# Natalie E. Tolliver

6/8/2020

ITNFDN 100A

Assignment 08

<https://tollivne.github.io/IntroToProg-Python-Mod08/>

Using Custom Classes

## Introduction

The purpose of this script is to create code that, instead of managing a “ToDo” list as in the previous code, manages a “Products List.” The user will input the product name and product price instead of an item name and item priority. The menu of user choices was simplified to delete the option to remove an item; and an choice was added to display the current options. This demonstrated how functions can be easily reused for different purposes by changing the name of the variables used to call the functions. Next, the code was changed to incorporate the use of custom classes. A custom “Product” class was created to process Product Name and Product Price. Finally, GitHub desktop was used to push the code to the GitHub website.

# Phase I – Updating the Code to Manage a Product List

The code from the previous assignment managed a “ToDo” list with the user entering the Task Name and Task Priority. In this code, instead of Task and Priority, the user will be asked to input the Product Name and the Product Price. In addition, the option to remove an item has been deleted from the code and an option to display the current list of items has been added. The feature of displaying the current list when the program starts up has been removed. So as not to introduce errors, at each step of the code change, the program was tested before going on to the next update. As I was going through these changes, at first it seemed tedious, but then I realized how efficient the process was because the functions were being reused for a different application.

# Phase 2 – Copying the Previous Code

The next step was to copy the code from the modified ToDo list program into the “Products” program. I copied the sections for “Adding a Task,” “Displaying a Task” and “Saving to File”, and “Exiting the Program.” For this exercise, the data was loaded into memory automatically, so that option was no longer needed in the menu items. I took out the sections for “Removing an Item.” I renamed the variables from describing ToDo list items to describing product items. I renumbered the menu and tested the program at each phase.

# Phase 3 – Creating the Product Class

I created the “Product” class to organize the data. Classes help you to organize your data using the custom formatting, custom variables, and custom methods. A class can be used either directly or indirectly. To be used directly, the class is called similar to the way a function is called, by using the name of the class and the dot and then the variable name in the class. An example of using the class directly is shown below:

Product.product\_name = “Eggs”

To call the class indirectly, you create another variable which calls the class, this is referred to as creating an “object instance.” The advantage of this process is that you can have more than one object instance. An example is as follows:

objProduct1 = Product()

objProduct1.product\_name = “Eggs”

objProduct1.product\_price = 3.00

objProduct2 = Product()

objProduct2.product\_name = “Oranges”

objProduct2.product\_price = 2.50

You can also call a method from a class. A direct function call, for example, would look like this:

Product.ReadDataFromFile()

For this project, since the focus is on storing data, the class will be called indirectly so that multiple product objects can be used to represent the name and price of multiple products. The standard pattern for a custom class (classification) is as follows:

class MyClassName(MyBaseClassName):

# -- Fields –

# -- Constructor –

# -- Attributes –

# -- Properties –

# -- Methods

MyBaseClass is an optional feature and can be omitted. The following items were created as part of the class.

1. Fields - Variables or constants within the class are known as “fields.” They may also be referred to as “attributes” or “properties.” The Product Class had two fields, product\_name and product\_price. Product\_name was to be formatted as a string and product\_price was to be formatted as a floating point number. The values would be assigned later, so, similar to declaring variables at the beginning of a program, the fields are put at the beginning of the class.

# -- Fields –

strProductName = “”

intProductPrice = 0.0

Typically, you work with the fields indirectly by creating an object instance and then assigning the values. It is recommended that you use a prefix in field names as a “hint” as to the type of variable.

1. Constructors - Constructors are functions or methods that automatically run when you create an object from the class. They use the double underscore name of “\_\_init\_\_” and are usually used to set the initial values of your fields. Python automatically calls the \_\_init\_\_ method and passes any arguments you provide to it each time you make a new object. Constructors do not return anything. An example of code for a constructor would be as follows:

# -- Constructor –

def \_\_init\_\_(self, product\_name):

self.strProductName = product\_name

Where strProductName is the variable name inside the function and product\_name is the variable that is passed to the function. The word self has to be used when you plan to use the class by making object instances later on in the program. The constructor runs once at the time you call the class.

You can pass arguments into the constructor by calling the class with the syntax as shown in this example:

objProduct1 = Product(“Eggs”, 3.0) or alternatively

objProduct1 = Product(product\_name = “Eggs”, product\_price = 3.0)

1. Attributes – Attributes are similar to fields. It is a method of setting an “invisible” field. Typically, the variable names in the attributes do not use a prefix. So, the attributes could be used to set the invisible fields as follows:

Self.ProductName = product\_name

This statement would be placed AFTER the \_\_init\_\_ statement in the constructor whereas field statements are placed BEFORE the \_\_init\_\_ statement. If you use attributes, you typically don’t use fields anymore.

1. Properties – Properties are a way of managing the fields and attributes of a class. They are a special type of “method” to manage the data in the class. You typically use two of these methods for processing the data. One of the functions “gets” the data and another one “sets” or formats the data. They are called “getter”s and “setters.” Alternatively, they may be called “accesors” and “mutators.” Setters are a good way to add code to validate the data and include error handling while a getter is a good way to format the data in the class. If a value that is passed into the properties parameter is valid, then it gets assigned to the field or attribute. Here is an example of the setter for the product name.

@product\_name.setter *# The NAME MUST MATCH the properties!*def product\_name(self, value: str): *# (setter or mutator)* if str(value).isnumeric()== False:  
 self.\_\_product\_name = value  
 else:  
 raise Exception(**"Names cannot be numbers"**)

The “getter” property is used to format the variables (called fields). For the product name, in this project, I wanted to use title case, so the following code was used:

@property *# DON'T USE NAME for this directive!* def product\_name(self): *# (getter or accessor)* return str(self.\_\_product\_name).title() *# Title case*

Notice the oddity that it is not referred to as “setter.”

A Python best practice is to only work with the data in a class by using a method or property and not change the data in the main body of the program. For this reason, adding products to the list was moved from being a “Static Method” in the “Processing” class to being a “Method” inside of the custom Product class.

1. Methods – A function within a class is known as a “method” and is used to do tasks inside the class. They are also sometimes called “procedures” or “sub-procedures.” One of the methods that was used in this project was the \_\_string\_\_ which is known as a “dunder” for “double underscore.” It is used to convert the set of data inside the class (which includes product name and product price) into a string. If I hadn’t made a custom string method, the default python method is to put the set of data in text without any comma delimiting format. Without using the custom string method, when Python prints out your class, the invisible method is automatically called.

def \_\_str\_\_(self):  
 *""" converts product data to string"""* return self.\_\_str\_\_()

This format causes the product object data to print out in a format that looks like this

<\_\_main\_\_.Demo1 object at 0x000002195B2064C0>.

This is telling you that from the “Main” script, the class called Demo1 is stored at that hexadecimal address location. To override this hidden method, I used the following method to put the product name and price into one comma delimited string:

def \_\_str\_\_(self):  
 *""" Converts product name and price into aa comma separated string for writing to file  
 :param produce\_name string attribute  
 :param produce\_price floating point attribute to be converted to string  
 """* return self.product\_name + **","** + str(self.product\_price)

# Phase 4 – Putting it All Together

Once the class had been created, the script had to be modified to use the class. Recall, the main body of the program and all of the static methods used list data. They had to be modified to read in a list of products, write a list of products, display a list of products, and add to the list of products. Here is an example of the code updates that needed to be made for the product type data.

for line in file:  
 product\_name, product\_price = line.split(**","**)  
 row = Product(product\_name, float(product\_price))  
 list\_of\_product\_objects.append(row)  
file.close()  
return list\_of\_product\_objects, **"Data was read from file!"**

The code shown above reads the comma delimited data from a text file, puts it into the variables product\_name and product\_price for each line in the file. It then calls the Product class using the name and the product and appends the product data row to the existing data. All the functions and function calls in the code were modified accordingly to change to processing “product” data instead of list data.

Figure 1 shows a series of screenshots of the final program run in PyCharm.

A screenshot of a social media post

Description automatically generated

A screenshot of a social media post

Description automatically generated

# A picture containing bird Description automatically generated

A screenshot of a cell phone

Description automatically generated

Figure - Final Code Run in Python

The resulting products.txt file is shown in Figure 2.

A screenshot of a cell phone

Description automatically generated

Figure - Resulting text file

The following code was used for error handling because in the CMD window, it is necessary to navigate to the path that contains the script before running it.

import sys

try:  
 file = open(file\_name, **"r"**)  
except FileNotFoundError as e:  
 print(**"Please make sure the file by the name of : "** + file\_name + **" exists!"**)  
 print(**"If using the CMD window, change directory to the folder you're working in "  
 "by typing the following command cd c:\\_Name of Path"**)  
 sys.exit()

Figure 3 shows the results of running this code in the command window.

A screen shot of a computer

Description automatically generated

Figure - Error Handling in CMD Window

Once you navigate to the proper directory, the script runs as shown in Figure 4.

A screenshot of a computer

Description automatically generated

Figure - Program Run in CMD Window

# Summary

This project used custom classes to manage a product list. By using the custom class, I was able to format the variables and validate the variables using error handling before assigning the value to the variables and use custom methods for processing the data. Additionally, I was able to use the custom class when reading and writing to and from a file and when adding products to the list.