Keith Burdette

March 13th, 2020

Foundations of Programming: Python

Assignment 08

https://github.com/twkeith/IntroToProg-Python-Mod08

**Using Python Classes**

**INTRODUCTION**

For assignment 8, we were asked to utilize python classes to accomplish a task that tracks a pair of data featuring a product name and its price. This is similar to the list and dictionary objects that we have previously used to store and collect similar information. Classes give us greater control and flexibility over objects that we can customize to fit our needs.

**PRODUCT CLASS**

The first part of the assignment involved creating our own product class. Python has many built-in functions for classes that allow us to create and display our custom objects. A constructor is a function that creates new instances of our class. The base definition for our product class constructor is shown in figure 1.

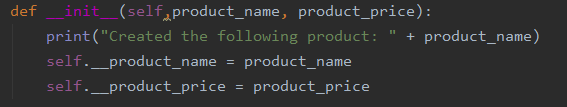


Figure 1 – Class Constructor

The \_\_init\_\_ name is the base constructor for all classes. It can take any number of parameters but the first one is always *self* which enables the function to refer the object just created. Subsequent parameters can be any combination of values to make up properties of the object. In this case, we have two extra parameters for product name and price which we use to create the two properties of our class.

In the body of the function, we parse our two parameters into two class properties. We take the self object and set two new properties using the form *self.\_\_propertyname*. The double-underscore in the property name sets that value to private which means that users of our class cannot directly access and change these properties of the class. For users to manipulate these new properties, we need to create *getter* and *setter* functions as methods in our class. These are shown in Figure 2.

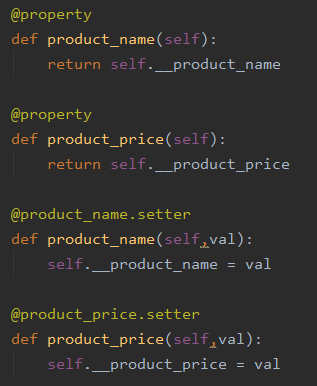


Figure 2- Getters and Setters

Getter and setter functions get and set values of our class. For a getter function, add the @property tag immediately before the function declaration which is simply def followed by the name of our property and the self object as a parameter. The body of the function is a return statement of self.propertyname. We built two getter functions for our class, one for product\_name and product\_price. Setter functions start with a tag of @propertyname.setter before the function declaration. The declaration line looks similar to the getter function but has an extra parameter for the value you wish to set to the property. In the body of this function, you can do some error handling to ensure the values fit the criteria for the property. For instance, in this case we could force all product\_prices to be numbers. I chose to do that in a different part of the code for this assignment

The last part of our product class is the string representation method. This allows our class to be displayed from the print function with a format we choose. To create this function, we declare a \_\_str\_\_ as a function with self as the parameter. In the return statement, we put a string of how we want our two properties displayed. For our product price, I ensure that my price value always displays with a $ and two decimals as shown in Figure 3.

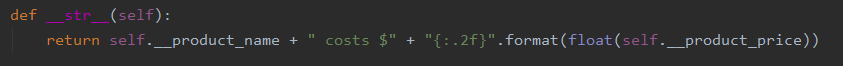


Figure 3 – String Representation Method

**USING THE CLASS**

To track objects in our product class, we use a second File Processor class with methods that read from a file, write to a file and add a new product object to our list. Reading data from a file, we assume our product price pair is in a comma-separated list. Since we also create the function that writes to this file, we can ensure that happens. Figure 4 shows our read\_data\_from\_file function.

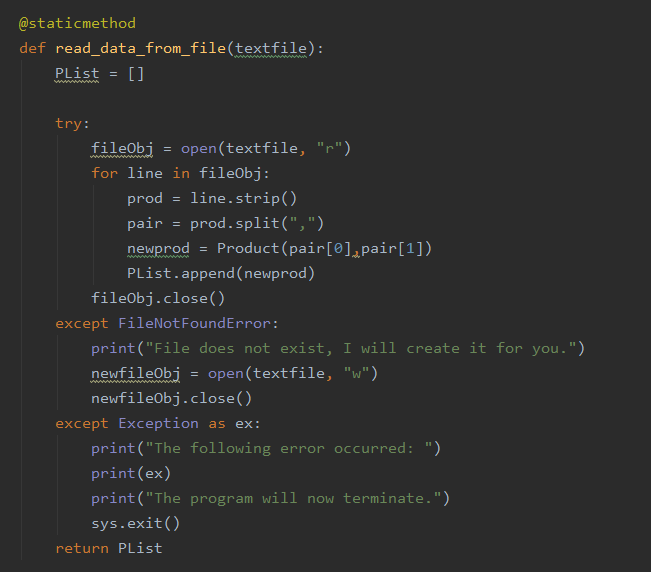


Figure 4- Read From File

The function declaration starts with a staticmethod tag which enables us to call this function without a File Processor class object. The function’s only parameter is the filename where our data lives. In the function, we create an empty list where we will collect our product class objects. In a try/except block, I open the file and process it. If the file doesn’t exist, I can catch that with a FileNotFound exception and create a new file so the program can continue. If another error happens, I added a generic exception that will print what happened and exit the program. I’m not sure what else could go wrong but if a user encounters this exception, they can share the error statement with me so I can fix it.

To process the text file, I create a for loop to go through it line by line. I strip any extra characters and use the split function to put my product and price pair into a list. I create a newprod variable that will be an instance of my Product class built using my Product Constructor with two arguments being the product and price. Last, I append this new product class object to my list.

Onto the write file. There are two parameters, one for the text file and a second for the list of product objects. I create a blank list where I can collect a formatted comma-separated product and price pair that I will ultimately run through the writelines function. I loop through the product list and use my class getter functions to format the string. This is shown in figure 5.

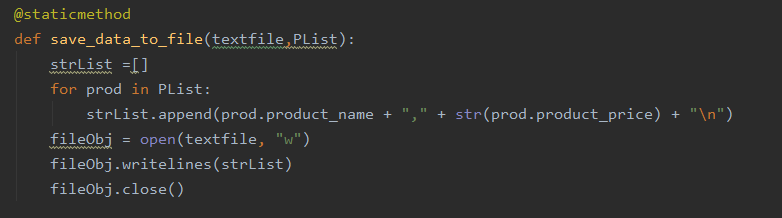


Figure 5 – Save Data to Textfile

To add new product, I created one last File Processor function which I also tag as a staticmethod. This takes three parameters, a product name, price and our list of products. In this function, I use the constructor to build the product object with the product name and price. Within the constructor, it prints out that the object has been created. Then this new produt gets added to the product list. This function is shown in figure 6.

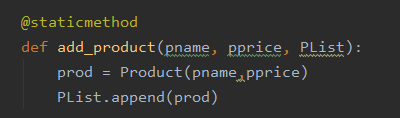


Figure 6 – Add Product to List

The last class created for this program was an IO class. This is very similar to the class we’ve used the last couple assignments. It has a series of staticmethod functions that show the user menu, grab input from users, asks for a new product and prints the list of products. The new product function asks for two inputs, a name and price. For the price, I try to convert the input to a float. If the input is not numeric, that raises an exception which tells the user to enter something that is a number and tries again. This is shown in figure 7. The print products function loops through all the products and uses the \_\_str\_\_ class function to print our formatted string telling you how much each product costs. This is shown in figure 8.

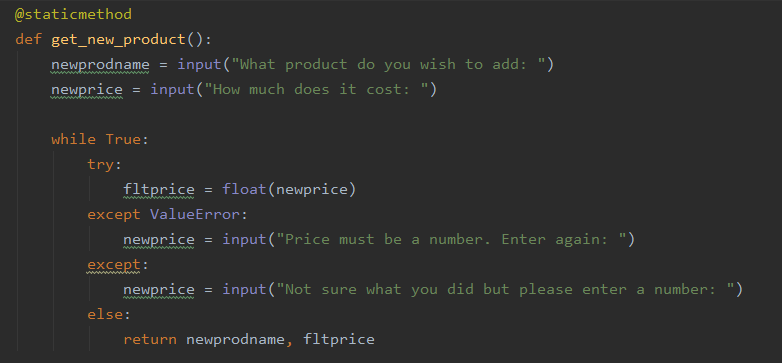


Figure 7 – Input Product Function

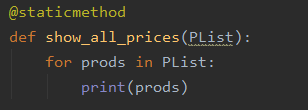


Figure 8 – Print All Products

**TESTING THE CODE**

Starting the program, the first thing that happens is loading the text file. I put some sample data in the text file so it had something to start with. For each product that gets created, it calls the constructor which prints a statement saying that particular object has been created. Then it shows the menu options available to the user. This sequence executed from PyCharm is shown in figure 9.

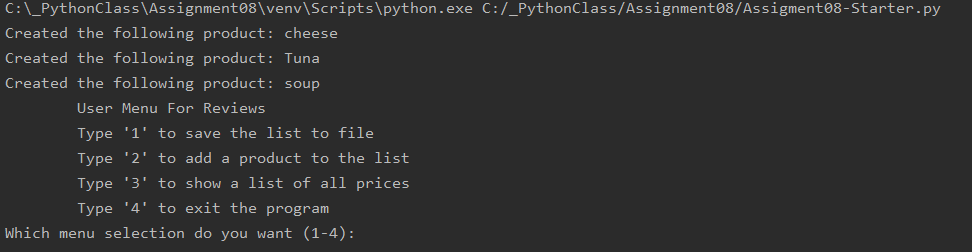


Figure 9 – Initial Run of Program

In figure 10, I show inputting a new item. I initially try to add a dollar sign to the price but it gives me an error and tells me to try again.

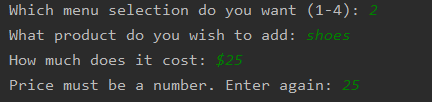


Figure 10 – Error Handling in Input Function

In figure 11, I switch to the command line which loads my changed text file. I then choose the show all prices option which prints all the items from my list. This covers all the functionality of our program.

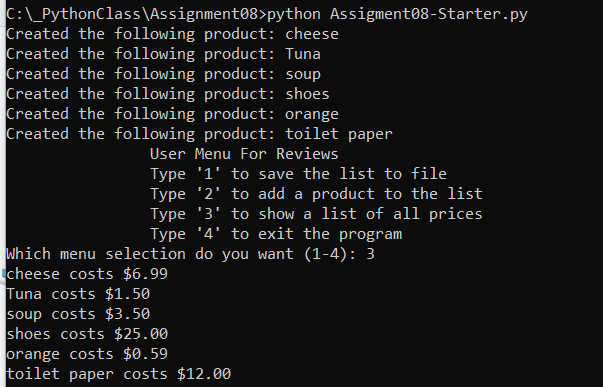


Figure 11- Command Line Start the Program, Print All Items

**SUMMARY**

This assignment asked us to demonstrate custom classes. Previously we have used classes that just held staticmethod functions that performed specific tasks. This week added the wrinkle of using properties of a class which act similar to attributes of items in a database. Classes help add a layer abstraction that can be hard to implement with a list object.

In our case, we tracked a product’s name and its price. We built a constructor that built an instance of our class and set initial values for its properties. We created getter and setter functions that allowed us to prevent bad data from getting into these properties. This class could easily be extended to create new functionality such as tracking how many of a particular item you have in stock or what a sale price would be if you want to take off a certain percentage. We combined these class capabilities with things we’ve learned previously like writing to a file and list manipulation that a store owner could use to track products they sell in their store.