PROGRAMMING WITH PYTHON

By Randal Root

Module 08

In this module, you learn about **creating scripts using custom classes**. Classes are **used to organize function or data** and have **several advanced features** to help them do so. Here, **we look at the important ones** you should know about, not just for programming **in Python, but for other languages** as well.

Classes

Classes are a way of grouping data and functions, and **most classes are designed to focus on either data or processing**. For example, a **developer might create one class** for processing data to and from a file**, called something like "FileManager," and another** for managing the data to be processed **called 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.

Data in a class is defined using **variables or constants**. However, when these are in a class, they are **called Fields**. Any **functions** you have in a class **are called Methods**.

Objects vs. Classes

When the class's code loads into memory, **you either use that code directly or indirectly**. To use the class's code **directly,** you use commands **like the following** pseudo-code:

```
Customer.Id = 100
Customer.Name = "Bob Smith"
```

To use the code **indirectly,** you **create an object instance** of the class and use the object's variable with commands **like the following** pseudo-code:

```
objC = Customer()
objC.Id = 100
objC.Name = "Bob Smith"
```

One advantage of using the code **indirectly** is that you **can have multiple object instances, each with a different address in memory** (Figure 1.) The data for each instance is kept separate for each object, and **each object would hold data about a different customer**.

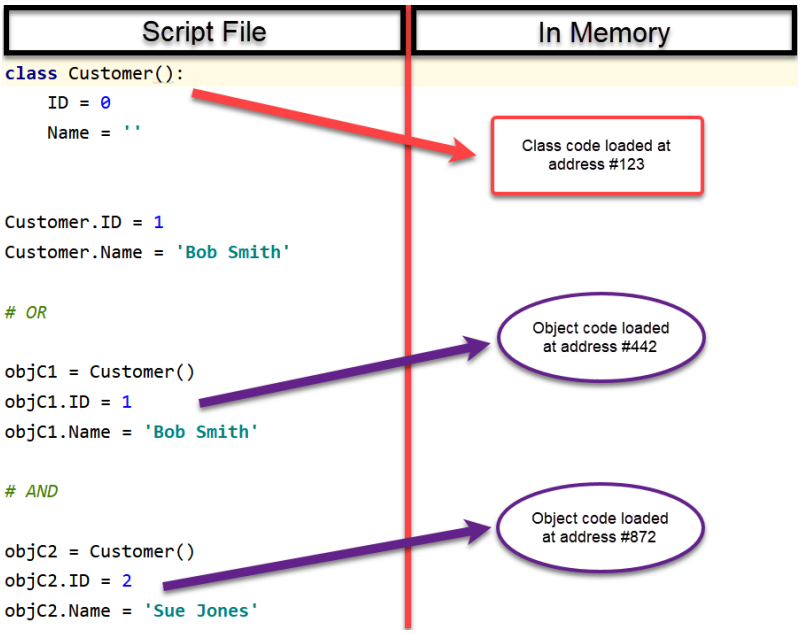

Figure 1. A simple example of objects made from a customer class

In general, you use a class **directly if its focus is on processing data** and **indirectly if its focus is on storing data**. This generalization may not always be true, but often enough that it provides 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 go work with them in the course!*

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.** Here is a pseudo-code example the different area that make-up a Python class:

```
class MyClassName(MyBaseClassName):

    # -- Fields --
    # -- Constructor --
    #     -- Attributes --
    # -- Properties --
    # -- Methods --
```

Fields

Fields are the data members of a class. Fields are created using variables and constants. Listing 1 shows an **example** of creating two object instances using a class called Person. **Each object instance can hold first name data unique to each person**.

```
# ---------------------------------------------- #
# Title: Listing01
# Description: A simple example of a class
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ---------------------------------------------- #

#--- Make the class ---
class Person():
    # --Fields--
    strFirstName = ""

    # -- Constructor --
        # -- Attributes --
    # -- Properties --
    # -- Methods --
# End of class

# --- Use the class ----
objP1 = Person()
objP1.strFirstName = "Bob"
```

```
objP2 = Person()
objP2.strFirstName = "Sue"

print(objP1.strFirstName)
print("-------------")
print(objP2.strFirstName)
```

Listing 1

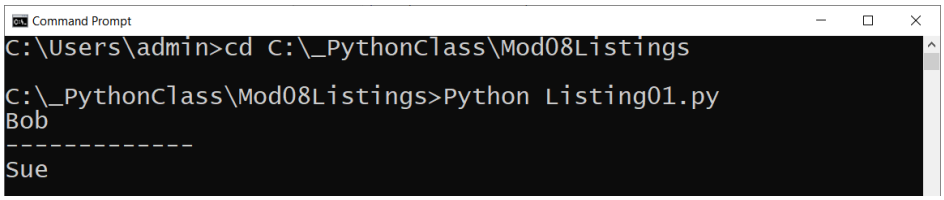

Figure 2. The results of listing 1

LAB 8-1

In this lab, you create a simple class to hold personal data.

1. **Create** a class called Person, using code like Listing 1.

2. **Add** both a strFirstName **and** a strLastName field to your Person class

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

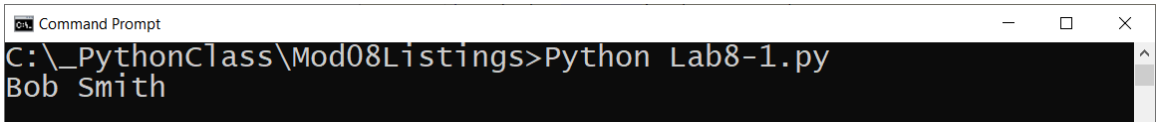

Figure 3. The results of Lab 8-1

Constructors

Constructors are **special methods** (functions) that **automatically runs when you create an object from the class**. Constructors are **often used to set the initial values** of Field data.

Python Constructor's use the **double underscore**("duder") **name** of **"__init__"** but in many languages, it is the same name as the class.

When you create an object instance from a class, **you use the class's name as if it were a function**.

```
objP1 = Person("Bob")
```

Python automatically calls the "__init__()" method and **passes any arguments you provide** to the "__init__()" method each time you make a new object.

Listing 2 shows an example of a constructor with one **parameter**, but of course, like any function, you **can have many** more.

```
# ------------------------------------------------ #
# Title: Listing02
# Description: A class with a constructor
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ------------------------------------------------ #



class Person(object):
    # --Fields--
    strFirstName = ""

    # -- Constructor --
    def __init__(self, first_name =''):
        #-- Attributes --
        self.strFirstName = first_name

    # -- Properties --
    # -- Methods --
# --End of class--

# --- Use the class ----
objP1 = Person()  # with only no argument
objP2 = Person(first_name="Sue")  # with the parameter and argument


print(objP1.strFirstName)  # will be empty
print("-------------")
print(objP2.strFirstName)  # will have first name
```

Listing 2

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

Destructors

Another special method is the "Destructor." These **automatically run when an object instance goes is removed from memory**. They are used to "clean up" any resources that are not needed once the object is gone. In Python, most of the resources are "self-cleaning," and so **you do not often see these in classes like you do Constructors**.

Destructors are considered an **advanced feature** and should be used with care. We **do not go into them in this course**, but their code looks like this.

```
    def __del__(self):
        """ automatically called when object is destroyed"""
        # TODO: Add some "cleanup" code
```

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,** but and **not directly in the class**. Many other languages use the word "this" instead of "self."

To understand the "self" keyword, start by remembering that the **code of a class always loads into memory** when your script starts running. There the class code sits, waiting, even if it never gets used.

```
class Customer():
      ID = 0
      Name = ''
```

On the other hand, you must **explicitly create** an **object instance in your script**.

```
objC1 = Customer()
```

The **class and the object instances** are **in different locations of a computer's memory** (Figure 4).

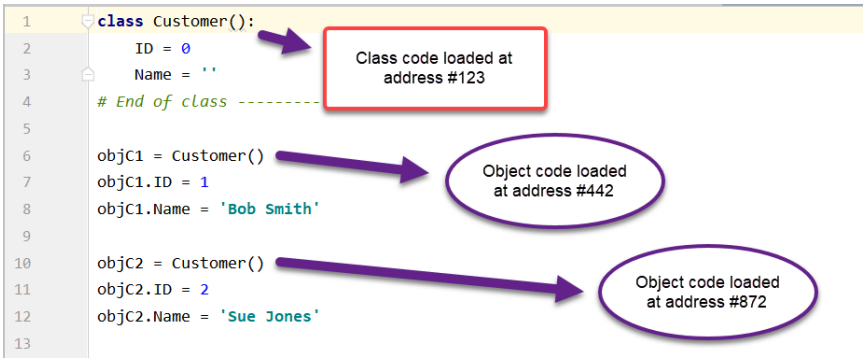

Figure 4. Classes and object instances load in different memory locations

You can **only load the class's code** in memory **once**, **but** you can have **multiple instances of a class**, each representing a "copy" of the classes code! **In Python, you identify which copy is referenced using the pronoun "self." like how two people conversing might each refer to themselves!**

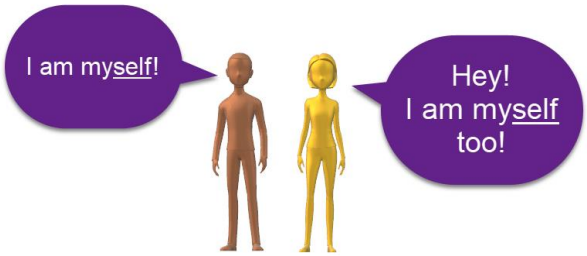

Figure 5. Using the "myself' pronoun is contextual

Because of this, the people who made Python made a **rule** that you **include a parameter called "self" in each method meant to be used from an object instance**.

Oddly, you **do not pass arguments into this parameter**. Nor is it **not automatically** assumed to be there **if you forget to type it in**.

LAB 8-2

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

1. **Create** a constructor for your Person class.

2. **Add** first_name and last_name parameters to the constructor of your Person class.

3. **Use** the parameter values to set the strFirstName and strLastName fields.

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

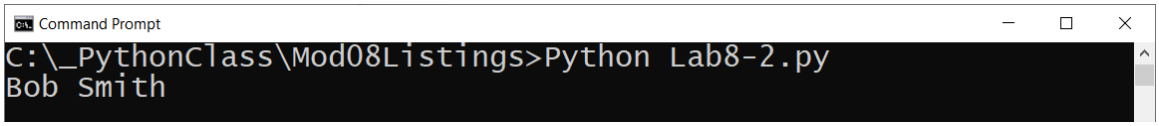

Figure 6. The results of Lab 8-2

Attributes

In Python, Attributes are "virtual" fields that hold internal data. An **invisible field** is created for you when you use the following syntax in the constructor.

Note: *This feature is* ***not typical in most other languages****, but it does keep to Python's mostly "automatic" nature!*

```
# ------------------------------------------------ #
# Title: Listing03
# Description: A class with an attribute
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ------------------------------------------------ #

class Person(object):
    # --Fields--
    strFirstName = ""

    # -- Constructor --
    def __init__(self, first_name):
        #Attributes
        self.FirstName = first_name

    # -- Properties --
    # -- Methods --
# --End of class--

# --- Use the class ----
objP1 = Person("Bob")
```

```
print(objP1.strFirstName)  # using the empty explicit field
print("-----------------")
print(objP1.FirstName)  # using the filled invisible implicit field/attribute
```

Listing 3

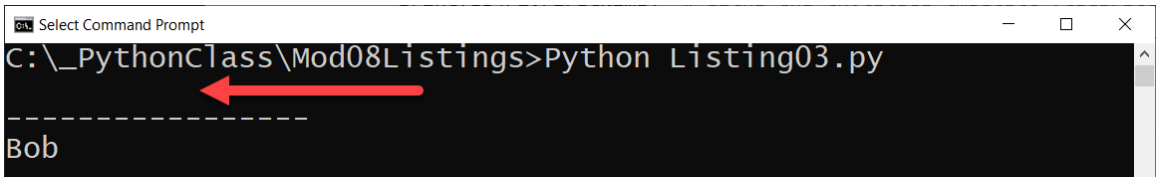

Figure 7. The results of Listing 3

Note: One problem with **Fields or Attributes** is that they **are just variables**. You do not have much control over what data goes into them unless you **write specific code to validate values before they are assigned**. **To help** with that, you can **use special methods (functions) called Properties**. We look at them after this next lab.

LAB 8-3

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

1. **Use** the constructor parameter values to **set** the FirstName and LastName attributes.

2. **Comment** out strFirstName and strLastName fields, since we are no longer using them.

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

Figure 8. The results of Lab 8-3

Properties

Properties are **functions used to manage field or attribute data**. You **typically create** two properties for each field/attribute, **one for "getting" data and one for "setting data**. In fact, you may hear them called "Getters" and "Setters" or "Accessors" and "Mutators."

Setter Properties 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 field or attribute. You **create a setter** like any other function, but it **must include** the @name_of_method.**setter directive**, and the **directive and function name must match**!

```
    @first_name.setter  # (setter or mutator)
    def first_name(self, value):  # The name must match the attribute!
```

```
        if str(value).isnumeric() == False:
            self.FirstName = value
        else:
            raise Exception("Names cannot be numbers")
```

Getter Properties let you add code to **format** a field's or attribute's data. Often, a Getter is **included in a class, even if there is no formatting code**. Inconsistently, Python **use the @property directive to indicate a getter** function.

```
    @property  # (getter or accessor)
    def first_name(self):  # The name must match the attribute!
        return str(self.__first_name).title()  # Title case
```

It is considered a **best practice to only work with the data in a class through a Method or Property**. This practice **creates a layer of "Abstraction"** and protects software using your class from internal changes to the Fields or Attributes. One of the reasons for this is Abstraction.

Note: Abstraction is an advanced topic that we come back to later in the course.

Forcing Property Use

When using Properties, you "**hide**" the attribute **using (2) underscores before the attribute's name**, which makes the attribute "***private***" and indicates not to use it directly!

Let me **rename the attribute to be "private**," and more Pythonic, by changing it to "__first_name."

```
def __init__(self, first_name):
    # -- Attributes --
    self.__first_name = first_name
```

I rename the attribute to be "private," and more Pythonic, by changing it to "__first_name." Now, our code looks like this:

```
    @first_name.setter  # (setter or mutator)
    def first_name(self, value):  # The name must match the attribute!
        if str(value).isnumeric() == False:
            self.__first_name = value
        else:
            raise Exception("Names cannot be numbers")
```

By convention, programmers should respect hidden attributes as being private. However, **Python does not vigorously enforce this privacy, unlike most other languages**. Listing 4 shows an example of how Properties and Attributes work together to manage data.

```
# ---------------------------------------------- #
# Title: Listing04
# Description: A class with an attribute
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ---------------------------------------------- #
```

```python
# --- Make the class ---
class Person(object):
    # --Fields--
    #strFirstName = ""

    # -- Constructor --
    def __init__(self, first_name):
        # -- Attributes --
        self.__first_name = first_name

    # -- Properties --
    # FirstName
    @property  # DON'T USE NAME for this directive!
    def first_name(self): # (getter or accessor)
        return str(self.__first_name).title()  # Title case

    @first_name.setter  # The NAME MUST MATCH the property's!
    def first_name(self, value):  # (setter or mutator)
        if str(value).isnumeric() == False:
            self.__first_name = value
        else:
            raise Exception("Names cannot be numbers")

    # -- Methods --

# --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"
print("Test Numbers -------------------------")
try:
    objP1.first_name = '123'
    # using the number causes a validation ERROR
except Exception as e:
    print(e)
print("Test Reading Attribute -----------------")
try:
    print(objP1.__first_name)
    # reading the attribute directly causes a runtime ERROR
except Exception as e:
    print(e.__doc__)
print("Test Assigning Attribute ----------------")
# inconsistently, you can set a hidden attribute WITHOUT an error,
objP1.__first_name = '123'  # but don't do this!
print(objP1.first_name)  # Besides, python seems to ignore it!
```

Listing 4

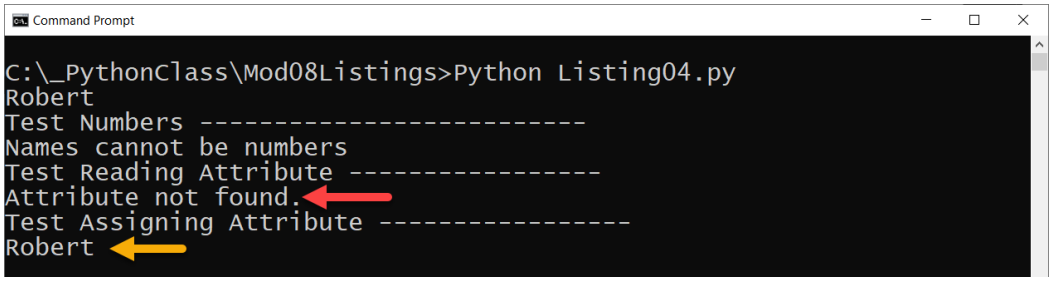

Figure 9. The results of listing 4.

LAB 8-4

In this lab, you **modify** the Person class you made in Lab 8-3 to use hidden **attributes**, with a getter and setter Property for each.

1. **Modify** the constructor's attribute name to __first_name and __last_name.

2. **Create** a getter and setter Property for both the __first_name and __last_name attributes.

Tip: *This lab does not require validation and formating code in the properties.*

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

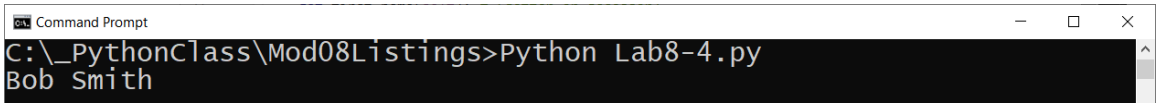

Figure 10. The results of Lab 8-4

Methods

While functions that manage attribute data are called properties, **other functions inside of a class are called Methods**. Methods allow you to organize your processing statements into named groups, just like functions in scripts do!

```
# ---------------------------------------------- #
# Title: Listing05
# Description: A class methods
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ---------------------------------------------- #

class FileReader:
    def ReadFileDataToString(self, file_name):
        with open(file_name, 'r') as file:
            return file.read()
```

```
f = FileReader()  # create an object
print(f.ReadFileDataToString("AppData.txt"))  # use its method
```

Listing 5

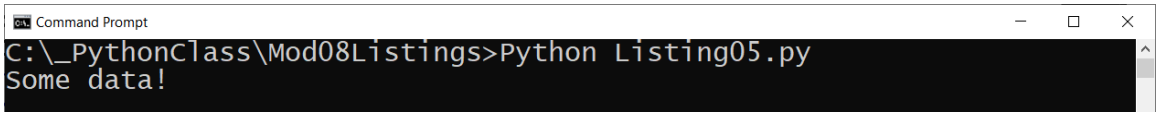

Figure 11. The results of Listing 5

The "__str__()" Method

Most classes, in most languages, include a **method that returns** some or all the **class's data as a string**. Python has a built-in method that performs this task called the "__str__()" method, but in many languages, it's called something like "ToString()."

Python includes an invisible "__str__()" method if you do not add one to a class. This **default** invisible method **only returns the name of class and an address identifier**, and you may have seen this going through the course. However, can **override this method to return something more useful**, such as the contents of the class's attributes (Listing 6).

```
# ------------------------------------------------ #
# Title: Listing06
# Description: A overriding the __str__() method
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ------------------------------------------------ #

class Demo1:
    var1 = "Some Data"

class Demo2:
    var1 = "Some Data"
    def __str__(self):
        return self.var1

d1 = Demo1()  # This object uses the default __str__() method
print(d1)

d2 = Demo2()  # This object uses the overridden __str__() method
s = str(d2)  # __str__() method run when the str() function is called
print(s)
print(d2)  # __str__() method run when the print() function is called
print(d2.__str__())  # __str__() method run when function is called directly
```

Listing 6

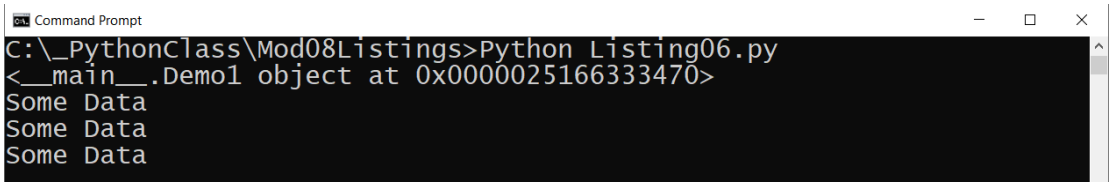

Figure 12. The results of Listing 6

Note: *There are many build-in methods automatically and invisible included in every class, using something call inheritance, which we talk in module 9.*

LAB 8-5

In this lab, you **modify** the Person class you made in Lab 8-3 to override the "__str__()" method.

1. **Add** a new method to override the **"**__str__()" built-in method.

2. **Add** code to the method to return both the first_name and last_name with a comma separator.

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.

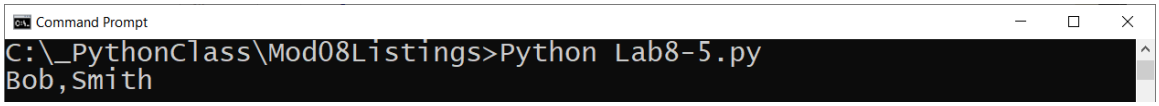

Figure 13. The results of Lab 8-5

Static Methods

If you want to include **methods called directly from the class**, without making an object first, you add the @staticmethod directive like this:

```
class Math(object):

    @staticmethod
    def Add(Value1, Value2): # You do not need the self keyword
        return Value1 + Value2
```

Now, you can call the method by using the name of the class and the name of the method.

```
Sum = Math.Add(5, 6)
print(Sum)
```

Important: *You **do not need the "self" keyword with a static method** since you are not calling the function from an object instance, and because the class is only loaded one, there is no need for a confusing pronoun!*

Classes can have both instance methods and static methods, but most will not. **In general**, when a **class focuses on processing data**, use "**static" methods. However,** when a **class focuses on storing data,** use **"instance" methods (the ones with self)**. For example, a class that holds a **customer's name, email address, phone number**, and other data about the customer, so use **mostly instance methods** because you would want to make **multiple copies of the class** as you program runs, each with its own **unique customer data**. On the other hand, a class that processes data from a list object into a file and from a file to a list object would be most about performing actions

that would not require multiple copies of the class with unique data. This generalization may not always be true, but often enough that it provides a good starting point.

Private methods

If you want a **method** to be **for internal processing only**, in other words, "**private**" to the object or class, you can **name it with (2) underscores**. These can be **occasionally useful** for things **like tracking how many objects are currently created from a class**

Counting objects is a critical task in older languages because you needed to make sure that **objects** were all removed from memory before the application closed. If you did not do so, they **stay in memory taking up valuable space and making the computer run slow**. To clear them from memory, you reboot the computer, a task that many of us do regularly.

Python self-cleans object to avoid this, **but let's look at an example anyway**, since it is something that might come up in other languages.

Note: *You do not need to include an object counter in this module's assignment.*

```
# ------------------------------------------------ #
# Title: Listing07
# Description: A class with public and private methods
# ChangeLog: (Who, When, What)
# RRoot,1.1.2030,Created Script
# ------------------------------------------------ #

# --- Make the class ---
class Person:
    # --Fields--
    # This is one of few the times a field is used as a global variable
    __counter = 0

    # -- Constructor --
    def __init__(self, first_name):
        # -- Attributes --
        self.__first_name = first_name
        # call the private method each time an object is made
        Person.__set_object_count()

    # -- Properties --
    # First Name
    @property  # DON'T USE NAME for this directive!
    def first_name(self): # (getter or accessor)
        return str(self.__first_name).title()  # Title case

    @first_name.setter  # The NAME MUST MATCH the property's!
    def first_name(self, value):  # (setter or mutator)
        if str(value).isnumeric() == False:
            self.__first_name = value
        else:
            raise Exception("Names cannot be numbers")

    # -- Methods --
    def __str__(self):
```

```
        return self.first_name

    @staticmethod  # You do not use the "self' keyword
    def get_object_count():  # This is a PUBLIC static method
        return Person.__counter

    @staticmethod  # You do not use the "self" keyword
    def __set_object_count():  # This is a PRIVATE and static method
        Person.__counter += 1

# --End of class--

# --- Use the class ----
objP1 = Person("Bob")
print(objP1)  # using the instance method __str__()
print(objP1.__doc__())  # using the instance method __str__()
print("--------------------------")
print(Person.get_object_count())
objP1 = Person("Sue")
print(Person.get_object_count())
```

Listing 7

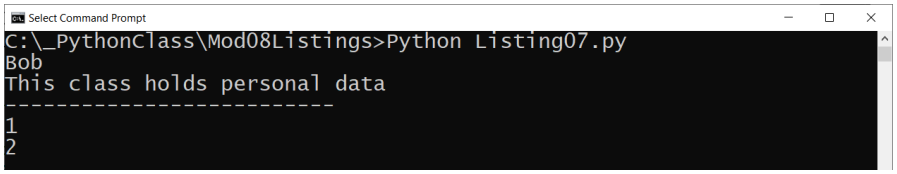

Figure 14. The results of Listing 7

Type Hints

Another documentation feature that was added to Python is "type hints." These are be used to what type of data is expected in a variable.

Unlike other languages, the Python runtime does not enforce parameter and variable types, so incorrect argument types can be passed into to a parameter. Still, type hints can help other developers understand what is wanted and some IDEs provided tips to reduce mis-typing.

Type hints are commonly used for function parameter and return types using the following syntax:

```
def __init__(self, first_name: str):
    """ Sets initial values and returns a Person object
ff    """
objP1 = Person("Bob")
```

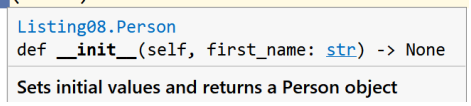

Figure 15. Type Hints in PyCharm IDE

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** to show you a 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).

```
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
     """
```

Listing 8

```
print(Person.__doc__)
objP1 = Person("Bob")
```

```
Listing08
C:\Users\admin\AppData\Local\Programs\Python\Python37-32\python.exe C:/_PythonClass/Mod08Listings/Listing08.py
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
```

Figure 16. The results of viewing a docstring in PyCharm

Using Classes

While processing classes are used directly by calling their static methods, **data classes are mostly used** indirectly through **a set of one or more objects**.

```
objP1 = Person("Bob", "Smith")
objP2 = Person("Sue", "Jones")
```

The **data in these objects are like a row of data. These "rows"** are **often collected into a list** object.

```
lstPersons = [objP1, objP2]
```

You can think of this collection **like a table** of data in a spreadsheet or database.

FirstName	LastName
Bob	Smith
Sue	Jones

Table 1

List like any list, **you can loop through the collection** and do something with the individual "rows" of data, such as **writing each object's data to a file**.

```
f = open("Persons.txt", "w")
for row in lstPersons:
    f.write(str(row) + "\n") # calls __str__()
f.close()
```

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**."

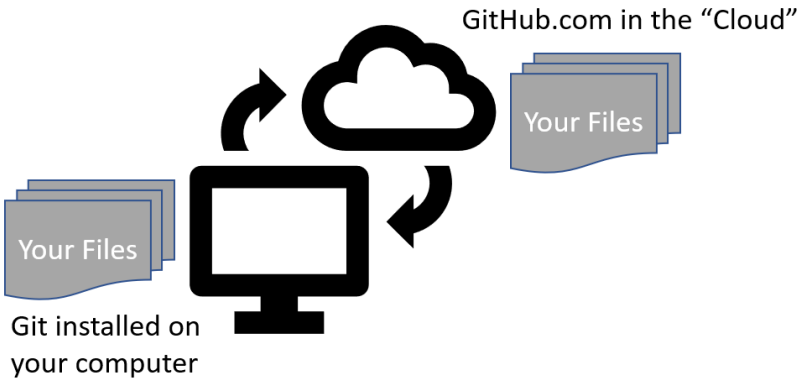

Figure 17. Git and GitHub work together to manage files

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).

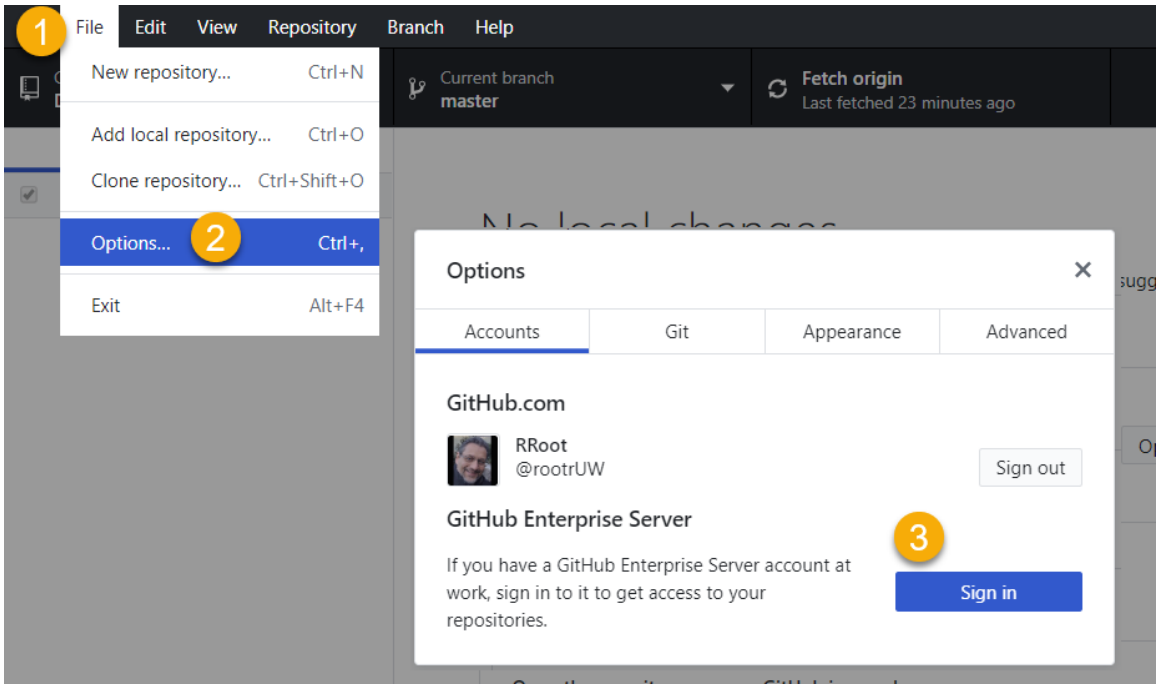

Figure 19. Changing the GitHub account option

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.

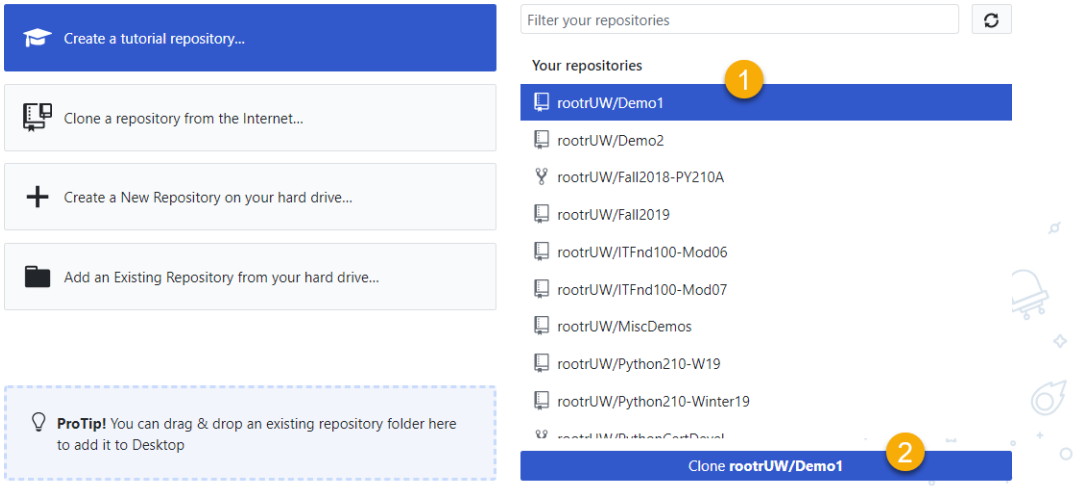

Figure 20. The repository listing

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.

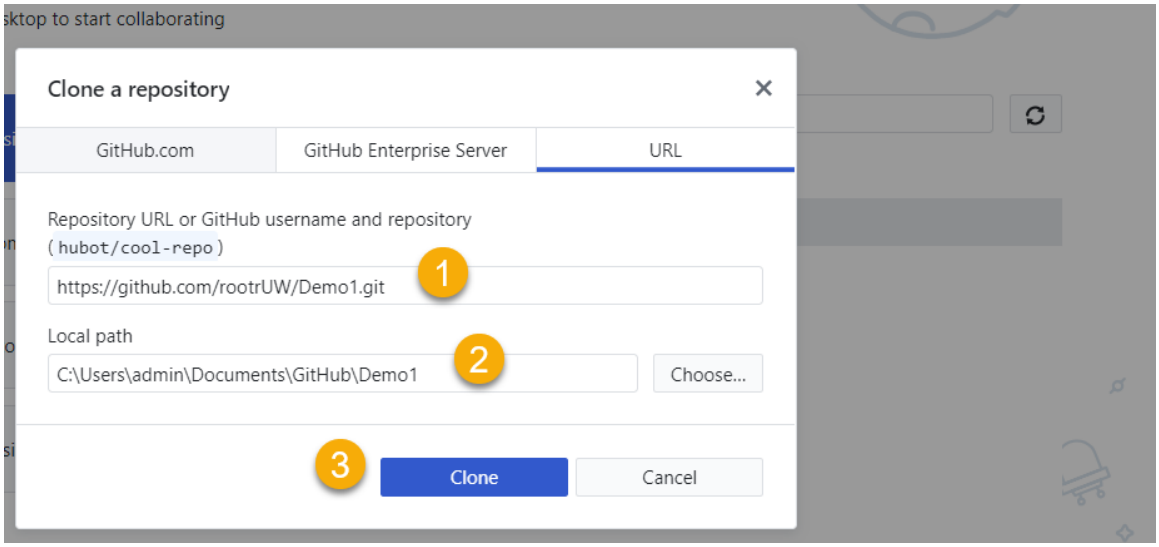

Figure 21. The "Clone a repository" dialog box

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).

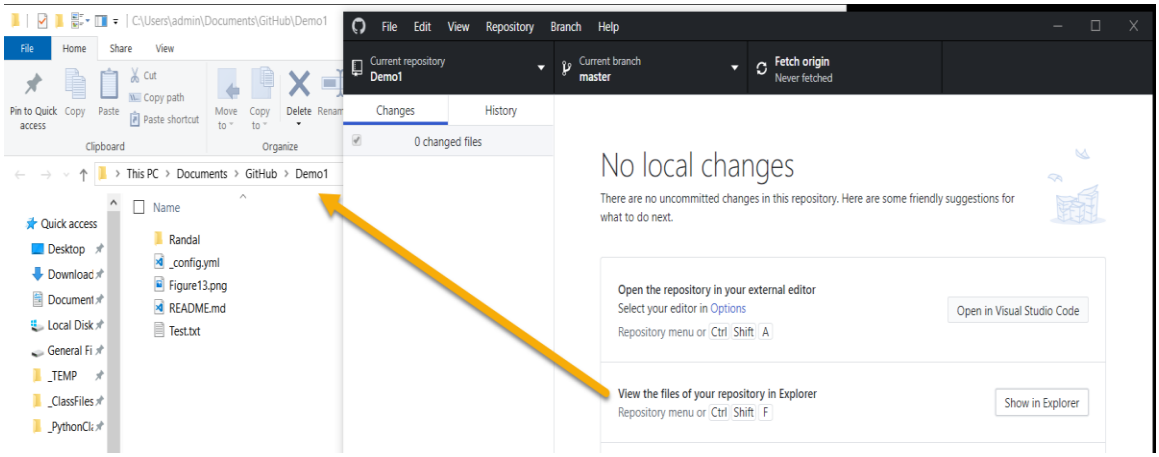

Figure 22. Viewing the local files

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. Figure 23 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**.

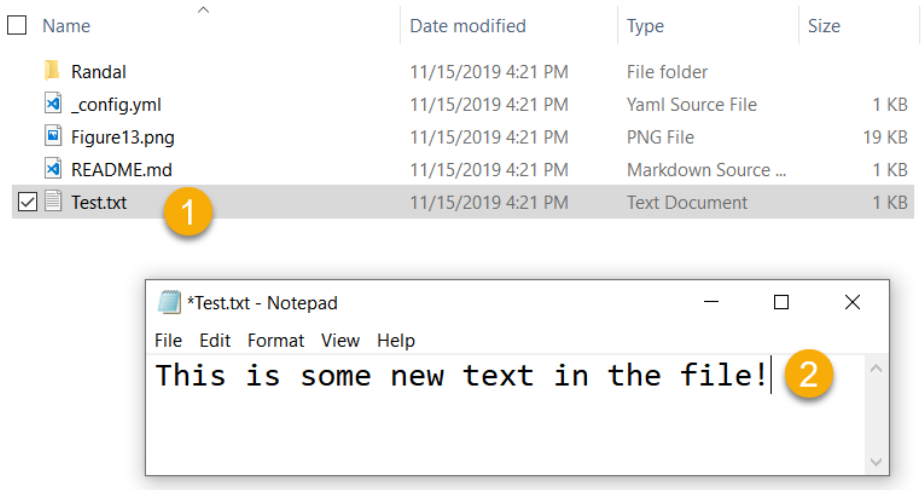

Figure 23. Changing text in a Git managed file

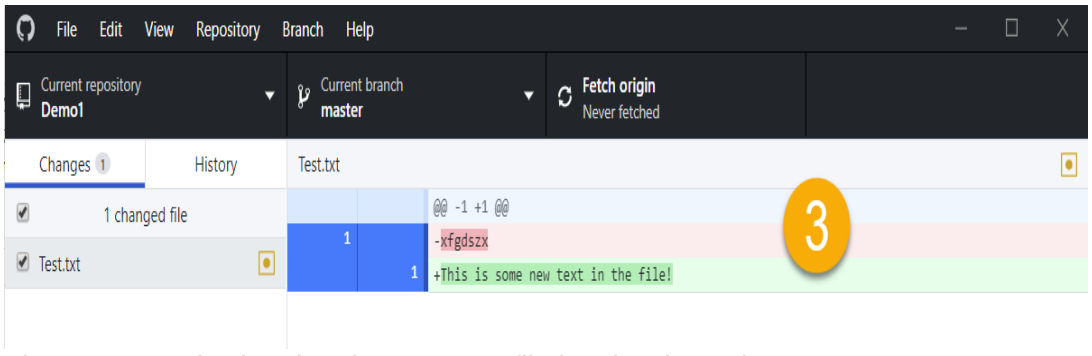

Figure 24. Reviewing the changes to a file in GitHub Desktop

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

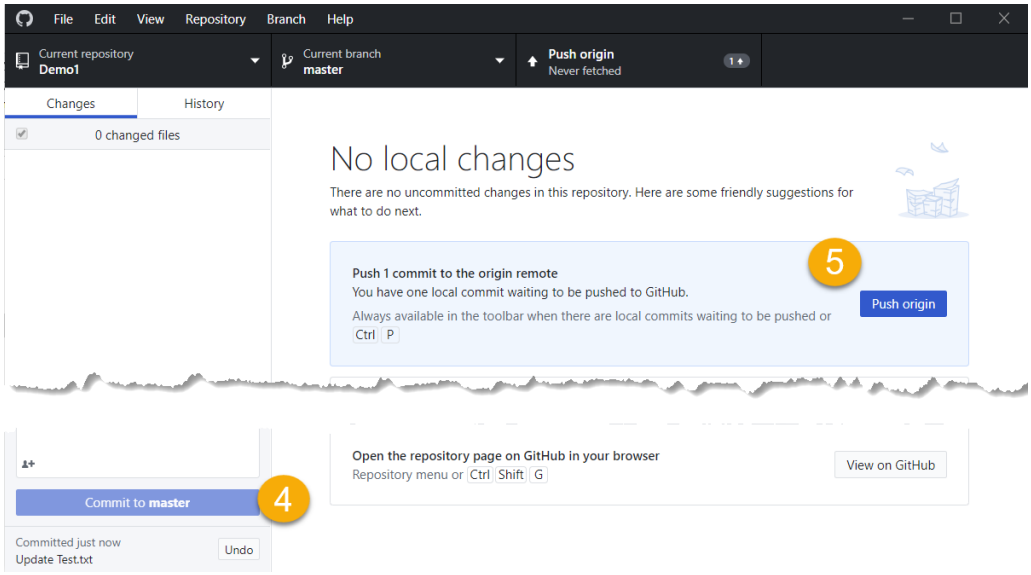

Figure 25. Uploading the changes to GitHub

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.

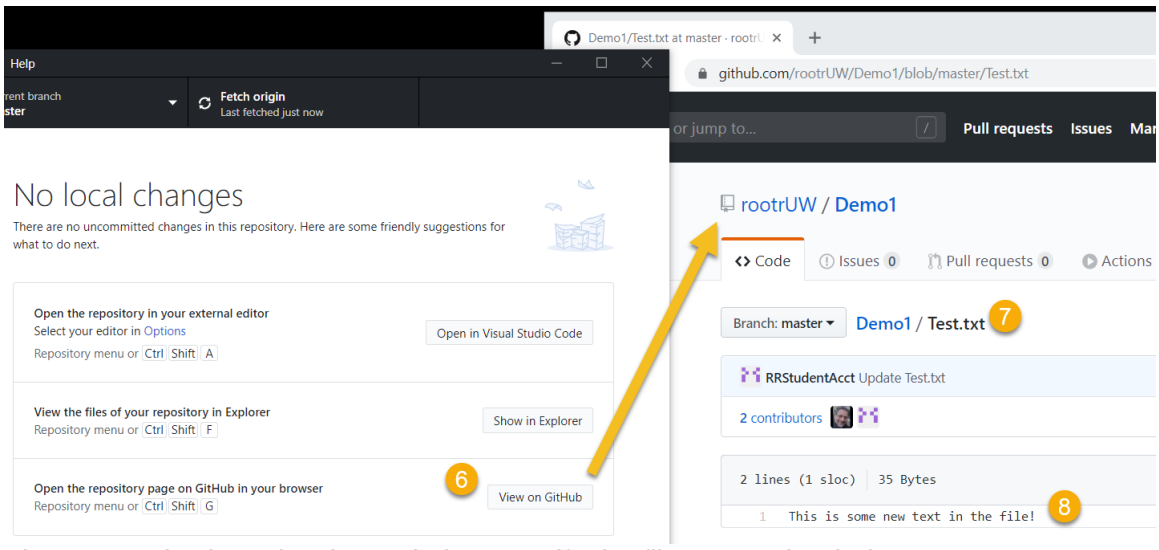

Figure 26. Viewing GitHub's website to verify the files are uploaded.

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.