Programming with Python

By Randal Root

Module 08

In this module, you learn to **create scripts using custom classes**. Classes help you **organize functions and data**. There are lots you can add to a class to help you with this organization, but here **we look at the important ones** you should know about, not only **in Python but in other languages too**.

[Classes 2](#_Toc120619563)

[Data Classes vs. Processing Classes 3](#_Toc120619564)

[A Standard Class Pattern 3](#_Toc120619565)

[Fields 3](#_Toc120619566)

[LAB 8-1 4](#_Toc120619567)

[Objects vs. Classes 5](#_Toc120619568)

[Constructors 6](#_Toc120619569)

[The Self Keyword 6](#_Toc120619570)

[LAB 8-2 7](#_Toc120619571)

[Attributes 8](#_Toc120619572)

[LAB 8-3 8](#_Toc120619573)

[Adding Error Handling 9](#_Toc120619574)

[Private Attributes 10](#_Toc120619575)

[Properties 10](#_Toc120619576)

[LAB 8-4 12](#_Toc120619577)

[Methods 12](#_Toc120619578)

[The Built-in "\_\_str\_\_()" Method 14](#_Toc120619579)

[LAB 8-5 16](#_Toc120619580)

[DocStrings 16](#_Toc120619581)

[Type Hints 17](#_Toc120619582)

[GitHub Desktop 18](#_Toc120619583)

[GIT 18](#_Toc120619584)

[GitHub Desktop 19](#_Toc120619585)

[Summary 23](#_Toc120619586)

# Classes

Programs use three basic components: statements, functions, and classes.

Here are three common statements: **data** statements, **processing** statements, and **conditional** statements.

# Statements  
data\_int = 123 # Data assignment statement  
**if** data\_int == 123: # Conditional statement  
 data\_int = 123 + 4 # Processing statement  
print(data\_int) # Calling the print() function

Listing 1

**Statements** are **organized into functions** when a programmer wants to use them many times in a program. For example, these statements have been "**wrapped" in a function called** **orgranize\_statements().**

# Functions  
**def** organize\_statements():  
 data\_int = 123 # Data assignment statement  
 **if** data\_int == 123: # Conditional statement  
 data\_int = 123 + 4 # Processing statement  
 print(data\_int) # Calling the print() function

Listing 2

**Functions** are **organized into classes** when a programmer believes they will be used many times in many programs.

# Classes  
**class** organize\_functions:  
  
 @staticmethod  
 **def** organize\_statements():  
 data\_int = 123 # Data assignment statement  
 **if** data\_int == 123: # Conditional statement  
 data\_int = 123 + 4 # Processing statement  
 print(data\_int) # Calling the print() function Listing 3

Listing 3

***Note:*** *Oddly enough,* ***data and functions in a class have different names****.* ***Variables and constants*** *in a class are****called either fields, attributes, or properties,****while****Functions****in a class****are called Methods****.*

One program can have many statements, many functions, and many classes. It all depends on how complex the program is. **Using functions to organize statements and classes to organize functions makes complex programs easier to create and manage.**

Listing 4 shows examples of statements, functions, and classes in a script. One thing to note is that the first block of statements immediately runs when the script starts. However, the **function and the class code** will **only load into memory**. Their code **is used later** in the script when the functions are called!

# ------------------------------------------------- #  
# Title: Demo01-Statements-Functions-Classes  
# Description: A statements, functions, and classes  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2020,Created Script  
# ------------------------------------------------- #  
  
# Modern programs are usually created using these three components:  
  
# Statements  
data\_int = 123 # Data assignment statement  
**if** data\_int == 123: # Conditional statement  
 data\_int = 123 + 4 # Processing statement  
print(data\_int) # Calling the print() function  
  
  
# Functions  
**def** organize\_statements():  
 data\_int = 123 # Data assignment statement  
 **if** data\_int == 123: # Conditional statement  
 data\_int = 123 + 4 # Processing statement  
 print(data\_int) # Calling the print() function  
  
  
# Classes  
**class** organize\_functions:  
  
 @staticmethod  
 **def** organize\_statements():  
 data\_int = 123 # Data assignment statement  
 **if** data\_int == 123: # Conditional statement  
 data\_int = 123 + 4 # Processing statement  
 print(data\_int) # Calling the print() function  
  
  
# ---------------------------  
  
input("Press enter to call the function in the main body of the script")  
organize\_statements()  
  
# Call the function within the class (aka. Method)  
input("Press enter to call the class function (aka. Method)")  
organize\_functions.organize\_statements()

Listing 4

# Data Classes vs. Processing Classes

**Classes are designed to focus on either data, processing, or interaction (input and output)**. For example, a **developer might create one class**for processing data to and from a file, **naming it something like "FileManager." They also might create another**class for managing the customer data, **naming it something like "Customer."**The focus of the FileManager" class would be to perform a set of actions, while the focus of the "Customer" class would be to organize data about a customer.

# A Standard Class Pattern

Classes **typically have Fields, Constructors, Properties, and Methods**. Like scripts, class code follows a **general design pattern in most of the languages.** To learn more about how classes work, let's look at each of these components, starting with fields.

## Fields

Fields are data members of a class. You create them using variables and constants. Listing 5 shows an **example** of creating three constant fields describing information about a program's data file.

# ------------------------------------------------- #  
# Title: Demo02-Fields  
# Description: A simple example of a class field  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
#--- Make the class ---  
**class** FileInfo:  
 # --Fields--  
 FILE\_FOLDER = os.getcwd() # Python's get the current working directory functionFILE\_NAME = **'program\_data.txt'** FULL\_PATH = FILE\_FOLDER + **'/'** + FILE\_NAME# End of class  
  
  
# --- Use the class ----  
**try**:  
 file\_obj = open(FileInfo.FULL\_PATH, **'r'**)  
**except** FileNotFoundError **as** e:  
 error\_msg = **'Please check that the '** + FileInfo.FILE\_NAME  
 error\_msg += **'file exists in the '** + FileInfo.FILE\_FOLDER  
 print(error\_msg)

Listing 5

Table

Description automatically generated

Figure: The results of listing 5 in a PyCharm window

Graphical user interface, text

Description automatically generated

Figure: The results of listing 5 in a terminal window

**Important:** Remember that the current working directory in PyCharm and in the Command window may be different since the current working directory is based on the folder your program is using when its running!

# LAB 8-1

In this lab, you create a simple class to organize data about a file.

1. **Use** the code in Listing 5 to create the FileInfo class. Try typing the code for practice first, but you can copy and paste if you have problems.

2. **Test** the code in PyCharm and in a terminal window.

3. **Note** the difference between the current working directory's location. The figure above shows an example from my computer, but yours may have a different path.

## Objects vs. Classes

When the class's code loads into memory, **you either use the class's code directly or indirectly by making a copy of the class’s code**. To use the code indirectly you create an "object instance." An object instance (or just object) can be thought of as an independent copy of the classes code.

For example, let's say we have a customer class with two fields (variables) called Id and Name.

class Customer:

Id

Name

To **directly** use the Customer's fields, **use code like this**:

Customer.Id = 100

Customer.Name = "Bob Smith"

Customer.Id = 200

Customer.Name = "Sue Jones"

The problem with this approch is that changing the id or name overwrites the previous data. However, **you can use an object instance of the class to avoid this.**

**You create and use an object instance** **a class** **like this**:

objC = Customer()

objC.Id = 100

objC.Name = "Bob Smith"

One advantage of using a copy of the code is that you **can have multiple object instances, each with a different address in memory.** The data in each object instance is separate from all other objects. This is useful since **each object can hold different data for each customer**.

Diagram

Description automatically generated

Figure: A simple example of objects made from a customer class

**In general**, use a class **directly if it does not unique data or its focus is on processing data,** and **use object instances of the class if its focus is on storing unique data (that way each copy has its own data)**. This generalization may not always be true, but it is enough to provide a good starting point.

***Notes****:*

* *The address numbers are just for illustration. I am using* ***made-up address numbers*** *to indicate different locations in a computer's memory.*
* *The* ***concepts*** *in this module are advanced and may* ***take time to understand****. Be patient; they become clearer as you work with class and functions!*
* *This basic explanation of the subject is purposely simplified to help students get a general idea of the topic.* ***More technical explanations are covered in higher level courses****.*

## Constructors

**When you create an object instance** from a class, **you call the class's name as if it were a function**. Using the class's name like a function automatically calls the class's constructor method.

Constructors are **special methods** (functions) that **automatically run when you create an object from a class**. **Constructors set initial data values**, which is why they are often called an "**initializer.**" In fact, Python's constructors are always **named** **"\_\_init\_\_".**

**Constructor methods use parameters to capture the initial values as shown in l**isting x.

# ------------------------------------------------- #  
# Title: Demo03-Constructors  
# Description: A class with a constructor  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
**class** Person():  
 # --Fields--  
 first\_name\_str = **""** # -- Constructor --  
 **def** \_\_init\_\_(self, **first\_name=''**): # The default is an empty string  
 self.first\_name\_str = first\_name  
  
# --End of class--  
  
# --- Use the class ----  
objP1 = Person() # With no arguments  
objP1.first\_name\_str = **"Bob"**objP2 = Person(first\_name=**"Sue"**) # With one parameter and argument  
  
print(objP1.first\_name\_str) # Will be empty because there was no argument  
print(**"-------------"**)  
print(objP2.first\_name\_str) # Will have first name of Sue

Listing 6

***Note:******constructors are a specialized function****, so you use them as a function by passing arguments into the parameters. However,* ***remember****, they* ***only run once, only when a new object instance of a class is created****!*

## The Self Keyword

You probably noticed the use of the **keyword "self"** in the constructor method. This keyword is used to **refer to data or functions found in an object instance,** and **not directly in the class**.

To understand the "self" keyword, remember that the **class’s code immediately loads into memory** when your script starts running and then waits to be used, either directly or through an object instance.

The following code first loads the Person class into memory and then an creates object objP1 and objP2 from the Person class.

**class** Person(): # 1. Load this class into memory  
 # --Fields--  
 first\_name\_str = **""** # -- Constructor --  
 **def** \_\_init\_\_(self, **first\_name=''**):

self.first\_name\_str = first\_name

objP1 = Person("Bob") # 2. Use the class  
objP2 = Person("Sue") # 2. Use the class

**While the class's code only loads into** memory **once**, it can have many **object instances of a class**, each representing a "copy" of the classes code! **In Python, you identify which copy is referenced using the pronoun "self." Just as two people conversing might each refer to themself!**

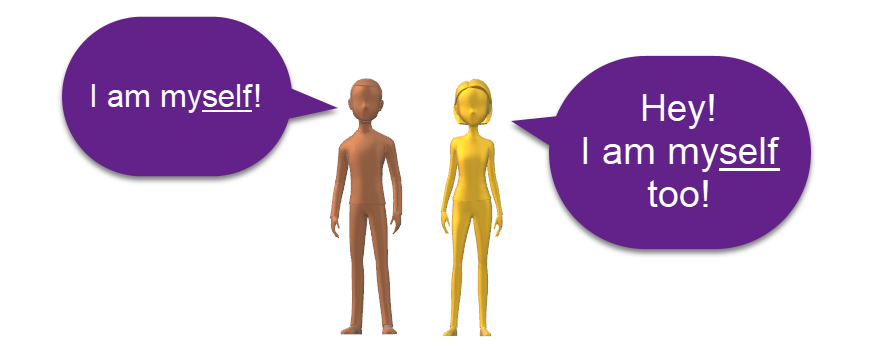


Figure: Using the "myself' pronoun is contextual

Other languages use the word "this" instead, but in Python it's always "self." The people who made the Python language made a **rule** to **include a parameter called "self" in each method meant to be used from an object instance**.

**Notes:**

* Oddly, you **cannot not pass an argument to the "self" parameter**, it is automatically filed with the object's referenced location in memory.
* When a **method is used directly** from the class, you **leave out the "self" parameter and** mark the method with the **@staticmethod** decorator as we have seen in earlier modules.

# LAB 8-2

In this lab, you **add** a **constructor** to the Person class you made in Lab 8-1.

1. **Modify** the Person class to include a constructor using the code in Listing 6. Try typing the code for practice first, but you can copy and paste if you have problems.

2. **Add** a last\_name parameter in addition to the first\_name parameter to the class's constructor.

3. **Use** these parameters to set a first\_name\_str and last\_name\_str fields.

4. **Test** the code by creating an object instance, setting the values of the fields, then printing the values of the fields.

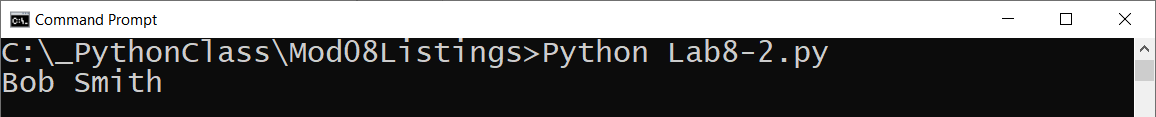


Figure: The results of Lab 8-2

**Important:** We covered Fields because it is a common option in programming languages. However, **Python does not use them this way.** Instead, we will use Attributes to hold the data in our object instances.

## Attributes

Instead of standard fields**, Python uses "virtual" fields** to hold internal object data. These virtual fields, **called Attributes,** can be created by declaring the variable in the constructor as show here:

# ------------------------------------------------- #  
# Title: Demo04-Attributes.py  
# Description: A class with an attribute  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
**class** Person(object):  
 # --Fields--  
 # first\_name\_str = "" <- Delete this. Python does not use fields for instance data  
  
 # -- Constructor --  
 **def** \_\_init\_\_(self, first\_name):  
 # Attributes  
 self.first\_name\_str = first\_name # this is a attribute (virtual field)

# --End of class--  
  
# --- Use the class ----  
objP1 = Person(**"Bob"**)  
print(objP1.first\_name\_str) # Using the empty explicit field

Listing 7

When the code runs, it still acts as if the field existed, but it is now using an attribute or "virtual field."

Text

Description automatically generated

Figure: The results of Listing 7.

# LAB 8-3

In this lab, you modify the constructor of the Person class you made in Lab 8-2 to use **attributes** instead of fields.

1. **Delete** the first\_name\_str and last\_name\_str fields since we are no longer using them.
2. **Verify** thatthe constructor **sets** the first\_name\_str and last\_name\_str attributes with parameter data as it did before. You do not need to make any other changes.
3. **Test** the code still works by creating an object instance, setting the values of the attributes, and printing the attribute values.

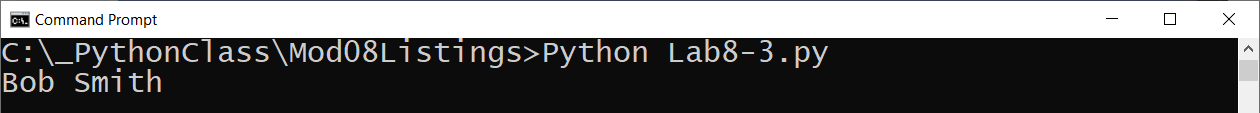


Figure: The results of Lab 8-3.

**Note:** Try typing the code for practice first, but you can copy and paste if you have problems.

## Adding Error Handling

Whenever you work with data there is a chance that a human can enter incorrect values. As such, it is important to add validation code to keep your data "clean." For example, we can add validation code to the constructor to stop a string of number being used for a name using Python's isnumeric() function. I am also adding formatting using the title() function to make the data more consistent.

# ------------------------------------------------- #  
# Title: Demo05- Validation Code  
# Description: A class with an attribute  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
**class** Person(object):  
  
 # -- Constructor --  
 **def** \_\_init\_\_(self, first\_name):  
 # Attributes  
 self.first\_name\_str = **''** # declare the attributes first  
 **try**:  
 **if** str(first\_name).isnumeric(): # test that parameter data is valid  
 **raise** ValueError  
 **else**:  
 self.first\_name\_str = first\_name.title() # set and format attribute  
 **except** ValueError **as** v:  
 print(v)  
 print(**'Names must include at least one alpha character!'**)  
  
# --End of class--  
  
# --- Use the class ----  
objP1 = Person(**"bob"**)  
print(objP1.first\_name\_str)  
objP1 = Person(**"123"**)  
print(objP1.first\_name\_str)  
  
# However, it does not stop us from using numbers later(an issue we will fix shortly!)  
objP1.first\_name\_str = **'123'**print(objP1.first\_name\_str)

Listing 8. Adding validation code

Graphical user interface, text

Description automatically generated

Figure, the results of listing 8.

## Private Attributes

It is a best practice to mark an attribute as private whenever its values are controlled by a method, like our constructor.

**def** \_\_init\_\_(self, first\_name):  
 self.\_\_first\_name = first\_name

**Note, I have added (2) underscores before the attribute's name.** This marks the attribute as "***private***" and indicates that developers should not access the attribute from code outside of the class. Instead, they should use a method to work with the attribute's values.

Oddly, **Python does not vigorously enforce this privacy, like most other languages**. So, it is up to you and your coworkers to respect the privacy of an attribute. The **recommended** way of doing this is by **using** the **property functions** to get and set attribute data indirectly.

## Properties

Properties are **special functions used to manage attribute data**. You **typically create** two properties for each attribute, **one for "getting" data and one for "setting" data**. In fact, these types of function are often called "Getters" and "Setters" or "Accessors" and "Mutators."

**Getter** property functions let you access the data and optionally add **formatting** code. Python **uses the @property decorator to indicate a fuction is a "getter"** like this:

@property # (getter or accessor)  
**def** first\_name(self): # This is the name used to work with the data  
 **return** str(self.\_\_first\_name\_str).title() # Formatting data in title case

**Setter** property functions let you add code for both **validation and error handling**. If a value passed into the Properties parameter is valid, then it is assigned to the attribute. You **create a setter** like any other function, but it **must include** the @name\_of\_property**.setter decorator.**

@first\_name.setter *# (setter or mutator)* **def** first\_name(self, value): *# make name match the getter!* **if** str(value).isnumeric() == **False**:  
 self.\_\_first\_name\_str = value  
 **else**:  
 **raise** Exception(**"Names cannot be numbers"**)

**Note**, that the **property** is now responsible for **managing** the double underscore (private) **attribute**!

**Important:** The **Setter** **decorator and function name must match the name of the Getter** for them to be a "getter and setter" pair! This odd and inconsistent syntax is unique to Python.

Next, you **modify the constructor to use the properties instead of the attributes**. This change forces you code to use the validation logic within those properties.

*# -- Constructor --* **def** \_\_init\_\_(self, first\_name):  
 *# Attributes  
 #* self.\_\_first\_name\_str = first\_name # Don’t use attributes in the constructor!

self.first\_name = first\_name # Use Properties instead!

Here is a listing of our updated code:

# ------------------------------------------------- #  
# Title: Listing06 - Properties  
# Description: A class with an attribute  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
# --- Make the class ---  
**class** Person:  
 # --Fields--  
 #strFirstName = ""  
  
 # -- Constructor --  
 **def** \_\_init\_\_(self, first\_name):  
 # Use a property to set the attribute  
 self.first\_name = first\_name  
 # The property and parameter can have the same name due to the self context  
  
 # -- Properties --  
 @property # You don't use for the getter's directive!  
 **def** first\_name(self): # (getter or accessor)  
 **return** str(self.\_\_first\_name).title() # Format attribute as Title case  
  
 @first\_name.setter # The @NAME.setter must match the getter's name!  
 **def** first\_name(self, value): # (setter or mutator)  
 **if** str(value).isnumeric() == **False**:  
 self.\_\_first\_name = value  
 **else**:  
 **raise** Exception(**"Names cannot be numbers"**)  
  
# --End of class--  
  
# --- Use the class ----  
objP1 = Person(**"Bob"**)  
  
objP1.first\_name = **'robert'** # using the Setter property  
print(objP1.first\_name) # using the Getter property  
  
**try**:  
 objP1.first\_name = **'123'** # testing that a number causes a validation ERROR  
**except** Exception **as** e:  
 print(e)

Listing 9

Text

Description automatically generated

Figure: The results of listing 9.

**Important:** **We don’t use fields or attributes in Python constructors.** If we do, the property’s validation logic won’t be used when the object is created.

# LAB 8-4

In this lab, you **modify** the Person class you made in Lab 8-3 to use a getter and setter property function for the first and last name **attributes**. You will also make the attributes private then add validation or formatting code to each property function.

1. **Create** a getter and setter Property function called first\_name to manage the private \_\_first\_name attribute. Make sure to use formatting and validation as shown in listing 9.
2. **Create** a getter and setter Property function called last\_name to manage the private \_\_last\_name attribute. Make sure to use formatting and validation as shown in listing 9.
3. **Modify** the constructor to use the property names **self.first\_name** and **self.last\_name**, instead of the attribute names.
4. **Test** the code by creating an object instance, setting the properties, then printing the values of the first and last name properties.
5. **Try** entering a number for the first and last name to **test** that an error occurs.

Text

Description automatically generated

Figure: The results of Lab 8-4.

**Note:** Try typing the code for practice first, but you can copy and paste the code from listing 9 if you have problems.

## Methods

While functions that manage attribute data are called properties, **other functions inside of a class are called Methods**. Methods are just like normal functions, letting you organize statements into named groups, but are part of a class's code instead of being just part of a script's general code.

**Most classes** include a **method that returns** some or all the **class's data as a string**. Here is an example of how that might work using the first and last name in a comma separated value format (which is commonly used throughout the IT industry.)

# ------------------------------------------------- #  
# Title: Demo-07-Methods  
# Description: A class methods  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
# --- Make the class ---  
**class** Person:  
  
 # -- Constructor --  
 **def** \_\_init\_\_(self, first\_name, last\_name):  
 # Use a property to set the attribute  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
 # -- Properties --  
 @property # You don't use for the getter's directive!  
 **def** first\_name(self): # (getter or accessor)  
 **return** str(self.\_\_first\_name).title() # Format attribute as Title case  
  
 @first\_name.setter # The @NAME.setter must match the getter's name!  
 **def** first\_name(self, value): # (setter or mutator)  
 **if** str(value).isnumeric() == **False**:  
 self.\_\_first\_name = value  
 **else**:  
 **raise** Exception(**"Names cannot be numbers"**)  
  
 @property # You don't use for the getter's directive!  
 **def** last\_name(self): # (getter or accessor)  
 **return** str(self.\_\_last\_name).title() # Format attribute as Title case  
  
 @last\_name.setter # The @NAME.setter must match the getter's name!  
 **def** last\_name(self, value): # (setter or mutator)  
 **if** str(value).isnumeric() == **False**:  
 self.\_\_last\_name = value  
 **else**:  
 **raise** Exception(**"Names cannot be numbers"**)  
  
 **def** to\_string(self):  
 """ Returns object data in a comma separated string of values  
  
 **:return**: (string) CSV data  
 """  
 object\_data\_csv = self.first\_name + **','** + self.last\_name  
 **return** object\_data\_csv  
  
# --End of class--  
  
# --- Use the class ----  
objP1 = Person(**"bob"**, **"smith"**)  
print(objP1.to\_string())

Listing 10

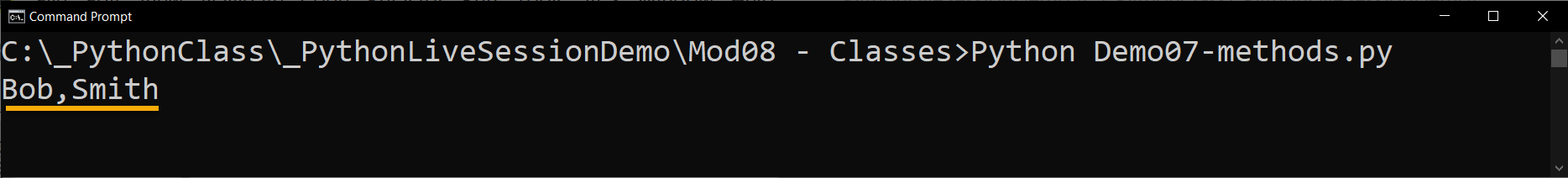


Figure: The results of Listing 10.

## The Built-in "\_\_str\_\_()" Method

Python has a built-in method called the "\_\_str\_\_()" method for returning the class's data as a string. This method exists in every class, even if you do not add it to the class yourself!

**Python's invisible "\_\_str\_\_()" method defaults** to **returning the memory address of the class's object instance (**whichyou may have seen this going through the course.) Here is an example:

# ---------------------------------------------------------------- #  
# Title: Demo-08-Overriding the built-in \_\_str \_\_() method  
# Description: A overriding the \_\_str\_\_() method  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ---------------------------------------------------------------- #  
  
**class** Demo1:  
 var1 = **"Some Data"**obj1 = Demo1() # This object uses the default \_\_str\_\_() method  
print(obj1.\_\_str\_\_())  
  
**class** Demo2:  
 var1 = **"Some Data"  
 def** \_\_str\_\_(self):  
 **return '<'** + self.var1 + **'>'**obj2 = Demo2() # object uses the overridden \_\_str\_\_() method  
s = str(obj2)  
print(str(obj2)) # str() automatically calls the \_\_str\_\_() method  
print(obj2) # print() function automatically calls the \_\_str\_\_() method  
print(obj2.\_\_str\_\_()) # but you can also call the \_\_str\_\_() method directly  
  
# Python's List class overrides the \_\_str\_\_() method too, but it is not very useful.  
data\_lst = [1,2,3]  
print(data\_lst)

Listing 11

Text

Description automatically generated

Figure: The results of listing 11.

In our Person example, we can **override the \_\_str\_\_() method to use the same code as our to\_string() method**, by calling our method from the Python built-in method like this:

# ------------------------------------------------- #  
# Title: Demo-09-Reusing method code  
# Description: A class methods  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
# --- Make the class ---  
**class** Person:  
 # -- Constructor --  
 **def** \_\_init\_\_(self, first\_name, last\_name):  
 # Use a property to set the attribute  
 self.first\_name = first\_name  
 self.last\_name = last\_name  
  
 # -- Properties --  
 @property # You don't use for the getter's directive!  
 **def** first\_name(self): # (getter or accessor)  
 **return** str(self.\_\_first\_name).title() # Format attribute as Title case  
  
 @first\_name.setter # The @NAME.setter must match the getter's name!  
 **def** first\_name(self, value): # (setter or mutator)  
 **if** str(value).isnumeric() == **False**:  
 self.\_\_first\_name = value  
 **else**:  
 **raise** Exception(**"Names cannot be numbers"**)  
  
 @property # You don't use for the getter's directive!  
 **def** last\_name(self): # (getter or accessor)  
 **return** str(self.\_\_last\_name).title() # Format attribute as Title case  
  
 @last\_name.setter # The @NAME.setter must match the getter's name!  
 **def** last\_name(self, value): # (setter or mutator)  
 **if** str(value).isnumeric() == **False**:  
 self.\_\_last\_name = value  
 **else**:  
 **raise** Exception(**"Names cannot be numbers"**)  
  
 **def** to\_string(self):  
 """ Returns object data in a comma separated string of values  
  
 **:return**: (string) CSV data  
 """  
 object\_data\_csv = self.first\_name + **','** + self.last\_name  
 **return** object\_data\_csv  
  
 **def** \_\_str\_\_(self):  
 """ Overrides Python's built-in method to  
 return object data in a comma separated string of values  
  
 **:return**: (string) CSV data  
 """  
 **return** self.to\_string()  
# --End of class--  
  
# --- Use the class ----  
objP1 = Person(**"bob"**, **"smith"**)  
print(objP1.to\_string())  
print(objP1.\_\_str\_\_())  
print(objP1)

Listing 11

Text

Description automatically generated

Figure: The result of listing 11.

***Note:*** *There are many automatic and invisible built-in methods included in every class. We will talk more about them and how they work module 9.*

# LAB 8-5

In this lab, you **modify** the Person class you made in Lab 8-4 to included a to\_string() method and an overridden "\_\_str\_\_()" method.

1. **Add** a new method to named to\_string() that returns both the first\_name and last\_name with a comma separator.
2. **Override** the **"**\_\_str\_\_()" built-in method to use your to\_string() method.
3. **Test** the code by creating an object instance, setting the properties, then using print() to run the automatically run the object's "\_\_str\_\_()" method.

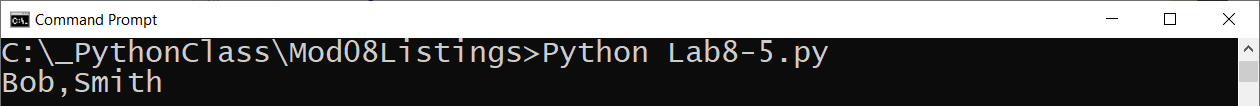


Figure: The results of Lab 8-5

**Note:** Try typing the code for practice first, but you can copy and paste the code from listing 11 if you have problems.

# DocStrings

Just as we did with functions, we should include a docstring for our classes. It can be helpful if **developers include additional notes in a docstring**. When they do, Integrated development environments like **PyCharm can display tooltips** showing the developer's notes (**use ctrl + q to activate** this option in PyCharm). You can **also** show the "DocString" using the built-in and inherited **\_\_doc\_\_ property** (Figure 16).

# ------------------------------------------------- #  
# Title: Demo-10 Class Docstring  
# Description: A class with a docstring  
# ChangeLog: (Who, When, What)  
# RRoot,1.1.2030,Created Script  
# ------------------------------------------------- #  
  
**class** Person:  
 """Stores data about a person:  
  
 properties:  
 first\_name: (string) with the person's first name  
 methods:  
 static: get\_object\_count() -> int with number of object instances created  
 changeLog: (When,Who,What)  
 RRoot,1.1.2030,Created Class  
 """  
 # pass is used as a temporary placeholder for code that will be added  
 **pass** # **TODO: Add code to the class**# --- Use the class ----  
  
  
print(Person.\_\_doc\_\_) # You don't use () for this one, because it actually a field  
print(list.\_\_doc\_\_) # Here is one from the Python programmers.

Listing 12

Text

Description automatically generated

Figure: The results of listing 12.

# Type Hints

Another documentation feature recently added to Python is "type hints." These identify what type of data is expected in a parameter. Type hints can help other developers understand what is wanted and some IDEs provided tips to reduce mistyping.

Graphical user interface, text, application, email

Description automatically generated

Figure: Type Hints in PyCharm IDE

**Note:** Unlike other languages, the Python runtime does not yet enforce parameter types, so incorrect argument types can still be passed into to a parameter.

# GitHub Desktop

In this course, we've used GitHub's website through a web browser, but **often developers work with GitHub differently**. Instead of browser, they work with GitHub **on their local computers using either with a command prompt application or a desktop application**. In both cases, the communication between the GitHub website and the local computer is **handled by a program called "Git**."

## GIT

The Git software **manages versions of one or more files**. It allows you to **make a clone (copy)** of a file, then **make changes** to clone, and **save it as a new version of the same file**. All while **maintaining a copy of the original** version. **In addition to managing the cloned file on your computer, by default, Git uses the GitHub website to store backup files in the "Cloud**."

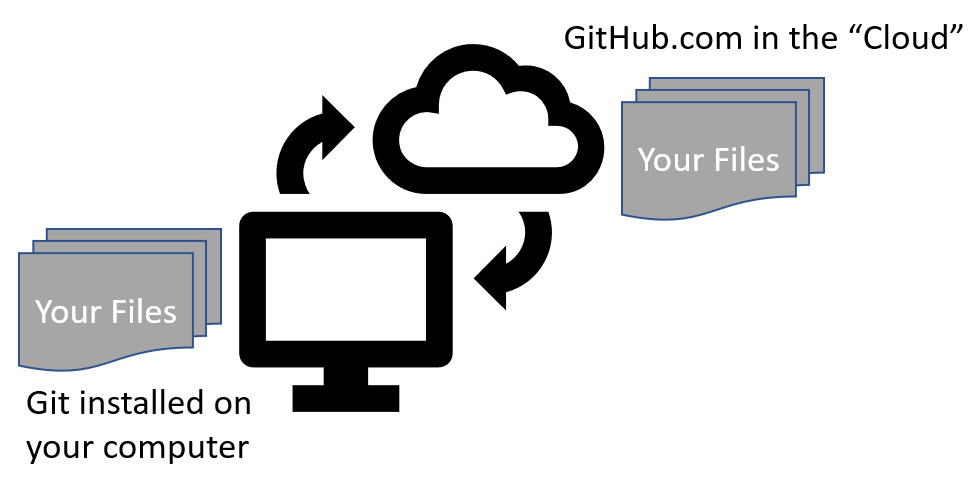


Figure 17. Git and GitHub work together to manage files

## GitHub Desktop

In most organizations, developers use command shell to interact with the website from their local computers, but in this module, we start with something more visual; GitHub Desktop. GitHub Desktop is a **free application** you can install on both **Windows and Mac** OS.

To install GitHub Desktop, you **navigate to its download page** (<https://desktop.github.com/>), select your operating system, and download the installation file. Then start the installation, which is quick and straightforward!



Figure 18. The GitHub Desktop download page

Once it installs, it opens the **application and asks you to log in to your GitHub website account**. You can always change that account later using the File -> Options menu, then the Sign in and Sign buttons (Figure 19).

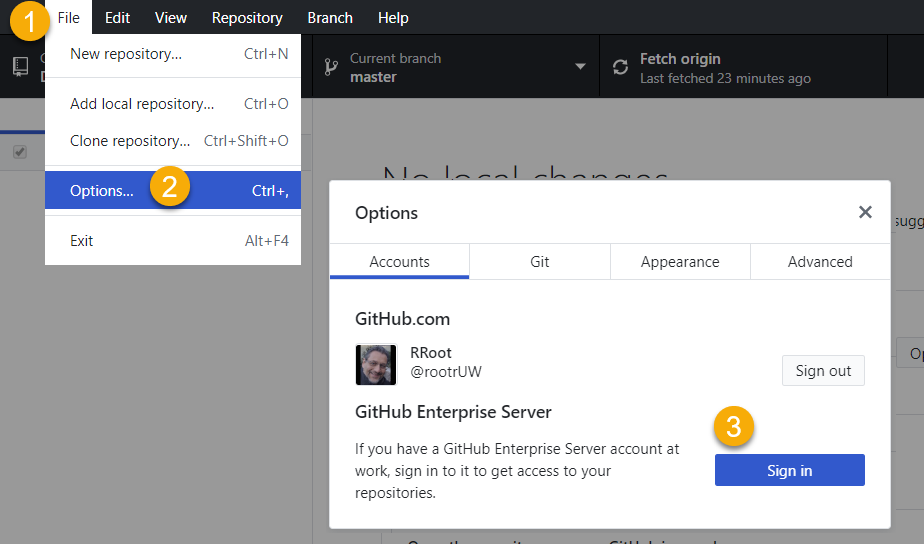


Figure 19. Changing the GitHub account option

Once you are signed in, you can see a listing of your GitHub repositories on the right-hand side of the UI. Click on a repository, then **click the "Clone" button to download a copy of your repository's files** (Figure 20).This button launches the "Clone a repository" dialog box.

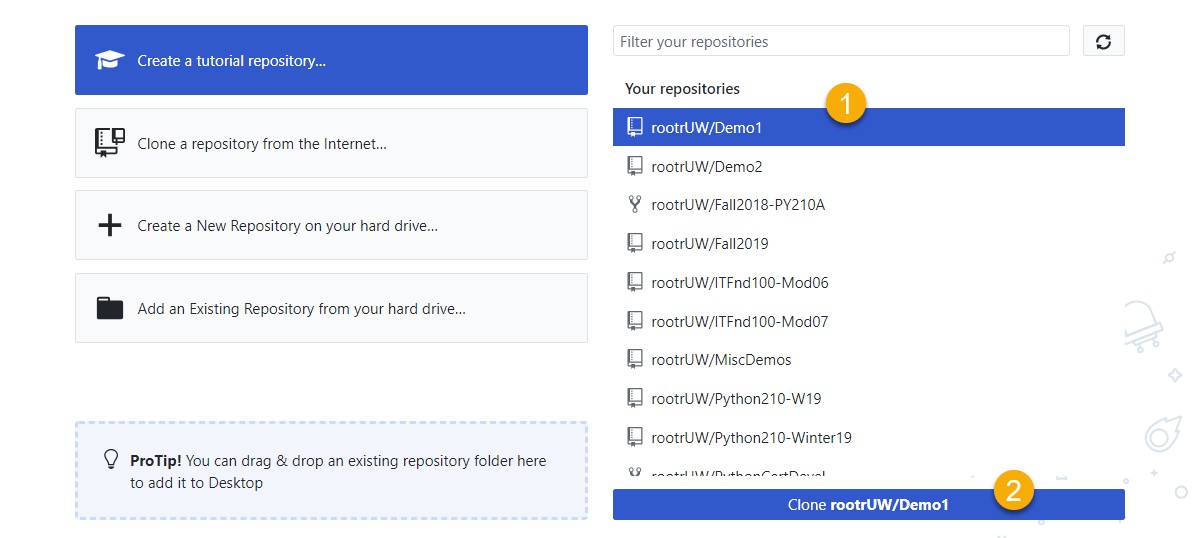


Figure 20. The repository listing

In the "Clone a repository" dialog box, **verify** the GitHub.com **repository and the local folder** where the files are copied to, **before clicking the "Clone" button** (Figure 21). This button starts the download process.

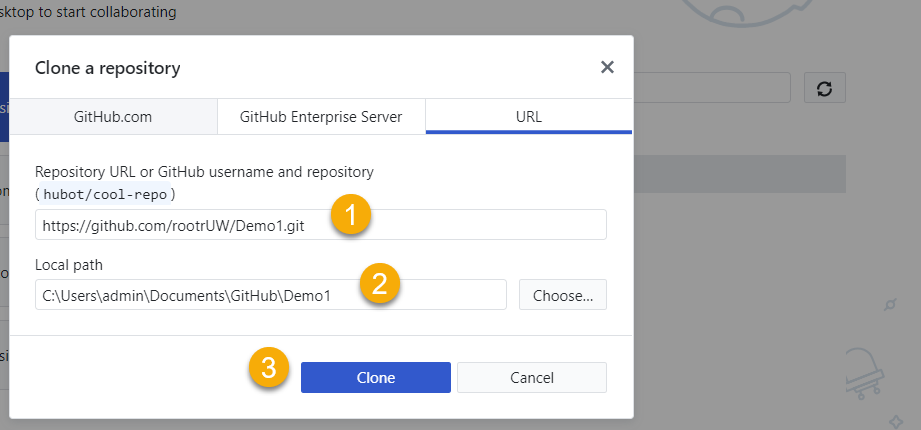


Figure 21. The "Clone a repository" dialog box

Once the files download into the folder, you can **open the folder and see the copied files**. GitHub Desktop offers a convenient button to open the folder, but of course, you can always use Windows Explorer or Finder to locate the files (Figure 22).

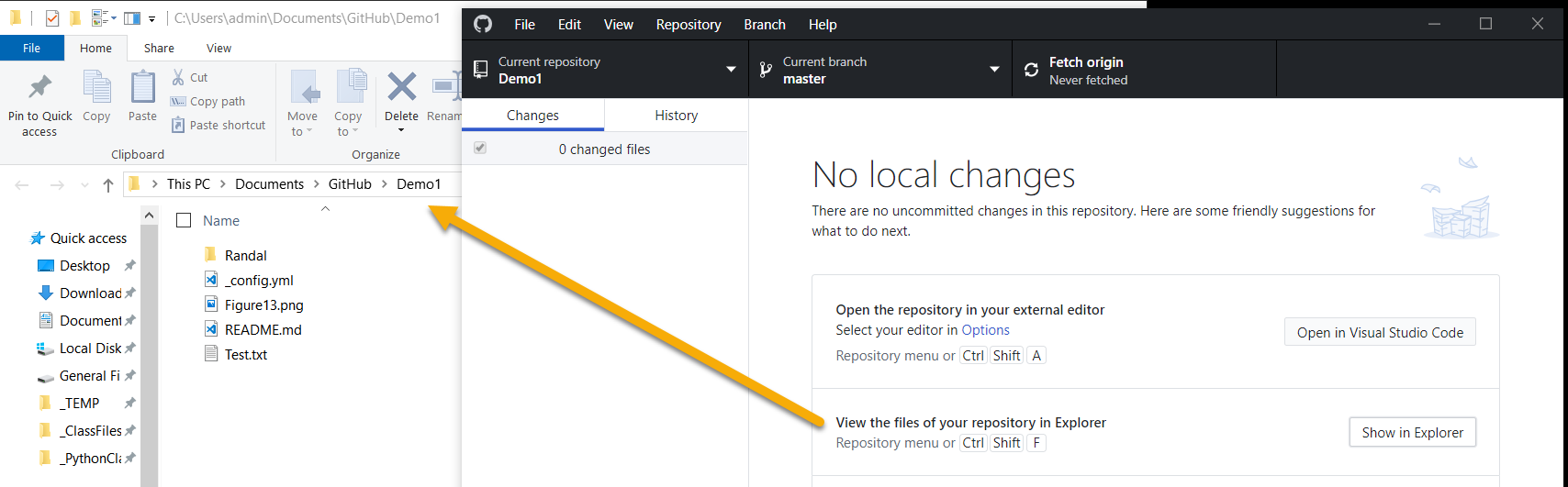


Figure 22. Viewing the local files

It may not look like it, but **Git is now managing the folder**! To see it in action, open a file, make a change to it, the go back to GitHub Desktop. The figure below shows some **text I added to a text file**. Figure 24 shows how **GitHub Desktop displays the previous text in red and the new text in green**.

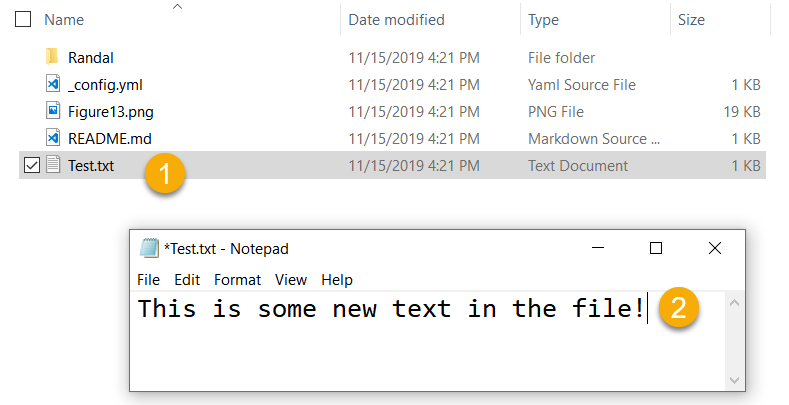


Figure 23. Changing text in a Git managed file

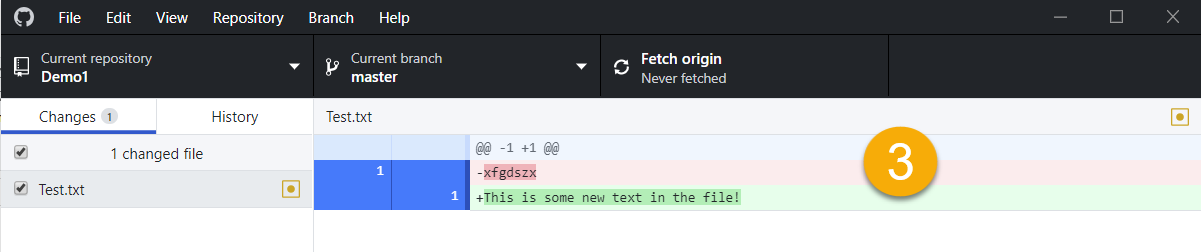


Figure 24. Reviewing the changes to a file in GitHub Desktop

If you want to **upload** your changes **to the Main folder of your GitHub repository**, click the **"Commit to Main" button**, then click the **"Push origin" button** when it appears (Figure 25).

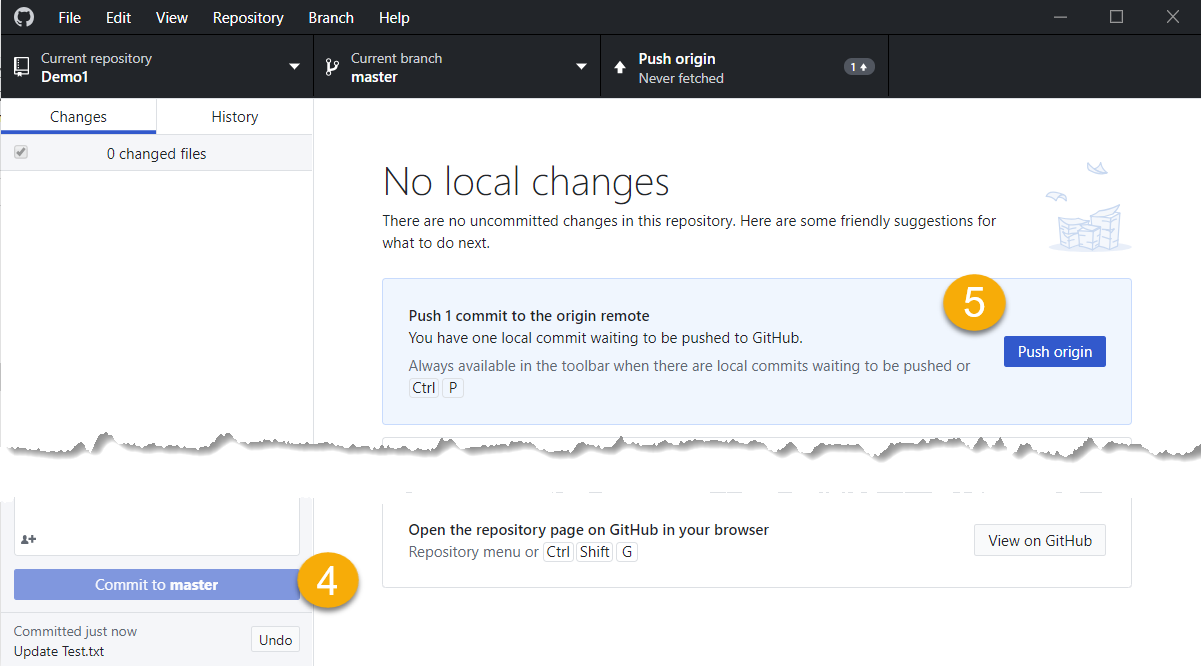


Figure 25. Uploading the changes to GitHub

Once the upload is complete, you can **navigate to GitHub and verify** that both the local and website version of the file are the same.

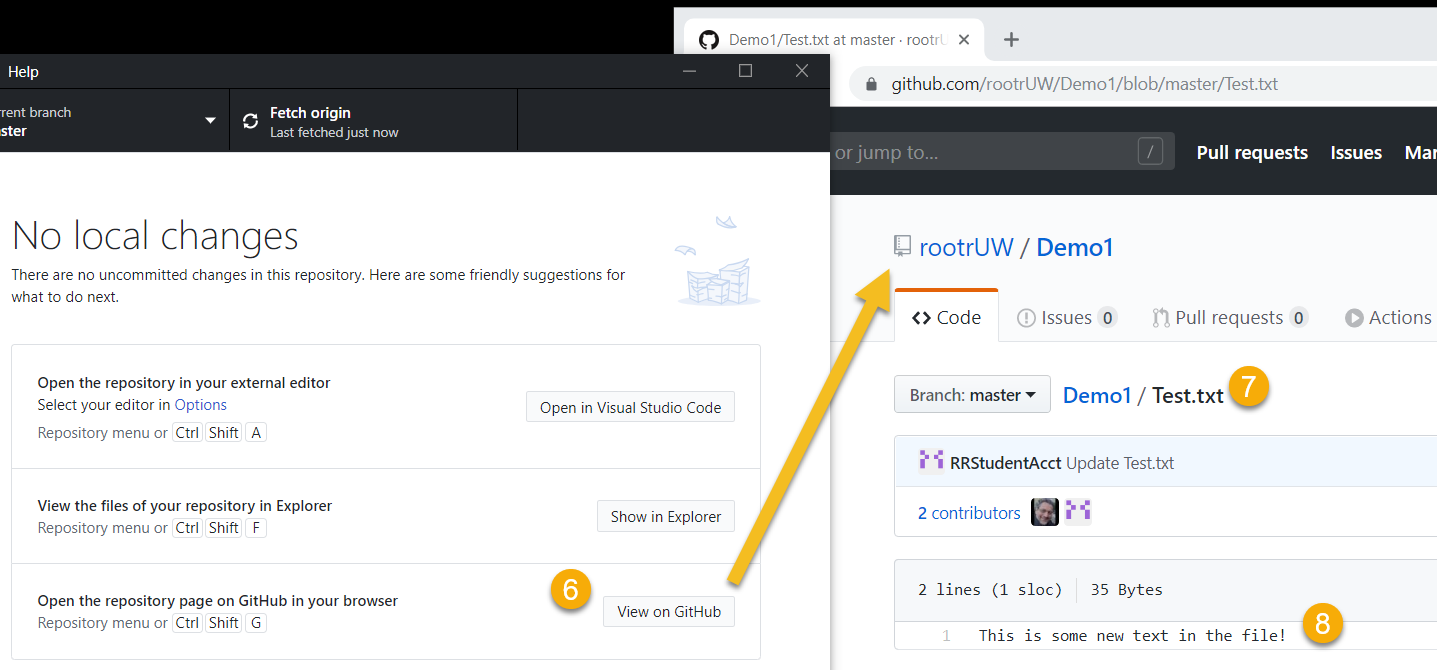


Figure 26. Viewing GitHub's website to verify the files are uploaded.

After that test is complete, you can start exploring the features of GitHub Desktop. You can **learn more about the application on this GitHub website** (<https://help.github.com/en/desktop>).

# Summary

In this module, we looked at how to use custom functions and try-except blocks to organize file management code and provided custom error handling. We also looked at ways you can improve your GitHub webpages 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 the difference between a class and the objects made from a class?
* What are the components that make up the standard pattern of a class?
* What is the purpose of a class constructor?
* When do you use the keyword "self?"
* When do you use the keyword "@staticmethod?"
* How are fields and attributes and property functions related?
* What is the difference between a property and a method?
* Why do you include a docstring in a class?
* What is the difference between Git and GitHub?
* What is GitHub Desktop?

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.