Jessi Velazquez

12-8-2020

Foundations of Programming: Python

Assignment 08

**Tracking Costs Using a Python Data Class**

**Introduction**

This document summarizes Python code that uses three classes to allow a user to keep track of products and prices paid in a text file. The script uses three classes, a data class where each product is stored as an object iteration, a processing class of functions which process data, and an IO/presentation class which houses functions related to displaying information or taking information from the user. The new element of learning in this code is the data class, Product().

**Functionality of the Script**

As was the case with other recent Python script summaries, this script utilizes multiple classes to effectively achieve Separation of Concerns (SOC). In this case, we have a single data class called Product(), and two classes containing functions. These classes are called within the main body of code, so to effectively explain this script, we will describe the main code and go into further detail on specific functions where necessary.

We begin our script with a simple try/except block that completes the task of reading the data from an existing copy of “products.txt” file into our program. If that file does not exist, we catch the FileNotFoundError and display a message to user to notify them that no file was found, and so no data was read into the program, and that any data added during this instance of the program will be saved into a new file.

The function being called to read the data in the file is called FileProcessor.read\_data\_from\_file. This works in a similar manner to other recent programs we have summarized, except that this time rather than using dictionary items, we are using our data class, Product(), to store our data as a object iterations within a class of objects. Each object within the Product() class is given two parameters. These are product\_name and product\_price. In the code block below, you can see that like before, we use a for loop to split each line of data using a comma, assigning two local variables “name” and “price”. But now rather than creating a dictionary row with the values for these two variables and appending that to the data list, we pass the values assigned to local variables “name” and “price” into the Product() class, as it’s two parameters “product\_name” and “product\_price”, assigning an object iteration called “Obj”, and then appending that object iteration to the data list. The code is as follows:

def read\_data\_from\_file(file\_name, list\_of\_rows):  
 *""" Reads data from a file into a list of dictionary rows* ***:param*** *file\_name: (string) with name of file:* ***:param*** *list\_of\_rows: (list) you want filled with file data:* ***:return****: (list) of dictionary rows  
 """* list\_of\_rows.clear() # clear current data  
 file = open(file\_name, "r")  
 for line in file:  
 name, price = line.split(",")  
 Obj = Product(name, price)  
 list\_of\_rows.append(Obj)  
 file.close()  
 return list\_of\_rows, 'Success'

After reading the data into the program, we close the text file and enter our main while(True) loop. The code in this loop is encapsulated by a try/except statement to catch all errors within the code. We will revisit this at the end to explain the different errors we expect to catch.

First we call a function, IO.print\_menu\_tasks(), to display a menu to our user. Then, we take their menu selection choice as global variable “strChoice”, by calling function IO.input\_new\_choice(). If their choice is “1”, we display the data list to the user by calling the function IO.print\_current\_list\_of\_products. The code for this function is as follows:

def print\_current\_list\_of\_products(Product):  
 *""" Shows the current Tasks in the list of dictionaries rows* ***:param*** *list\_of\_rows: (list) of rows you want to display* ***:return****: nothing  
 """* print("\*\*\*\*\*\*\* The current products in list are: \*\*\*\*\*\*\*")  
 for objProd in Product:  
 print(objProd)  
 print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")  
 print() # Add an extra line for looks

Here we use a for loop to access each object iteration within our Product() class, printing each until there are none left. This works in a similar manner to previous scripts where we were accessing each item in a list of dictionary of items, and printing each in a for loop. However, this time we simply call the objects from the data class, Product().

If the user’s menu choice (strChoice) is “2”, we have a code block to add new data to the data list, which looks like this:

elif strChoice == "2":  
 name, price = IO.input\_new\_product\_and\_price()  
 objProd = Product(name, price)  
 IO.print\_entry\_to\_user()  
 FileProcessor.add\_data\_to\_list(name, str(price), lstOfProductObjects)  
 IO.input\_press\_to\_continue()  
 continue

The first line of code after the “elif” statement calls the function IO.input\_new\_product\_and\_price() to collect user input, and assigns the two return values to global variables “name” and “price”. The next line of code passes the global variables “name” and “product” into our Product() class as it’s parameters “product\_name” and “product\_price”. It is important to not here that as defined within the Product() class, “product\_name” is a string value, and “product\_price” is a float value. This is because price must be a numeric value, and in most cases has a decimal value. This is defined within the “constructors” section of our Product() class:

# Constructors  
def \_\_init\_\_(self, product\_name, product\_price):  
 # Attributes  
 try:  
 self.\_\_product\_name = str(product\_name)  
 self.\_\_product\_price = float(product\_price)  
 except Exception as e:  
 raise Exception("Error setting initial values")

After the user enters a new product and price, we print a statement back to them that shows their entry, calling a function which prints the string value of the object created. In out IO.print\_entry\_to\_user() function we use the line of code print(str(objProd) + “ has been added to list.”) to do so, but we format the manner in which the product prints in the “methods” section of our Product() class, which is the following:

# Methods  
def \_\_str\_\_(self):  
 return self.\_\_product\_name + "," + str(self.\_\_product\_price)

We then call function FileProcessor.add\_data\_to\_list(), passing our “name” and “price” variables collected from the user, along with “lstOfProductObjects” into it’s parameters. The function’s code then passes those values into the Product() class, assigning an object iteration within the class, before appending that object iteration to the data list.

If the user selects option 3, we want to save the current data to the “products.txt” file and close the program. So, we call the function “FileProcessor.write\_data\_to\_file()”, and pass the global variables “strFileName” and “lstOfProductObjects” into it’s parameters. The code fot this function is the following:

@staticmethod  
def write\_data\_to\_file(file\_name, Product):  
 *"""* ***:param*** *file\_name: Text file to be opened and written to by program.* ***:param*** *list\_of\_rows: (list) Task list data table* ***:return****: (list) of dictionary tows  
 """* file\_name = open(file\_name, "w")  
 for objProd in Product:  
 file\_name.write(str(objProd) + "\n")  
 file\_name.close()

As you can see, we open the file in write mode, and then use a for loop to write each object iteration within our Products() class to the file as a new line item. We then close the file, and print a message to the user signaling that the file has been saved, before breaking the while(True) loop and exiting the program.

At the very end of our script are the “except” clauses, or error exceptions that correspond to the “try” line at the start of the main body of code. In this program, the errors we foresee occurring are FileNotFoundError, and ValueError. We already handled the FileNotFoundError above our while(True) loop, but we place the exception for ValueError at the end of the script. This error will occur if/when a user enters a non-numerical value as a product’s price. The error explains that to the user, and puts back to the main menu for another chance at entering data without a program crash.

**Summary**

In this document, we have described the actions performed by our Python script to manage data regarding products and pricing in a program that utilizes three classes, most notable a data class called “Product()”.