Alexander Bryant

6/7/2020

Foundations of Programming

Assignment 08

https://github.com/abryant-seattle/IntroToProg-Python-Mod08

Managing the Product List

Intro

This week's module expanded on creating and using classes and class objects. The assignment this week was to write a program with several new classes that help a user manage a product list, while including file processing features.

Process

```python
# Data ---------------------------------------------------------------------- #

strFileName = 'products.txt'  # A string that holds the relevant filename
menu_option = ''  # A string that holds the user's menu choice
lstOfProductObjects = []  # A list that holds the Product objects while program is running


class Product:
    """Stores data about a product:

    properties:
        product_name: (string) with the products's name
        product_price: (float) with the products's standard price
    methods:
        __init__(self, product_name, product_price): a constructor method to create Product objects
    changelog: (When,Who,What)
        RRoot,1.1.2030,Created Class
        ABryant,6.5.2020, Added constructor to Product Class
    """
    pass

    # TODO: Add Code to the Product class
    # --Constructor --
    def __init__(self, product_name, product_price):
        self.str_product_name = product_name
        self.str_product_price = product_price
```

Figure 1: The data section includes the Product class and the global variables.

The script begins with a data portion outlining the global variables and the first class, *Product*. *Product* class objects hold information about a product, with the attributes *product_name* and *product_price*. A constructor method creates new Product objects with a user supplied name and price.

```
    # TODO: Add Code to process data from a file
    @staticmethod
    def read_data_from_file(file_name):
        temp_list_of_products = []
        active_file = open(file_name, "r")
        for row in active_file:
            row = row.strip('\n')
            temp_list_of_products.append(row)
        active_file.close()
        return temp_list_of_products

    # TODO: Add Code to process data to a file
    @staticmethod
    def save_data_to_file(file_name, list_of_product_objects):
        active_file = open(file_name, 'w')
        for row in list_of_product_objects:
            active_file.write(str(row) + '\n')
        active_file.close()
        print('Products have been saved.')
        return
```

Figure 2: The FileProcessor class handles reading and writing data.

The processing section includes another new class, *FileProcessor*. In ***Figure 2*** above, two new methods are created, *read_data_from_file* and *save_data_to_file*. The reading method is supplied a file name parameter, and the file contents are returned as a list, which becomes the working product list *lstOfProductObjects* which the user interacts with. The saving method takes this same list and writes it to a new file as strings using a for loop, returning nothing.

```
    # TODO: Add code to show menu to user
    @staticmethod
    def menu_options():
        print('\nMenu options: \n 1 - Review product list\n 2 - Add product\n 3 - Remove product\n 4 - Save and quit')
        return

    # TODO: Add code to get user's choice
    @staticmethod
    def menu_input():
        temp_menu_choice = int(input('Please select a menu option: '))
        return temp_menu_choice
```

Figure 3: The IO class contains input and output, presentation focused methods.

The presentation section includes the new *IO* class, which manages presentation related methods. In ***Figure 3*** above, the static method *menu_options* receives no parameters and presents a menu to the user. The static method *menu_input* receives no parameters and requests the user to select a menu option.

```
    # TODO: Add code to show the current data from the file to user
    @staticmethod
    def review_products():
        if len(lstOfProductObjects) == 0:
            print('No products have been added yet.')
        else:
            for row in lstOfProductObjects:
                print(row)
        return

    # TODO: Add code to get product data from user
    @staticmethod
    def add_products():
        name_input = str(input('Please enter product name: ')).lower()
        price_input = float(input('Please enter product price: '))
        new_product = Product(name_input, price_input)
        lstOfProductObjects.append((new_product.str_product_name + ', $' + str(new_product.str_product_price)))
        print('Product has been added.')
        return
```

Figure 4: The methods above allow the user to review and add to the product list.

In ***Figure 4*** above, the *review_products* method allows the user to review any products in the product list. If the program initializes and reads an empty file, a confirmation message lets the user know there are no products. This method takes no parameters. The *add_products* method allows the user to add products through several prompts, which then append the information to the working product list, *lstOfProductObjects*. This method takes no parameters.

```
@staticmethod
def remove_products():
    product_to_remove = str(input('Enter the product name to be removed: ')).lower()
    row_counter = 0
    for row in lstOfProductObjects:
        if product_to_remove in row.split(','):
            lstOfProductObjects.remove(row)
            print('"' + product_to_remove + '" has been removed.')
            row_counter += 1
        elif row_counter > 0:
            break
    if row_counter == 0:
        print('"' + product_to_remove + '" not found in product list.')
```

Figure 5: The remove_products method allows the user to remove products from the list.

In ***Figure 5*** above, I added an additional feature that allows the product to remove products from the product list. The static method *remove_products* uses a for loop and a counter to remove products that are no longer wanted. The counter prevents the loop from printing a 'product not found' message for every row in the product list that doesn't match the user submitted *product_to_remove*, which could be very annoying on a longer list. The user may receive one of two confirmation messages depending on if the submitted product name was found and removed or not.

```
list_file = open(strFileName, 'a')
list_file.close()
lstOfProductObjects = FileProcessor.read_data_from_file(strFileName)
```

Figure 6: The script initializes by checking for a product.txt file and creates it if necessary.

Finally, the main body of the script begins by opening or creating a *product.txt* file and loading it into the working product list, *lstOfProductObjects*. This process is visible in ***Figure 6*** above.

```
# Show user a menu of options
while True:
    IO.menu_options()

    # Get user's menu option choice
    try:
        menu_choice = IO.menu_input()
    except ValueError as e:
        print('**Invalid Entry**\nPlease select a valid menu option.')
        menu_choice = ''

    # Show user current data in the list of product objects
    if menu_choice == int(1):
        IO.review_products()

    # Let user add data to the list of product objects
    elif menu_choice == int(2):
        try:
            IO.add_products()
        except ValueError as e:
            print('\n**Invalid Entry**\nProduct names must be alphanumeric.\nProduct prices must be numeric.')
            print('Neither field may be empty.')
```

Figure 7: Try-Except blocks are used to prevent the code from crashing if the user tries to submit an invalid menu option.

In ***Figure 7*** above, the main body of the script exists mostly inside of a while loop. After the menu prints out, the user is prompted to make a selection. For the purposes of demonstrating a try-except block, I specified that the menu options must be integers. If a user submits a character that cannot be interpreted as an integer, an error message is displayed and the *menu_choice* variable is reset to a blank string. Without this, after receiving the error message, the program would automatically proceed to the user's last valid menu option, an experience likely to be very confusing for a user. For menu option 1, the script launches the *review_products* method. For menu option 2, the script attempts the *add_products* method. While the user can submit any valid string for the name, the price must be interpretable as a float. If the user submits any invalid or empty submissions, a *ValueError* is caught and reminds the user of rules on submitting product names and prices.

```
# extra: let the user remove products
elif menu_choice == int(3):
    IO.remove_products()

# let user save current data to file and exit program
elif menu_choice == int(4):
    confirm_save = str(input('Ready to save and quit? Y/N: ')).upper()
    if confirm_save == 'Y':
        FileProcessor.save_data_to_file(strFileName, lstOfProductObjects)
        break
    elif confirm_save == 'N':
        print('Products have not been saved.\nReturning to menu.')
    else:
        print('**Invalid Selection**\nProducts have not been saved.\nReturning to menu.')

else:
    print('**Invalid Entry**\nPlease select a valid menu option.')
```

Figure 8: Removing products and saving the product list with menu options 3 and 4.

In ***Figure 8*** above, menu option 3 launches the *remove_products* method. Finally, menu option 4 launches the *save_data_to_file* method once the user confirms they are ready to save. While the try-except block in ***Figure 7*** prevents the user from selecting a non-integer menu option, a final else statement prevents the user from selecting any integers that aren't the explicitly listed options 1, 2, 3 or 4.

```
Menu options:
 1 - Review product list
 2 - Add product
 3 - Remove product
 4 - Save and quit
Please select a menu option: 1
cherries, $1.99
peaches, $0.99

Menu options:
 1 - Review product list
 2 - Add product
 3 - Remove product
 4 - Save and quit
Please select a menu option: 2
Please enter product name: apples
Please enter product price: 2.99
Product has been added.
```

Figure 9: A PyCharm user reviews and adds to the product list.

In ***Figure 9*** above, menu option 1 calls the *review_products* method to allow the user to review the existing product list. The cherries and peaches on the list were read in from a file written in a previous session. Menu option 2 calls the *add_products* method which lets the user add apples to the product list. These changes will be held in memory until the user saves later.

```
Menu options:
 1 - Review product list
 2 - Add product
 3 - Remove product
 4 - Save and quit
Please select a menu option: MENU OPTION 3 PLEASE
**Invalid Entry**
Please select a valid menu option.

Menu options:
 1 - Review product list
 2 - Add product
 3 - Remove product
 4 - Save and quit
Please select a menu option:
```

Figure 10: A command prompt user tests the limits of the try-except block.

In ***Figure 10*** above, the command prompt is used to run the product list program. While submitting the integer *3* would work properly, the string provided is caught in the try-except block from ***Figure 7***, as the string submitted is not interpretable as an integer. In ***Figure 11*** below, the apples added by the PyCharm user are visible in a later session from the command prompt.

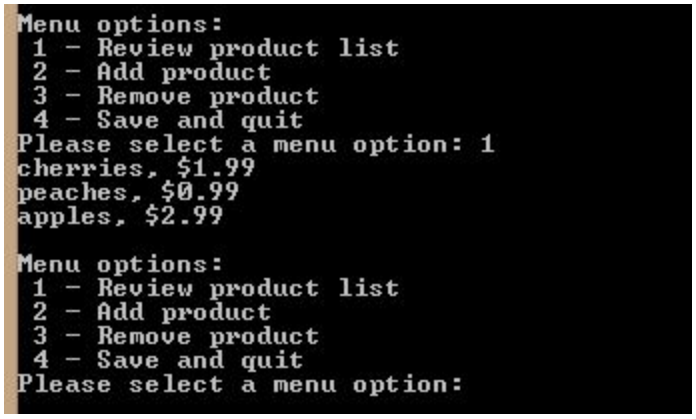

Figure 11: Once the product list is saved, it is accessible in later sessions.

Summary

This assignment emphasized how using classes effectively can help create a very clean and concise program. By using separate classes that hand off parameters through methods when necessary, the data, processing, presentation, and main body of my script are nearly all but separated, apart from a save confirmation prompt within the main body section. The *FileProcessor* class made it much easier to write and read from text files in a neat, straightforward manner.