Classes

# Introduction

We have been asked to describe and demonstrate the use of Python classes. The program reads an existing list from a file and can add, and print the products and their values, or save the information back to the file. When adding products, the program asks for the name of a product, and then asks for its estimated value. Both pieces of data are stored as attributes in a class object. Each object is then added to a list to create a table of data which can then be printed or saved. This is similar to the Home Inventory program developed for the Module 4 assignment, but uses classes and functions to organize the script similar to Module 6.

# Writing the script

To begin with, we were given a starting template to follow in developing the code. The code was broken down into sections for data, processing, and presentation, along with the main body of the script. The processing and presentation sections were defined as classes called *FileProcessor* and *IO* respectively. The variables had already been declared in the data section, and some of the functions laid out. Most of the missing functions were simply re-used. While developing the final code, many of the functions were copied to another file to be worked individually then combined to generate the complete script.

## Main Body

The main body of the script is similar to much of our earlier work: create a menu for the user to choose from then use a branching structure inside of a loop to do the work (see Figure 1, Figure 2).

*# Main Body of Script ---------------------------------------------------- #  
# TODO: Add Data Code to the Main body  
# Load data from file into a list of product objects when script starts  
# Show user a menu of options  
# Get user's menu option choice  
 # Show user current data in the list of product objects  
 # Let user add data to the list of product objects  
 # let user save current data to file and exit program*

Figure 1: Main body pseudo-code

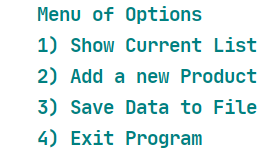


Figure : Option menu

The branched code was copied directly from previous assignments, except that I added a loop to deal with incorrect entries as seen in Figure 3.

*# Get user's menu option choice*strChoice = IO.input\_menu\_choice(**"Which option would you like to perform? [1 to 4] - "**) *# message*while strChoice not in str(**"1, 2, 3, 4"**):  
 strChoice = IO.input\_menu\_choice(**"Entry should be 1, 2, 3, or 4 - "**) *# message*

Figure : Code to enforce choices

## Create the Product Class

We were given three classes to begin with. The *Product* class is used to store data about a product. The class itself behaves as a template for one or more instances, known as objects.

Classes provide a means of bundling data and functionality together. Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state. (The Python Tutorial, <https://docs.python.org/3/tutorial/classes.html>, 2020) (External Site)

In this case, the attributes are the product’s name and its price. I want to control modifications to these variables, so I made them private. Doing so limits users’ ability to modify the attributes directly, but means I need to use a getter and setter to access the attributes as shown in Figure 4.

@property *# getter*def product\_name(self):  
 return str(self.\_\_product\_name).upper() *# Upper case*@product\_name.setter  
def product\_name(self, value):  
 if not str(value).isnumeric():  
 self.\_\_product\_name = value  
 else:  
 raise Exception(**"Product names cannot be numbers."**)

Figure : Getter and setter

According to GeeksforGeeks,

[Private variables in python](https://www.geeksforgeeks.org/private-variables-python/) are not actually hidden fields like in other object oriented languages. Getters and Setters in python are often used when:

* We use getters & setters to add validation logic around getting and setting a value.
* To avoid direct access of a class field i.e. private variables cannot be accessed directly or modified by external user. (<https://www.geeksforgeeks.org/getter-and-setter-in-python/>, 2020) (External Site)

## Complete Class Methods

The other two classes, *FileProcessor* and *IO*, are made up of static methods (@staticmethod). According to Real Python “a static method can neither modify object state nor class state. Static methods are restricted in what data they can access - and they’re primarily a way to namespace your methods.” ([https://realpython.com/instance-class-and-static-methods-demystified/ - static-methods](https://realpython.com/instance-class-and-static-methods-demystified/%20-%20static-methods), 2020) (External Site) In this case, the functions required were nearly identical to those from [Module 6](https://github.com/jaws522/IntroToProg-Python-Mod06/blob/master/ToDoList2.py) (External Site) and [Module 7](https://github.com/jaws522/IntroToProg-Python-Mod07/blob/master/ToDoList3.py) (External Site), so I copied them over and changed the variables as needed. There were some exceptions required to accommodate class objects rather than dictionaries.

Figure 5 shows the modification for appending the class object to the list table (the original append command is commented out).

@staticmethod  
def add\_data\_to\_list(product, price, list\_of\_products):  
 *""" Adds a new object row into a list of object rows  
  
 :param product: (string) name of product  
 :param price: (string) price of product  
 :param list\_of\_products: (list) you want filled with file data  
 :return: (list) of object rows, (string) status  
 """* list\_of\_products.append(Product(product, price))  
 *# lstTable.append({"Task": task, "Priority": priority})*

return list\_of\_products, **'Success!'**

Figure : Append objects to list

Figure 6 shows the modification for printing each row (product) from the list separately (the original print command is commented out).

@staticmethod  
def print\_current\_Products\_in\_list(list\_of\_products):  
 *""" Shows the current products in the list of object rows  
  
 :param list\_of\_products: (list) of products you want to display  
 :return: nothing  
 """* print(**"\*\*\*\*\*\*\* The current Products listed are: \*\*\*\*\*\*\*"**)  
 for row in list\_of\_products:  
 print(row.product\_name + **" ("** + str(row.product\_price) + **")"**)  
 *# print(row["Task"] + " (" + row["Priority"] + ")")*

print(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**)  
 print() *# Add an extra line for looks*

Figure : Print current products

# Running the script

To see whether the code functioned as intended, I first ran it in PyCharm. The data from products.txt is loaded and can be seen by entering option 1 as shown in Figure 7.

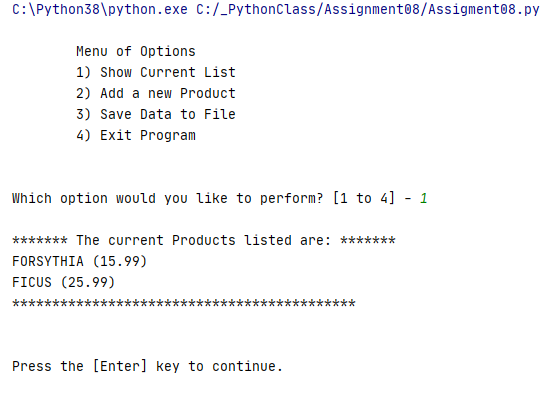


Figure : Show current data - PyCharm

I then added a product using option 2 and showed the current list again (see Figure 8).

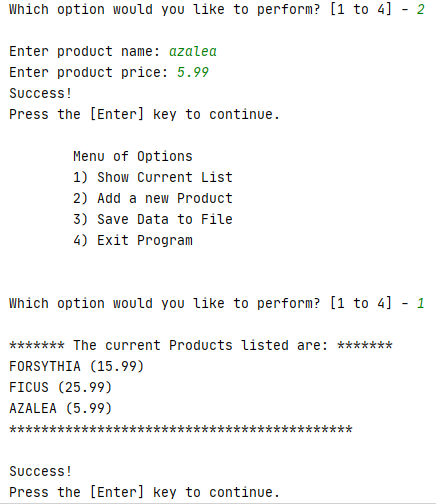


Figure : Adding a product - PyCharm

Then I entered option 4 to exit the program.

Finally, I ran the program from the command window. I added a product and saved the data to the file as shown in Figure 9.

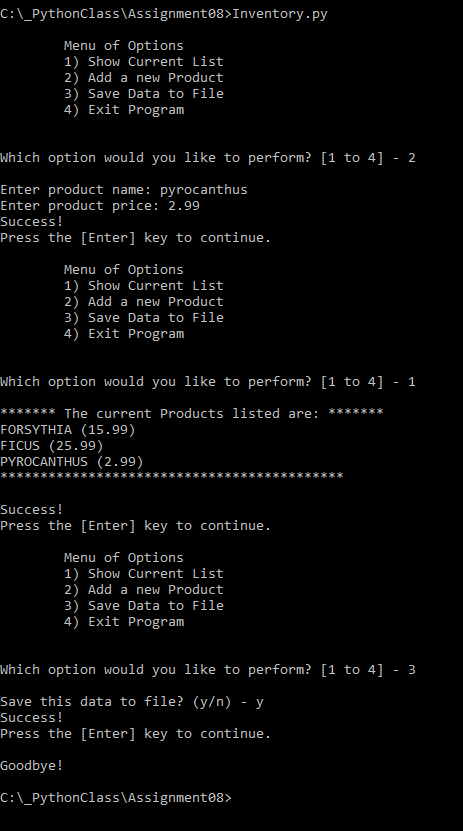


Figure : Run program – cmd

# Summary

Using the textbook, some additional websites, and the Module 8 documentation, I created the ‘Inventory’ program using classes and objects, and successfully ran it in PyCharm and the OS command window.