Naomi Martin

August 31, 2022

Assignment 08

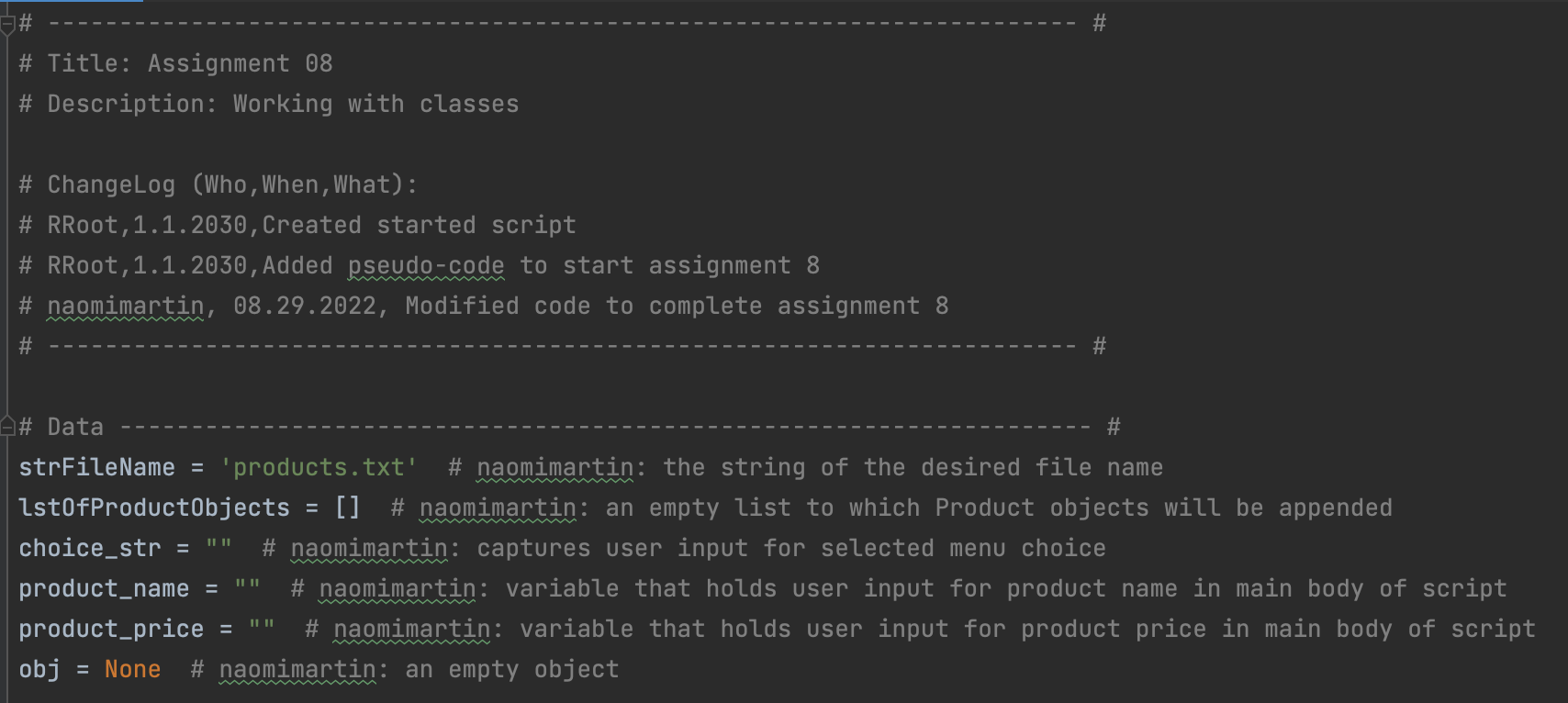
**Classes and Objects: The Product Object List Program**

**Introduction**

This assignment was intended to give us an introduction to object-oriented programming, and to extend our knowledge of classes and functions to create software objects complete with their own defined methods and attributes. Object-oriented programming is a widely used modern software methodology, a knowledge of which is essential to programming. In this document, I will go through my code for the Product Object List program, using new concepts of classes and objects, while incorporating previously learned concepts of error handling to complete a comprehensive program. I will also discuss some difficulties I had encountered in completing this assignment.

**Writing the Program: Header**

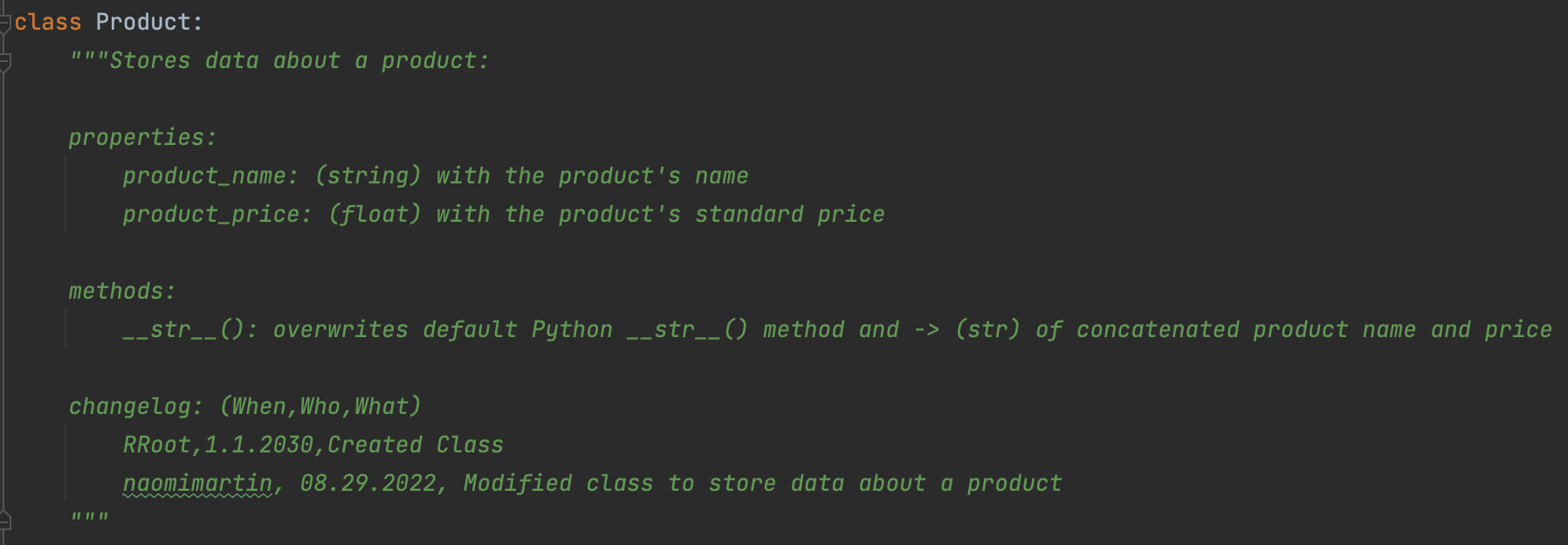
The first thing that I did, as should be done with any code, was to update the program’s change log in the header, to reflect that I made changes to the code on the given date. I also made sure to remember to include my name on all my comments in the code, in order to differentiate the comments that I made from Professor Root’s comments. The top of the program also has variables declared, to which I added as I was writing my code (figure 1).



**Figure 1.** The updated header and variables declared at the top of the program.

**Writing the Program: The Product Class**

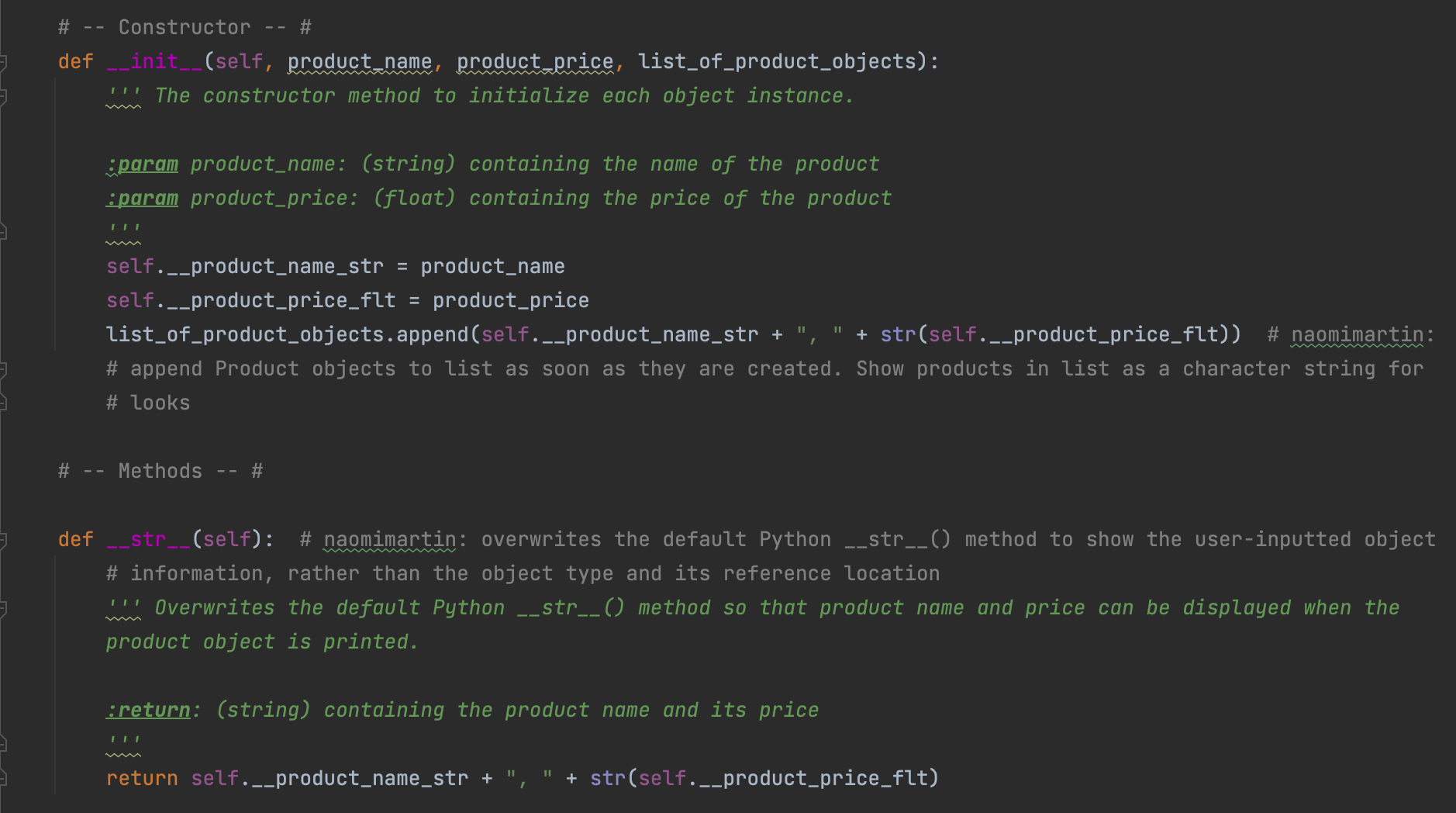
The product class contains information about a product, and contains code that initializes a product object with its own attributes and methods. The product class docstring shows information about these methods (figure 2).



**Figure 2.** Product class docstring.

The properties, or getter functions, product\_name and product\_price, provide indirect access to the private attributes of the same name, so that these attributes can be set in a safe way. I will go over this briefly. I also wrote another \_\_str\_\_() method for this class just as Professor Root did, to overwrite the default Python \_\_str\_\_() method to allow for the product object attributes (name and price) to be printed, rather than reference address location of the object. I wrote this so that the product object attributes can be displayed to the user.

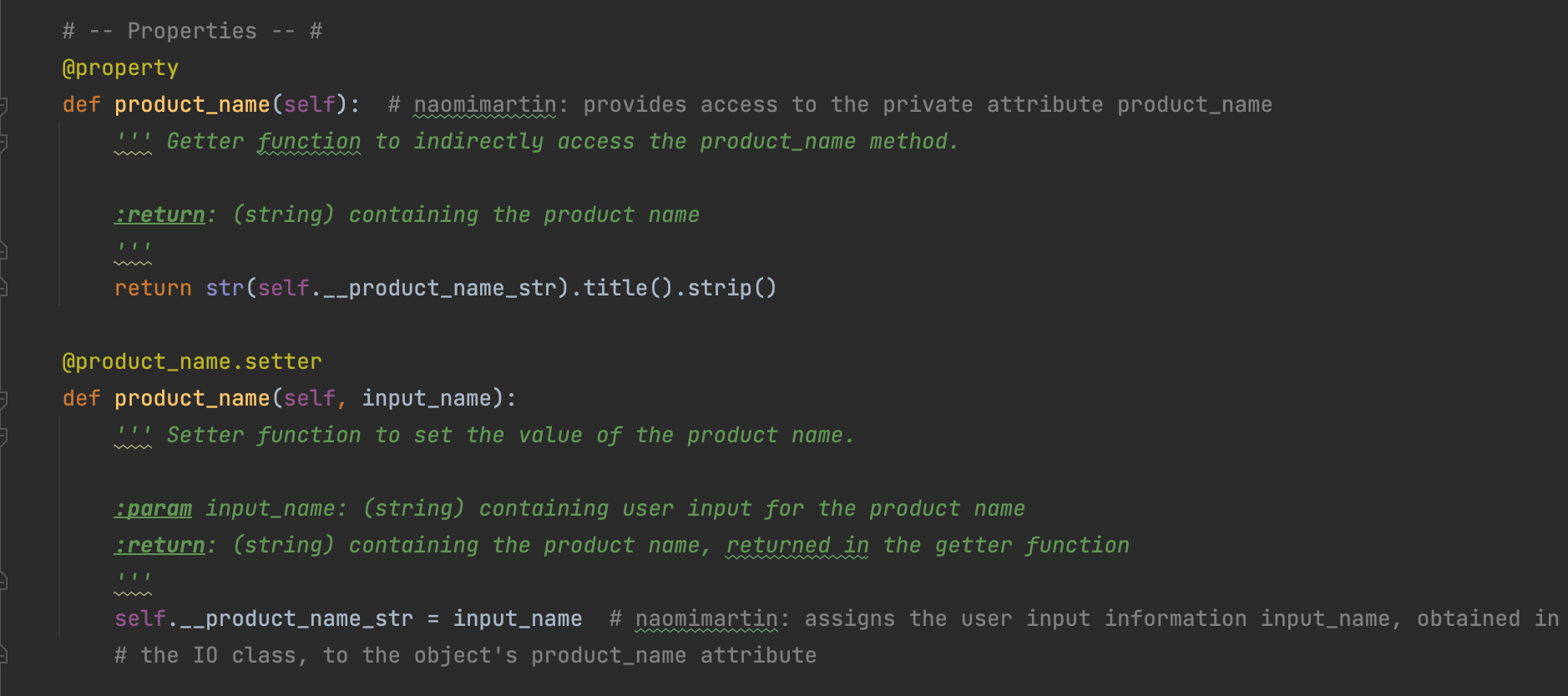
The figures below contain the code for the main body of the Product class (figures 3–5).

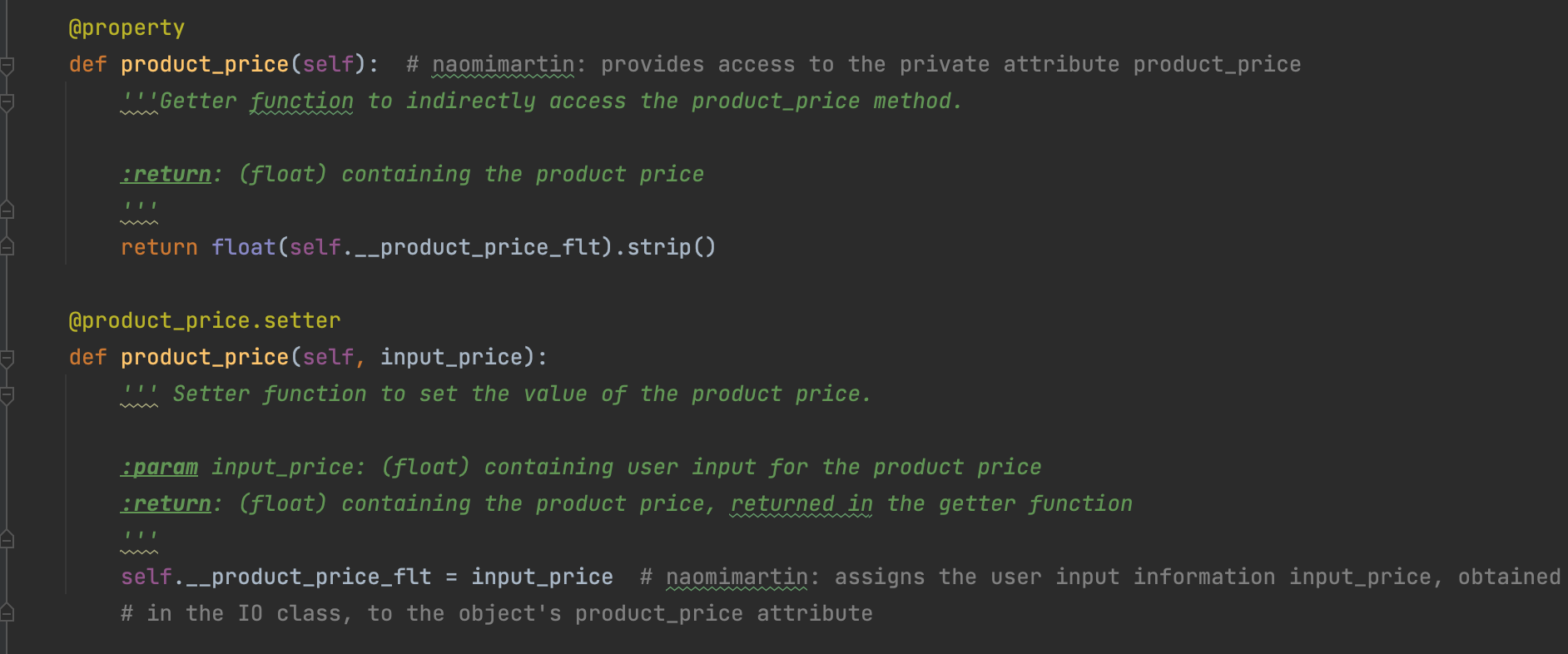


**Figure 3.** First part of the main body of the Product class, containing code for the constructor and methods of the class.

The constructor initializes each object when the Product class is called, with its own set of attributes. The parameters for the constructor are its attributes *product\_name* and *product\_price*, and the *list\_of\_product\_objects*. I included the list as a parameter to that I could append each object to the list automatically as soon as each object is created. I chose to write the constructor this way to ensure that the created object is always appended. I declared one empty “obj” variable at the beginning of the code, so every time that the program is run, the object is overwritten with a new object. I did not want to create a new object for each user-inputted product, especially since we don’t know how many objects the user wanted to create. Therefore, I figured that overwriting each object after appending its data to the list was a memory-efficient way to execute the program. The object information is appended to the list in the form of a character string for looks, to omit the quotation marks normally around the product name that indicates that the name is a string.

The methods section defines the new \_\_str\_\_() method that overwrites the default Python method of the same name, to display object data in a more intuitive way. When it is called, the new \_\_str\_\_() method displays the object’s product name and product price in the form of a character string, to show the user information about the most recently inputted object. This is in contrast to the default Python method, which displays the object’s reference address location.



****

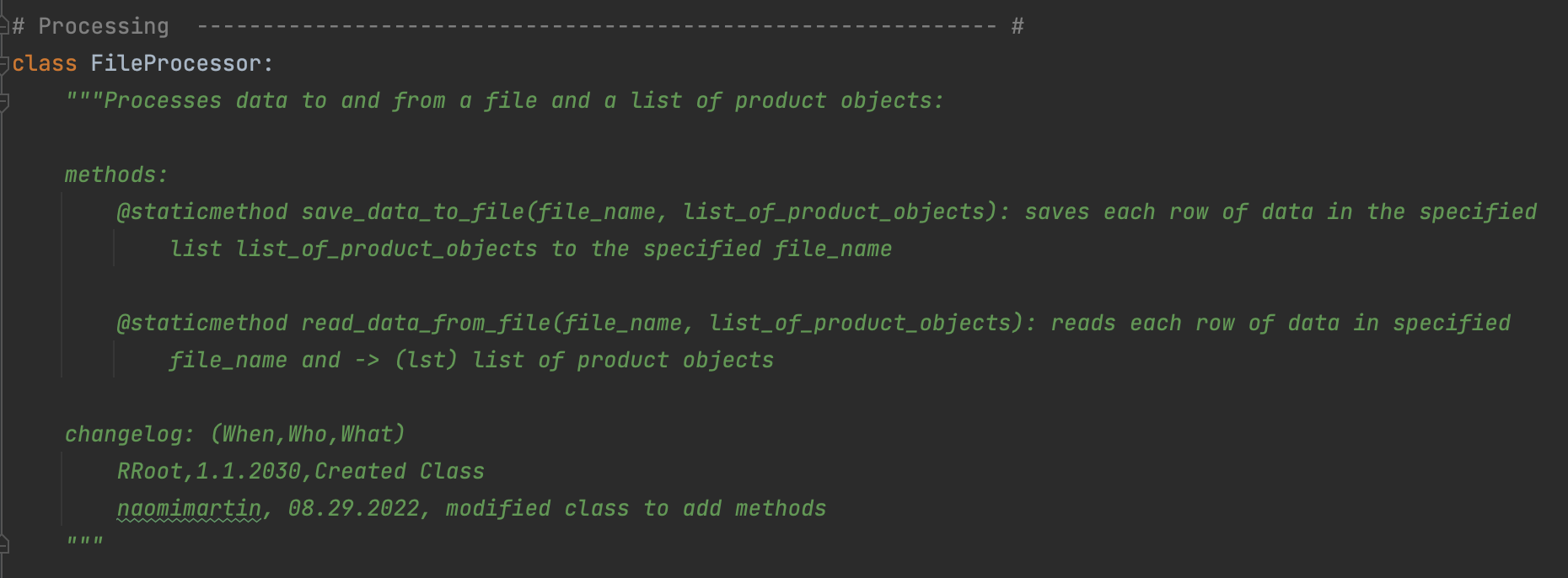
**Figure 4.** Second part of the main body of the Product class, containing code for the product\_name and product\_price getter and setter functions.

The properties section defines the getter and setter functions for an object’s product\_name and product\_price. The getter functions allow indirect access to the private attributes product\_name and product\_price, private as indicated by the double underscore prefix on these attributes. By providing indirect access, we can set these attributes in a safe way, in accordance with object encapsulation. The getter functions then return the private attribteus for later use in the program. The setter functions simply assign the user input for name and price to the attributes product\_name and product\_price. I struggled for a while figuring out how I wanted to write the code for this section, as I wanted to capture user input and test that their input was valid (i.e. the product name only contains alphabetical characters, and the product price only contains numeric characters). As this was the first section that I tackled the code for (and not the IO class below), I tried to write the user input functions within the Product class. I attempted to write a static method in the Product class to allow for the validation of the user input before each object was created and assigned a name and price. I also tried to use the templates that Professor Root had used in his module08 notes, by testing for user input validity within the setter functions themselves. I cannot recall the exact pitfalls that I experienced, but through much experimentation and trial and error, I decided that the best way to do this would be do request user input in the IO class and test for input validity there. Only after the input has been validated can the input be used as the product name and price in the Product class. This was a better way to do it than testing for user input validity within the setter functions themselves, as it allows for a better separation of concerns. It also simplifies the code for the product name and price setter functions, by simply assigning the product name and price attributes the value of the user input obtained in the IO class.

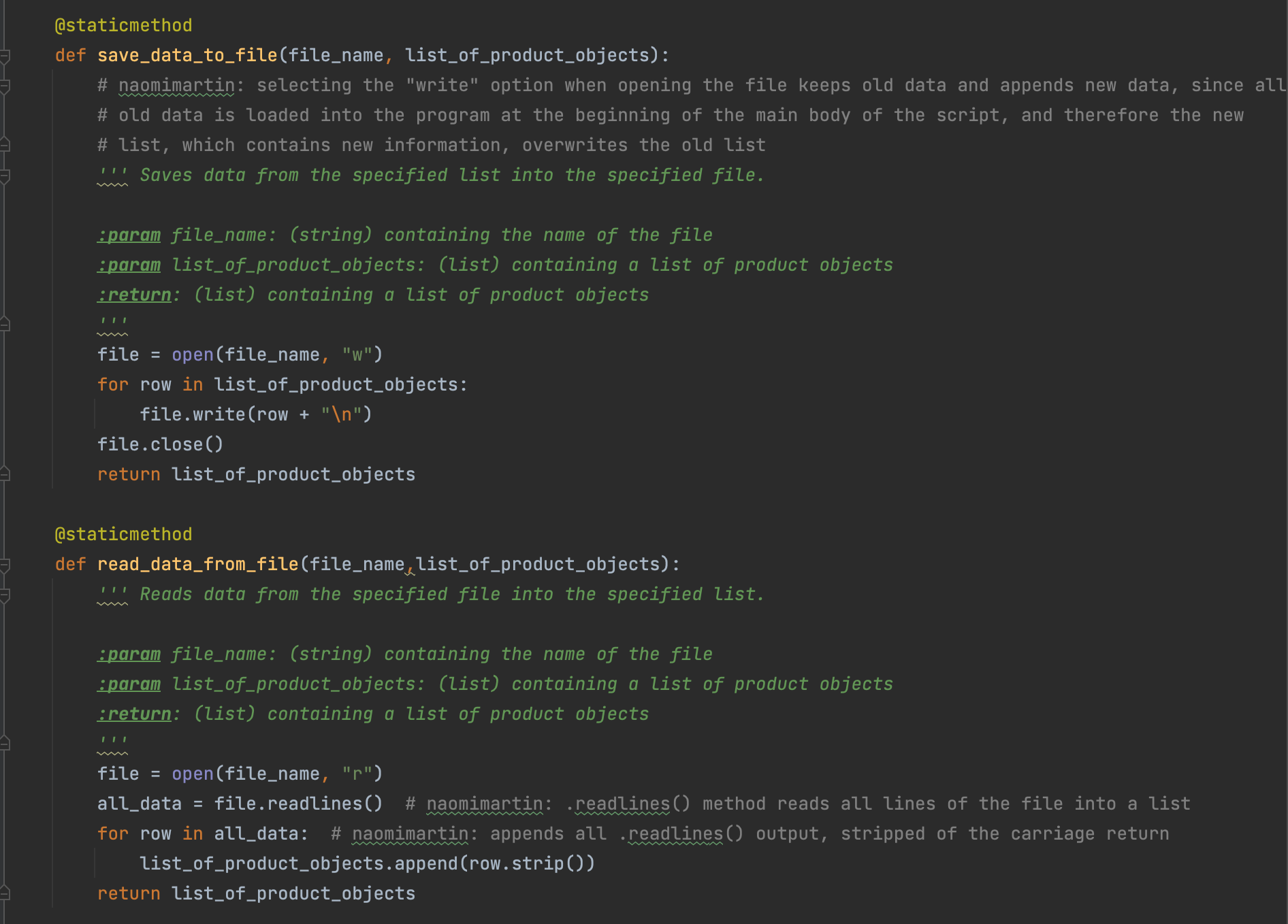
None of the methods written in this Class are static methods, as they all apply to the objects created. Therefore, they all take in the parameter *self*. The constructor method takes in the additional parameters *product\_name*, *product\_price*, and *list\_of\_product\_objects*, to assign each object a name and a price and append each object to the list. The product\_name() setter method takes in an additional parameter *input\_name* to assign the user-inputted product name to the product object, and the product\_price() setter method similarly takes in an additional parameter *input\_price* to assign the user-inputted product price to the product object.

**Writing the Code: The FileProcessor Class**

The FileProcessor Class was relatively simple to write code for; the class only contained two functions to save data to a file, and to read data from a file (figure 5).



**Figure 5. a)** Docstring for the FileProcessor class.



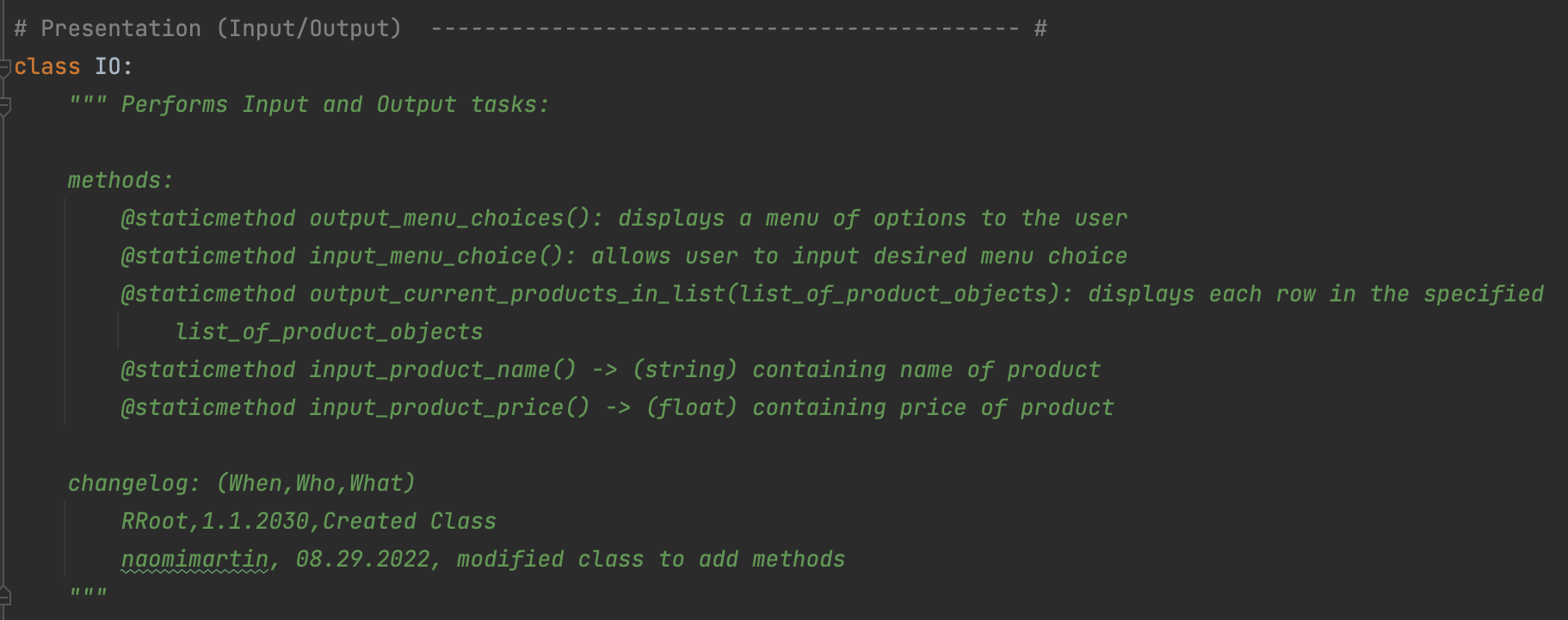
**Figure 5. b)** Main body of code for the FileProcessor Class.

Both definitions are static methods, as they are executed for the entire class without the need for initializing objects. For the save\_data\_to\_file() method, which takes in parameters *file\_name* and *list\_of\_product\_objects*, I chose to write the data to the file each time, rather than append. My reasoning for this is as follows. At the beginning of the program, the data from the file is loaded into the list of product objects and displayed to the user. The user can then add more objects to the list throughout the program by entering in the names and prices of various products. These product objects are appended to the list, and the list, which contains both the previous list as well as newly added product objects, then overwrites the previous data in the file. The end result is that the file contains both previously recorded product object data and newly added product object data. Each row of the list is written into the file with a carriage return at the end, so that the rows aren’t jumbled into one line. The list is returned at the end of the method so that it can be accessed again for later use in the program.

The read\_data\_from\_file() method takes in the parameters *file\_name* and *list\_of\_product\_objects*. I use the “r” access mode to read in the data from the file, and use the .readlines() method to read each line of the file into the variable *all\_data* as a list. For each row in this variable *all\_data*, each row is appended to the *list\_of\_product\_objects* list. The list is returned at the end of the method so that it can be accessed again for later use in the program.

**Writing the Program: The IO Class**

The IO class was also relatively simple to write, as it used IO methods that we had written in previous homework assignments. The figure below shows the docstring for the IO class, and indicates that each method in the class is a static method (figure 6).



**Figure 6.** The docstring for the IO class indicates that each method is a static method.

The code for the IO class was quite long; the figure below shows the code for the first part of the IO class (figure 7).



**Figure 7.** First part of the code for the IO class.

The output\_menu\_choices() method simply prints the menu of options for the user to select from. I chose to write my program so that the user has the option to view current products in the list, rather than print the current products in the list each time an input is given, as was done in previous assignments.

The input\_menu\_choice() method allows the user to input a number corresponding to the menu option they want to perform. The user choice is captured in the *choice* variable. I later write code that handles the situation in which a valid menu option is not selected: the program displays a message to the user to select a valid menu option, and the menu is shown again.

The output\_current\_products\_in\_list() takes in the parameter *list\_of\_product\_objects*, and prints each row in the specified list.

The figure below shows the code for the second part of the IO class (figure 8).



**Figure 8**. Code for the second part of the IO class.

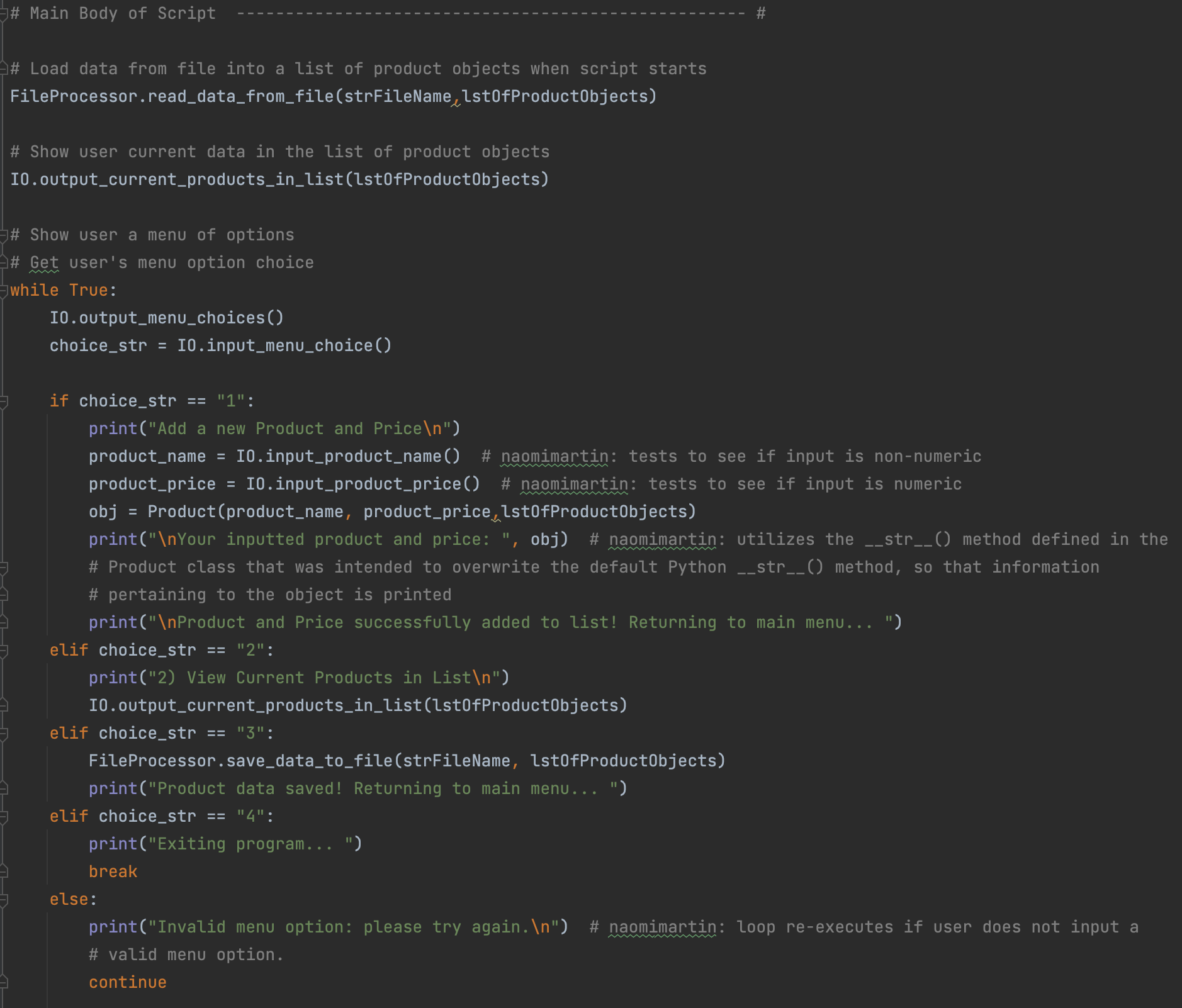
The input\_product\_name() static method contains code for the user to input the product name. I chose to write it so that if a character string was not inputted, i.e. if a numerical value was inputted, the program alerts the user of the invalid input and asks them again for valid input. I do this with a while loop. If a numerical input was given, an exception is raised informing the user that only a character string is valid for this input. If the exception is not encountered, i.e. if no numerical input was given, the user input is stored in the *name* variable, and the while loop breaks to indicate that valid input was received. The *name* variable is then returned, so it can later be assigned to the product object when it is created.

Similarly, the input\_product\_price() static method contains code for the user to input the product price. Here, if a number value is not inputted, the program uses the same method to ask the user for a valid number. Valid input is converted into a floating-point value and stored in the *price* variable, which is then returned so it can later be assigned to the product object when it is created.

These methods were those that I had originally attempted to write in the Product class as static methods, but decided to move to the IO class to maintain better separation of concerns.

**Writing the Program: Main Body of Script**

Once all the classes and functions were written, it was relatively straightforward to compile them all into the main body of the script (figure 9).



**Figure 9.** Code for the main body of the script.

First, the read\_data\_from\_file() method from the FileProcessor class is called to read data from the specified file name into the specified list of product objects. I then call the output\_current\_products\_in\_list() method from the IO class to show the user at the beginning of the program what the current recorded products are. Next, a while loop is written to allow the program to engage with the user until they decide to exit. At the beginning of the loop, the menu choices are shown to the user with the output\_menu\_choices() method from the IO class, and user input is requested with the input\_menu\_choice() method from the IO class. These two methods are called each time a menu option is complete, prompting the user for the next menu option selection.

If menu option 1 is selected, input\_product\_name() and input\_product\_price() methods from the IO class are called, and assigned to their respective variables product\_name and product\_price, which are declared at the beginning of the script. An object is then created and stored in the obj variable, which is also declared at the beginning of the script. The object is created by calling the Product class with parameters *product\_name*, *product\_price*, and *lstOfProductObjects*, as this automatically calls the \_\_init\_\_() method of the Product class. The specified *product\_name* and *product\_price* attributes are assigned to the object, and the object information is appended as a character string to the list. I then print the information that the user had just inputted, to demonstrate the use of the \_\_str\_\_() method I created in the Product class, to overwrite the default Python \_\_str\_\_() method. A message is then printed that tells the user their information has been added to the list.

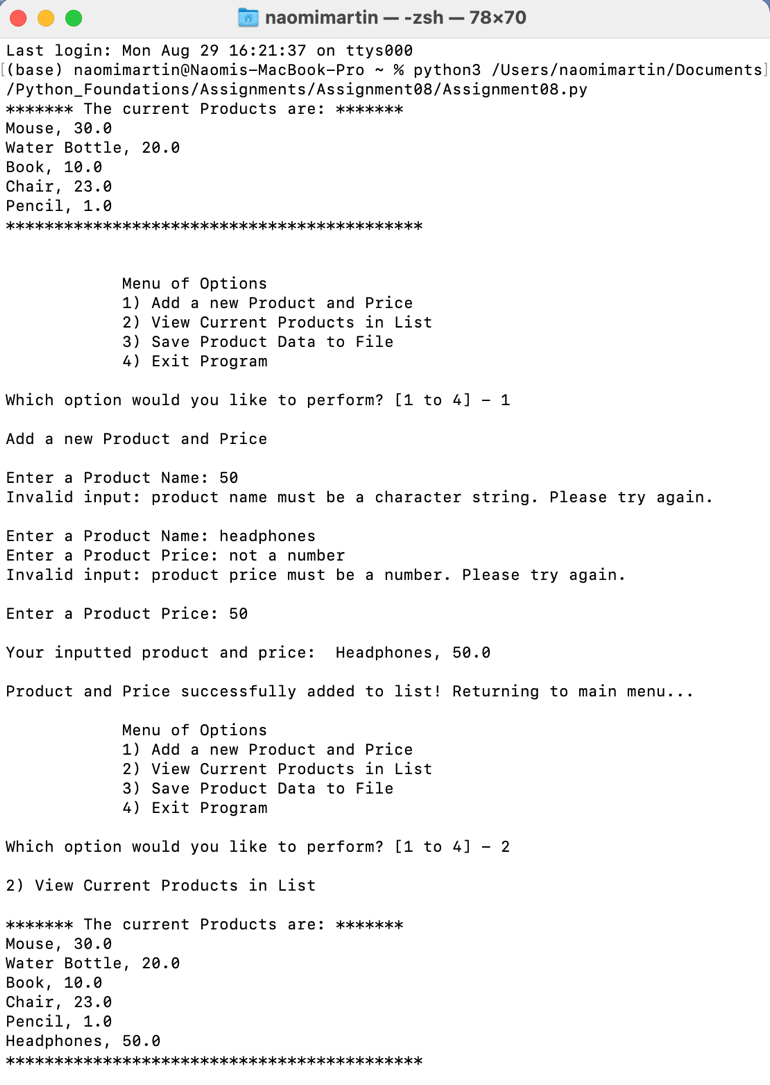
If menu option 2 is selected, the output\_current\_products\_in\_list() method from the IO class with the *lstOfProductObjects* parameter is called to print the contents of the *lstOfProductObjects* list to the screen. This shows the user all of the product objects that they had recorded.

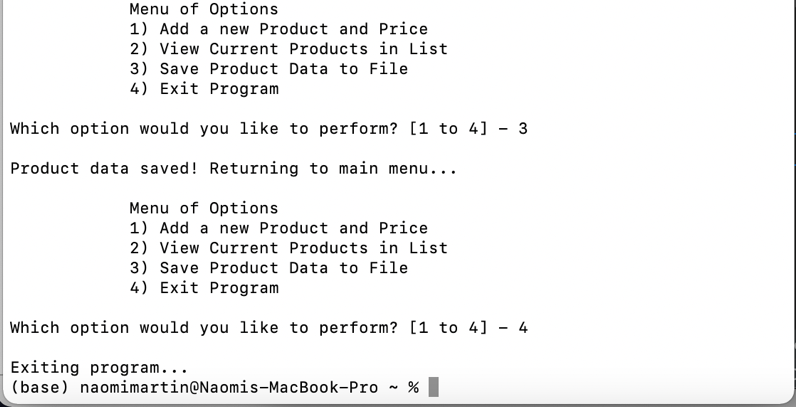
If menu option 3 is selected, the save\_data\_to\_file() method from the FileProcessor class with the specified file name and list is called to save the data from the *lstOfProductObjects* list to the file. As mentioned before, data in the file is overwritten, but as data was read into the list at the beginning of the script, overwriting this data results in no data being lost.

If menu option 4 is selected, the program exits by breaking the while loop. If none of the above options are selected, a message is printed that an invalid menu option was selected, and the while loop re-executes to attempt to obtain valid user input.

**Testing the Program: MacOS Terminal Command Line**

Once the program had been completely written and tested in PyCharm, I tested it out in the MacOS Terminal Command Line to ensure proper functionality. The figure below shows the results of this test (figure 10).



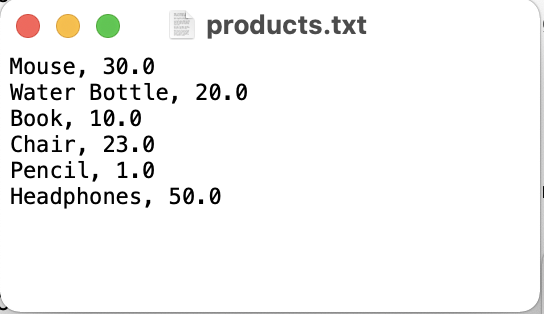


**Figure 10.** Successful run in the MacOS Terminal Command Line.

At the beginning of the program, data from the text file is read in. In this instance, it loaded in previous data that I had inputted during my PyCharm test runs. I will clear the contents of the text file upon submission of my assignment, so you can start the program fresh, and see how overwriting the file with the list data results in old data being saved and new data being added.

I chose to demonstrate my exception handling by inputting the number “50” in the “Enter a Product Name:” input field. As expected, an exception is raised and the program tells me to enter a valid character string. I then demonstrate exception handling in the “Enter a Product Price:” input field, by inputting the character string “not a number”. As this cannot be turned into a floating-point value, the program alerts me that a number must be inputted. Upon successful input in these two fields, the product object is created, and printed back to me. I can see that the product name and price of the object is printed back to me, rather than the object’s reference location address. This shows that my custom \_\_str\_\_() method written in the Product class worked.

I then save the data by choosing menu option 3, and check the text file to ensure that my data was properly saved (figure 11). All looks well and the program is complete.



**Figure 11.** The text file after the run of my program in the MacOS Terminal Command Line

**Struggles**

In previous assignments, some of the primary struggles I encountered involved proper syntax, thinking of the most efficient way to code for a specific problem, or being stuck on a specific error message. Somewhat differently, the primary struggle I encountered on this assignment involved the mere understanding of classes and objects. Like all things, it takes a bit of time and practice to fully grasp a concept, but I felt as if my understanding of classes and objects took longer to settle in (it is still confusing for me). Some of the main concepts I struggled with or as follows:

1. Determining when to write functions for a specific object as opposed to the entire class.
2. Figuring out the proper syntax and variable names for private attributes, especially for their access outside of their defined functions.
3. Becoming comfortable with the proper syntax for parameters inside functions and variables outside functions in general, and maintaining consistency between the two.
   1. This would probably become more clear once I use functions for applications to a variety of variables, rather than just one variable as we have been doing.
4. Becoming comfortable with the definitions of *methods* and *attributes*.
   1. These terms make sense when I think a little bit about them, but I would like to reach a level of comfort where I understand them as soon as I see them.
5. Overall, becoming comfortable with the various variables and parameters defined within and without function, class, method, and attribute definitions.

I found the textbook especially to be very helpful in explaining the basic concepts of objects with their demonstration through the simple Critter Caretaker Program. It was just a matter of applying these concepts to my assignment that I found difficult. Although the concepts of objects and classes seem difficult to me now, I believe that further practice, especially through writing programs of my own, will allow me to become more comfortable with them. Additionally, once I had completed the main body of the code for the Product class which involves initializing objects, I found writing definitions for functions in the other classes relatively straightforward, showing progress in understanding.

**Conclusion**

As with any newly learned concept, it will take some time and practice before I fully understand it and am comfortable with manipulating it. I found this assignment to be challenging, due to these new concepts, but overall went quicker as it built off of previous code that we had already written. I feel as if I benefitted greatly from this assignment not only in learning a new concept, but also because it offered great practice with functions to streamline the main body of the code. I will have to continue to work with parameter and variable names within and without functions, but overall, I feel more comfortable with functions and classes after this assignment.