Kathleen Wong

December 10, 2021

Intro to Programming

Assignment 08

https://github.com/kathleen101/IntroToProg-Python-Mod08

# Class

**Introduction**

This week, we learned about and how to set up classes, attributes, and properties. After the lessons learned, we were tasked with updating the starter script, applying the lessons learned. In this paper, I will be discussing the lessons learned and how I updated the script.

**Class**

Classes can essentially be seen as a blueprint. Through classes, objects created every time the class is called and returns an object (<https://www.w3schools.com/python/python_classes.asp>, External Site). If a programmer wants to create a class, they simply need to follow the syntax, writing “class Name(object):” As mentioned before, Randall had started the script, and it’s on us to finish the script. Randall had already created the class for us. In this class, he had created three different classes for the script: Product, FileProcessor, and IO. Product’s purpose is to vet the entry. FileProcessor is to interact with the text file. IO is to perform the input and output.

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:  
 changelog: (When,Who,What)  
 RRoot,1.1.2030,Created Class  
 Kathleen Wong,12.08.2021,Modified code to complete assignment 8  
 """*

class FileProcessor:  
 *"""Processes data to and from a file and a list of product objects:  
  
 methods:  
 save\_data\_to\_file(file\_name, list\_of\_product\_objects)  
 read\_data\_from\_file(file\_name): -> (a list of product objects)  
  
 changelog: (When,Who,What)  
 RRoot,1.1.2030,Created Class  
 Kathleen Wong, 01.09.2021,Modified code to complete assignment 8  
 """*

class IO:  
 *"""  
 a class to perform input and output  
  
 methods:  
 print\_options():  
 get\_option(list\_of\_rows):  
 input\_product\_data():  
  
 changelog:  
 RRoot, 1.1.2030, Created Class  
 Kathleen Wong, 12.10.2021, finished class to complete Assignment 08  
 """*

**Constructors**

Within the Product class, I decided to use the constructor (or initialization) method. I started this by using the constructor. This is an already method name recognized by Python, and by using it, Python knows it is my constructor method. It is run at the time of the object creation and is run as soon as an object of a class is instantiated. It’s useful to do any initialization you want to do with your object (<https://www.geeksforgeeks.org/__init__-in-python/>, External Site).

# -- Constructor -- #  
def \_\_init\_\_(self, product\_name: str, product\_price: float):  
 # -- Attributes --  
 self.\_\_product\_name = product\_name  
 self.\_\_product\_price = product\_price

**Properties**

Properties allow programmers to set up a getter and setter functions, allowing programmers to receive the data and prep the data to their preferences (<https://www.programiz.com/python-programming/property>, External Site). To set up a getter property, programmers simply need to start with @property and create their method. Every property needs to reference a “self.” To set up a setter property, programmers simply need to start with @method\_name.setter then create their method.

In my case, the Product class has two sets of getters and setters. The getter methods access the product name and price while the setter methods check to see if the entries meet the requirements. The product name setter verifies whether they are numeric and tell the user to try again if it is. The product price setter verifies whether they are numeric and tell the user to try again if it is not. It is important to note that the getter and setter properties need to stay together, meaning product name’s getter needs to stay with their setter. Otherwise, the script will not work as planned.

# -- Properties -- #  
# product\_name  
@property  
def product\_name(self): # getter or accessor  
 *"""  
 property: gets product\_name  
 setter: validates product\_name. if product\_name is numeric, script will tell user to try again  
 return: product\_name  
 """* # gets product name  
 return str(self.\_\_product\_name)  
  
@product\_name.setter # setter  
def product\_name(self, value):  
 if value.isnumeric() == True:  
 self.\_\_product\_name = value  
 else:  
 raise Exception("Product names cannot be numbers")  
  
# product\_price  
@property  
def product\_price(self): # getter or accessor  
 *"""  
 getter: get product price  
 #setter: set price value to product\_price if numeric. else, raise error message* ***:return****: product\_price  
 """* return float(self.\_\_product\_price)  
  
@product\_price.setter  
def product\_price(self, value):  
 try:  
 self.\_\_product\_price = float(value)  
 except Exception as e:  
 raise Exception("Not numeric. Try again.")

**Methods**

A method is essentially a function associated with an object (Python Programming Third Edition, External Site). In my case, I set up a method under a class, converting the entries to strings and formatting after the the properties’ vetting.

# -- Methods --  
def to\_string(self):  
 # convert results into string  
 return self.\_\_str\_\_()  
  
def \_\_str\_\_(self):  
 # formats product name and string as: name, price  
 return self.product\_name + ',' + str(self.product\_price)

**Static Methods**

Static methods do not require a class to be called directly from a class and do not require an object to be made first. Because I do not know whether or not the user will have a file already started and saved to their computer, I decided to use a static method. To start a static method, I simply needed to start my methods with @staticmethod. I wrote a few static methods under the FileProcessor class, so I would need to call on the FileProcessor class and method afterwards.

The save\_data\_to\_file method simply tries and opens the file to write the data into the file. Afterwards, it closes the file and informs the user it’s been saved successfully. Then, it returns the list. However, if it fails, it informs the user the attempt has failed and to try again.

The read\_data\_from\_file method reads the data already in the file, splitting up the data by commas and taking the first and setting it as the product name into the Product class and the second as the product price into the Product class then vetting the data and appending it to the list.

@staticmethod  
def save\_data\_to\_file(file\_name: str, list\_of\_rows: list):  
 *"""* ***:param*** *file\_name:* ***:param*** *list\_of\_rows (list):* ***:return****: list\_of\_rows (list) of data  
 """* try:  
 file\_obj = open(file\_name, "w") # opens file to write in  
 for product in list\_of\_rows:  
 file\_obj.write(str(product) + "\n")  
 file\_obj.close()  
 print("File saved successfully!")  
 except:  
 print("File did not save successfully. Try adding some data and try again!")  
 return list\_of\_rows  
  
@staticmethod  
def read\_data\_from\_file(file\_name: str):  
 *"""* ***:param*** *file\_name:* ***:return****: list\_of\_rows (list) containing all items in file  
 """* list\_of\_rows = []  
 try:  
 # reads file and writes data from file into list\_of\_rows  
 file\_obj = open(file\_name, "r")  
 for line in file\_obj:  
 product = line.split(",")  
 row = Product(product[0], product[1])  
 list\_of\_rows.append(row)  
 file\_obj.close()  
 except:  
 print("There is an error! Try adding some data first")  
 return list\_of\_rows

Under the IO class, I also have a few static methods set up. The print\_options method prints out the options available to the user. The get\_option method asks for the option the user wants to select. The show\_current\_data loops through each entry in the list of rows and prints out what is already in the list. The add\_product\_data asks for the product name and price input from the user and vets the data through the Product class.

@staticmethod  
def print\_options():  
 # prints options  
 print("""  
 Menu of options:  
 1. Show current data in list of product objects  
 2. Add to list  
 3. Save current data to file   
 4. Exit   
 """)  
  
@staticmethod  
def get\_option():  
 # Asks which option the user wants to select  
 option = str(input("Which option would you like to select? Pick from 1-4. "))  
 return option  
  
@staticmethod  
def show\_current\_data(list\_of\_rows: list):  
 # prints current data  
 print("\*\*\*\*\*\*\* The current items are: \*\*\*\*\*\*\*")  
 for row in list\_of\_rows:  
 print(row.product\_name, ",", str(row.product\_price))  
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
 print() # Add an extra line for looks  
  
@staticmethod  
def add\_product\_data():  
 *""" asks for product name and price and calls for class to vet the entry* ***:return****: vetted product name and price  
 """* try:  
 str\_name = str(input("What product do you want to add? ")).strip()  
 float\_price = float(input("How much is the product? "))  
 product = Product(product\_name = str\_name, product\_price = float\_price)  
 except Exception as e:  
 print(e)  
 return product

**Options**

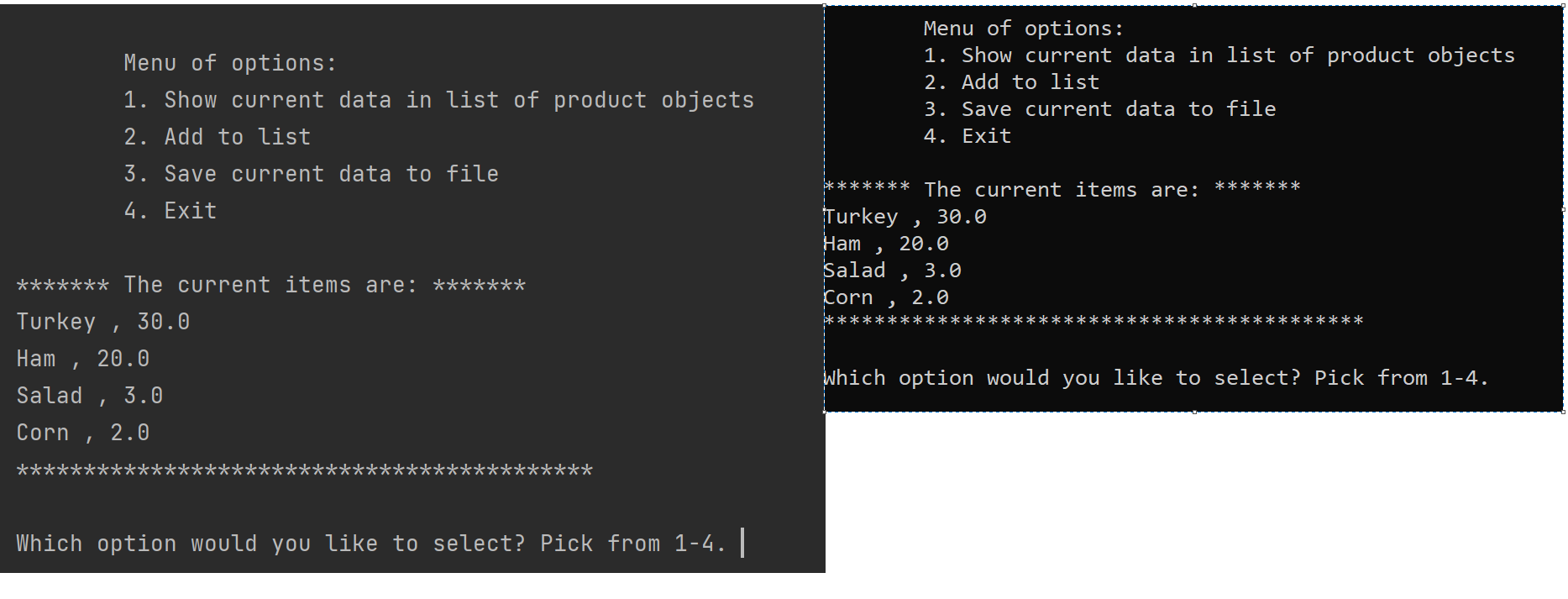
Similarly to previous assignments, we were given a list of options to provide the user and told to loop through each possible outcome. Because I cannot be sure I’ve captured every possible outcome, I decided to set up a try/except scenario, trying to read the data in the file and appending it to the list before providing the options. Under the try situation, it reads the options every time and shows the current data before asking for a selection. Afterwards, the user would loop through each possible option, which calls for previous classes and methods already made.

try:  
 """"  
 actions taken:  
 1. prints options  
 2. loops through file and saves it to list  
 3. asks for option  
 4. loops through options until 4 is selected  
 """  
 lstOfProductObjects = FileProcessor.read\_data\_from\_file(strFileName)  
 while True:  
 # printing options  
 IO.print\_options()  
 # printing items already in list  
 IO.show\_current\_data(lstOfProductObjects)  
 # getting option  
 option = IO.get\_option()

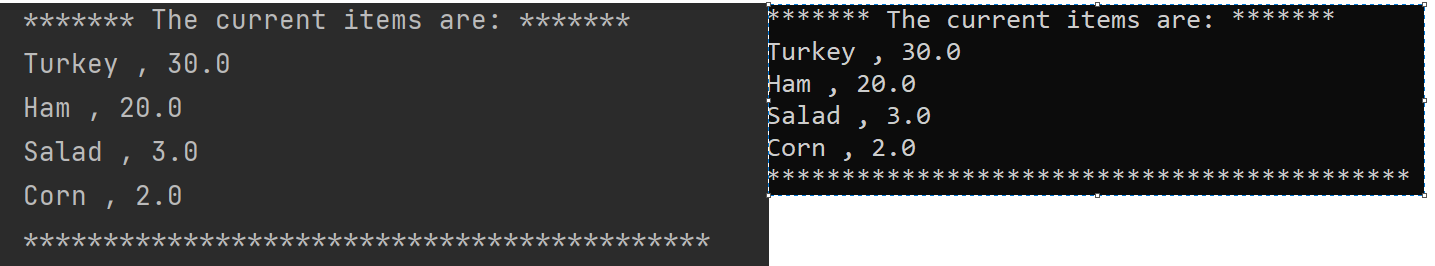
if option == "1":  
 # shows current data  
 IO.show\_current\_data(lstOfProductObjects)  
 continue  
  
 elif option == "2":  
 # adds data after going through Product class  
 lstOfProductObjects.append(IO.add\_product\_data())  
  
 elif option == "3":  
 # saves data to file  
 FileProcessor.save\_data\_to\_file(strFileName, lstOfProductObjects)  
  
 elif option == "4":  
 # ends  
 break  
except Exception as e:  
 # prints error message  
 print(e)  
 print("Error in file. Check for mistake")

**Testing the Code**

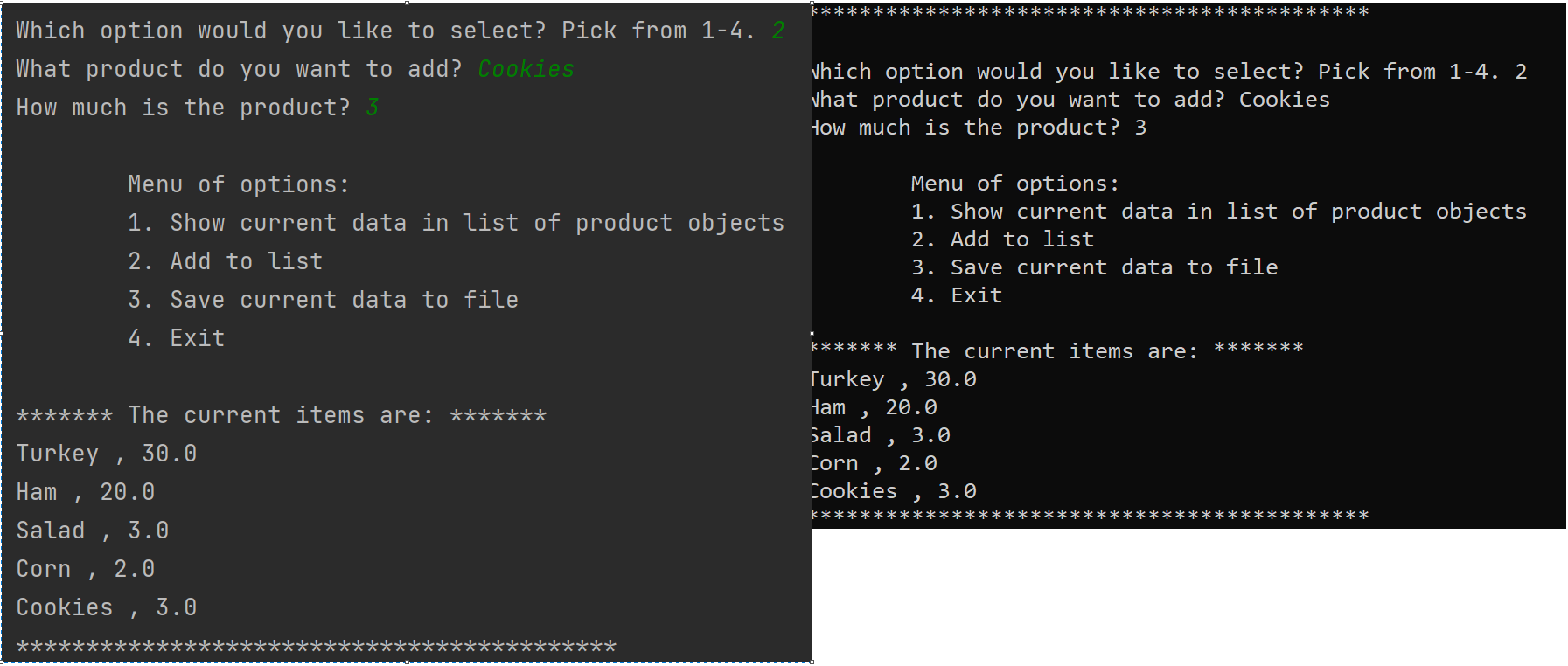
Start:



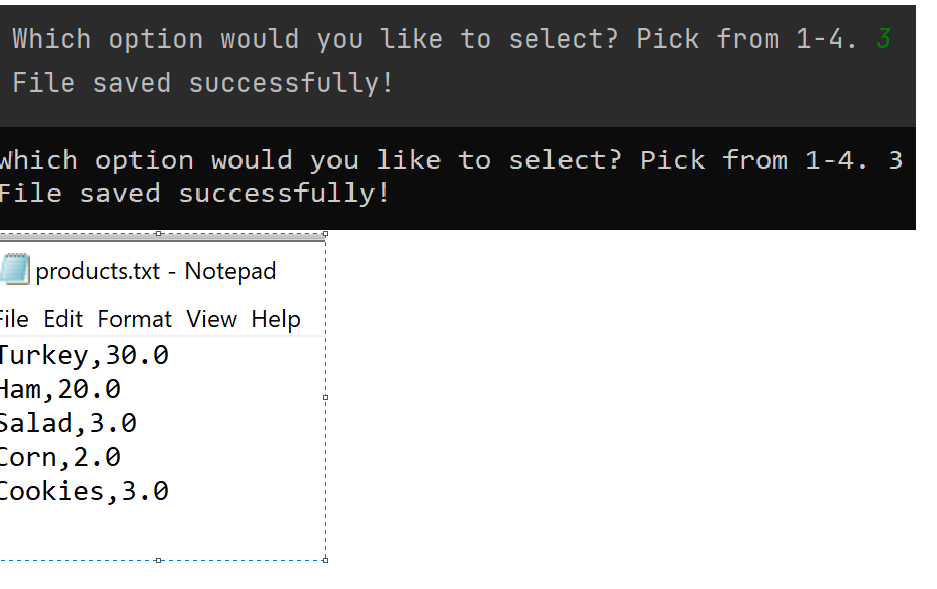
Printing List:



Adding to List:



Saving to File



**Conclusion**

This week, I learned how to set up classes, methods, attributes, properties, and more. While this lesson is certainly more challenging than previous weeks’ lessons, I can see the benefits to using the options, allowing myself to become a more effective programmer writing cleaner code.