store your files on the Internet at GitHub.com. Here is an excerpt from an article on what GitHub is:

> "Version control systems **keep** these **revisions straight**, storing the modifications in a **central repository**. This allows developers to easily **collaborate**, as they can download a new version of the software, make changes, and upload the newest revision. Every developer can see these new changes, download them, and contribute.
>
> …
>
> Git is the preferred version control system of most developers, since it has multiple advantages over the other systems available. It stores file changes more efficiently and ensures file integrity better. If you're interested in knowing the details, the **Git Basics page** has a thorough explanation on how Git works."
>
> (**http://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/**)

To get started using GitHub.com, **you will need an account**. This process is like creating most web software accounts and is **tied to an email account**.



Create your personal account

There were problems creating your account.

Username

RandalsTestStudent ✓

Email Address

RandalsTestStudent@gmail.com ✓

Password

•••••••••

Use at least one lowercase letter, one numeral, and seven characters.

By clicking on "Create an account" below, you are agreeing to the Terms of Service and the Privacy Policy.

Create an account

You'll love GitHub

Unlimited collaborators

Unlimited public repositories

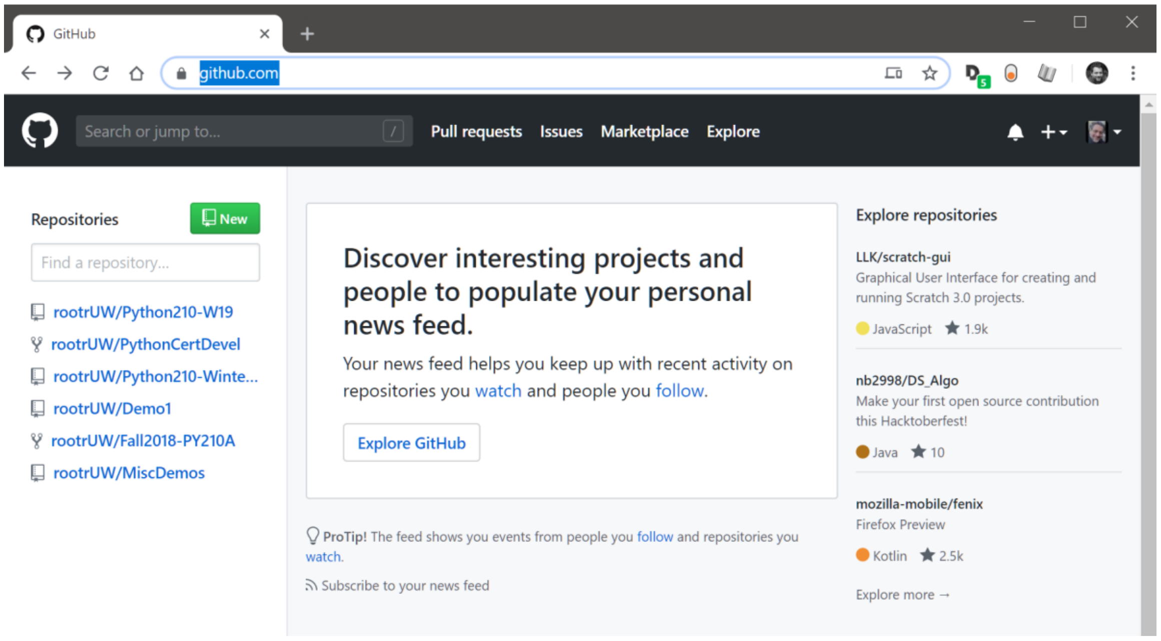
✓ Great communication

✓ Frictionless development

✓ Open source community

Figure 17. Creating an account on GitHub

You can set up multiple user accounts and each account can have multiple repositories. You can think of a **repository** as a set of **shared folders** where your files are stored and **managed through GitHub's web server.**



GitHub

github.com

Search or jump to...

Pull requests Issues Marketplace Explore

Repositories New

Find a repository...

rootrUW/Python210-W19

rootrUW/PythonCertDevel

rootrUW/Python210-Winte...

rootrUW/Demo1

rootrUW/Fall2018-PY210A

rootrUW/MiscDemos

Discover interesting projects and people to populate your personal news feed.

Your news feed helps you keep up with recent activity on repositories you watch and people you follow.

Explore GitHub

ProTip! The feed shows you events from people you follow and repositories you watch.

Subscribe to your news feed

Explore repositories

LLK/scratch-gui

Graphical User Interface for creating and running Scratch 3.0 projects.

JavaScript ★ 1.9k

nb2998/DS_Algo

Make your first open source contribution this Hacktoberfest!

Java ★ 10

mozilla-mobile/fenix

Firefox Preview

Kotlin ★ 2.5k

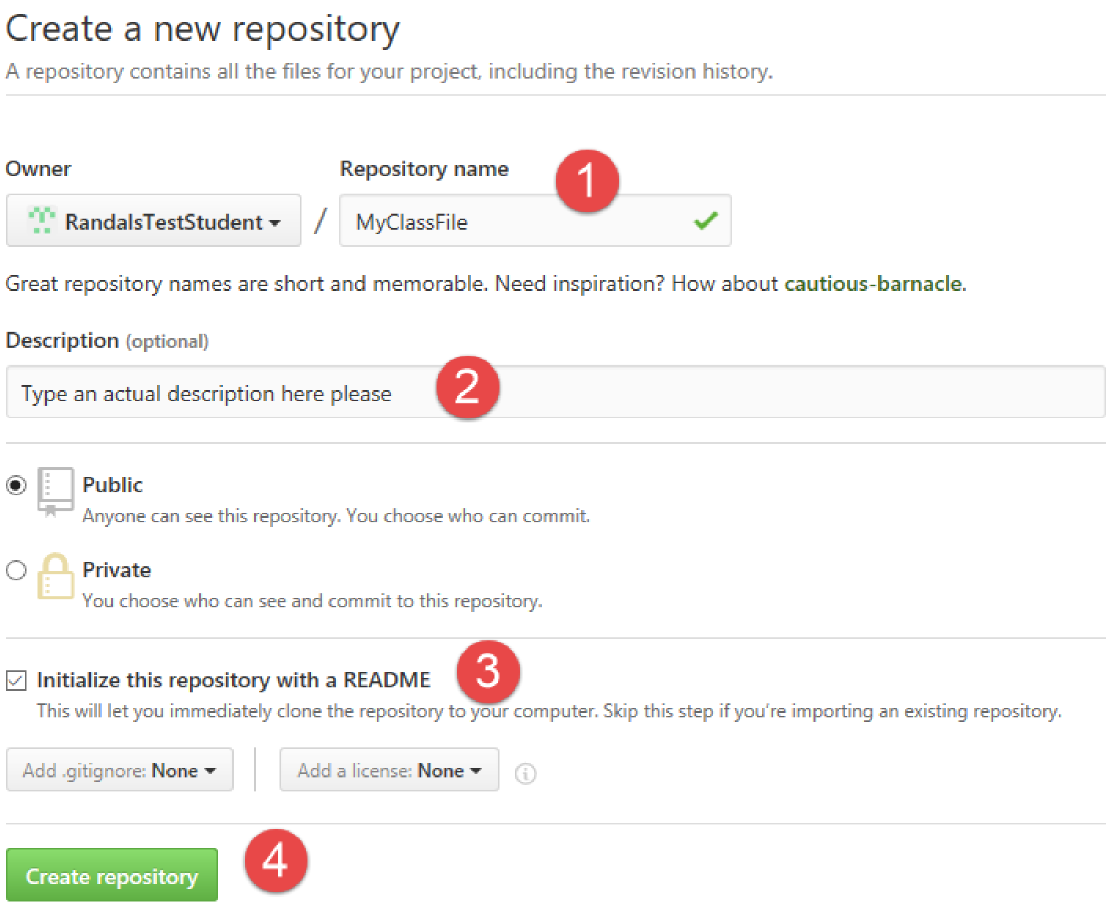
Explore more →

Figure 18. A list of repositories on the GitHub website

To create a repository, you need to **supply its name** and **configure it to be** either private or **public**. If the repository is public, it can easily be seen and shared with other **people.**

You should include a starting readme file in the repository. If you don't you will not see the repository folders materialize until you start loading files into it via the Git command-line too**l.**

Note: In this course, we use the Web UI to make using GitHub more accessible; however, using the Git command tool is the preferred method used in the industry.



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: RandalsTestStudent / Repository name: MyClassFile ✓ (1)

Great repository names are short and memorable. Need inspiration? How about cautious-barnacle.

Description (optional): Type an actual description here please (2)

● Public — Anyone can see this repository. You choose who can commit.

○ Private — You choose who can see and commit to this repository.

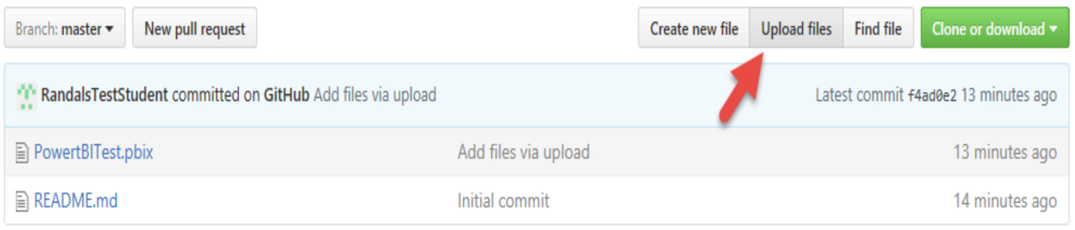
☑ Initialize this repository with a README (3)

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None

Figure 19. Creating a repository using the GitHub.com web interface

Once you create the repository, you upload files to it. These files can be viewed, downloaded, and modified by anyone with a link, so be careful what you upload!



RandalsTestStudent committed on GitHub Add files via upload — Latest commit f4ad0e2 13 minutes ago

PowertBITest.pbix	Add files via upload	13 minutes ago
README.md	Initial commit	14 minutes ago

Figure 20. Using the GitHub.com web interface to upload files

Summary

In this module, we looked at how you use Lists and Dictionaries to store collections of data in memory and text files. We looked at ways you can improve your scripts and make them more professional and saw how to create a GitHub repository.

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 the difference between a List and a Dictionary?
- What is the between an Index and a Key?
- How do you read data from a file into a List?
- How do you read data from a file into a Dictionary?
- What is the programming pattern called “Separations of Concerns?”
- How would you use a function to organize your code?
- Why is a script template useful?
- Why is error handling using Try-Except recommend?
- What is GitHub, and why is it used?

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.