Jessi Velazquez

11-30-2020

Foundations of Programming: Python

Assignment 07

**Tracking Savings in an Obscure Data File**

**Introduction**

This document summarizes the code a Python script file which allows a user to input both the sale price paid and the regular retail price of a purchased item, calculates the percentage saved on the purchase, and collects the data in an obscure data file (binary file).

**Functionality of the Script**

To explain the beginning of this script, we must begin with its end goal. This program will save data as a byte stream to a binary file, for the purpose of efficiently storing it in a database while maintaining the object hierarchy of the Python language. To do this, we will need to use a Python module called “Pickle”. Thus, the first line of code reads “import pickle”, and it does just what it says.

Since our code is separated out by sections for data, processing, presentation, main code, and error handling, we can skip to the main code section, which is where actionable code is located. Our main code here is a series of function calls to collect user data, do a basic calculation with that data, and append the data and the calculation as a list to our master binary file “SavingsTracker.bin”. The code is the following:

while(True):  
 try:  
 SalePrice, FullPrice = input\_new\_sale() # Get input data  
 Savings = (calculate\_savings(SalePrice, FullPrice)) # calculate savings  
 display\_savings(Savings)  
 add\_data\_to\_list(SalePrice, FullPrice, Savings, lstSale) # create list object  
 append\_list\_to\_bin\_file(objFile, lstSale) # append list object to binary file  
 strChoice = input\_cont\_or\_exit()  
 if strChoice == "y":  
 continue  
 else:  
 break

In this code block, we first create a list with the numeric values added by the user for sale price paid and full retail price of an item purchased by calling the function “input\_new\_sale()”. This function asks for these two pieces of information, then returns them, and we capture the return values as the global variables SalePrice and FullPrice.

Then, we create the global variable Savings, and assign that a value by passing SalePrice and FullPrice into the function “calculate\_savings(sale, full)”, which performs basic math to calculate the percent saved on the purchase. This function is as follows:

def calculate\_savings(sale\_price, full\_price):  
 *"""* ***:param*** *sale\_price:* ***:param*** *full\_price:* ***:return****:  
 """* savings = round(float((full\_price - sale\_price)/full\_price), 3) \* 100  
 return savings

Next, we display the user a message which notifies them of the percent they saved on the purchase by calling the “display\_savings(savings)” function. We then add SalePrice, FullPrice, and Savings to a list object called lstSale by calling the function “add\_data\_to\_list(sale\_price, full\_price, savngs, list\_of\_rows)”. The code for this is as follows:

def add\_data\_to\_list(sale, full, savings, list\_of\_rows):  
 *"""* ***:param*** *sale:* ***:param*** *full:* ***:param*** *savings:* ***:param*** *list\_of\_rows:* ***:return****:  
 """* row = {"Sale\_Price": sale, "Full\_Price": full, "Savings": savings}  
 list\_of\_rows.append(row)  
 return list\_of\_rows, 'Success'

At this point we have collected and processed our data to be appended to the master binary file, “SavingsTracker.bin”. Recall the first line of code where we imported the module “pickle”? Now is when we put that module to use. Calling the function “append\_list\_to\_bin\_file(file\_name, list)”, we run the following function code block:

def append\_list\_to\_bin\_file(file\_name, list):  
 *"""* ***:param*** *file\_name:* ***:param*** *list:* ***:return****:  
 """* file\_name = open("SavingsTracker.bin", "ab")  
 pickle.dump(lstSale, file\_name)  
 file\_name.close()

This code opens the “SavingsTracker.bin” file in append mode, which will append new data to the file without overwriting existing data, and if the file does not already exist, it will create it. The next line of code calls the “pickle” module, and uses the built in .dump function to “dump”, or add, the list object we have already created (“lstSale”) to the master file. The third line closes the file.

Then, we call the function “input\_cont\_or\_exit()” to get a user’s choice of whether to add another purchased item to the master binary file (by pressing y), or exit the program (by pressing any other key), and executing a simple if/else statement to either continue back to the start of the while(True) loop, or break the loop and close the program.

However, this is not the end of the Python script! We have yet to mention it, but you may have noticed the word “try” placed first in the main code block, right before the while(True) loop. What this word does is set up a try/except statement surrounding out main code, that serves the purpose of catching any possible user errors that may cause program crashes. By doing this, we are able to display a custom error message to the user and avoid a program crash, giving them another chance to get it right.

This try/except clause encapsulates the main code with the word “try” in front of the main code, and its corresponding “except” code blocks placed after the main code block. The except statements, also known as “exceptions” are as follows:

except ZeroDivisionError as e:  
 print(e)  
 print("Cannot divide by zero, please enter non-zero value for second number")  
 continue  
except ValueError as e:  
 print(e)  
 print("Price must be numeric value, please enter numeric value.")

This program is quite simple, and so we only have two exceptions. The first is a zero division error, meaning that the user input a zero value for the full price paid, causing the program to attempt to divide by zero when it reaches the “calculate\_savings(sale\_price, full\_price) function. This is mathematically impossible, so Python crashes and throws a built in error message that can be a little confusing and alarming. So instead of allowing the user to see this built in error and losing their work, we display the custom error message “Cannot divide by zero, please enter non-zero value for second number.”

The second exception occurs when a user inputs a non-numeric value into one of the two inputs for sale price or full price. Since prices must be numeric (we don’t track service trade agreements in this program), the program also crashes when it attempts to perform arithmetic with a value such as “jgd”. This is called a value error, and instead of allowing the program to crash and alarming the user, we simply display the message “price must be numeric value, please enter numeric value”.

**Summary**

In this document, we have described the actions performed by our Python script to take user input of sale price paid and full retail price for an item purchased, calculate the percentage saved, and collect the data in an obscure data file (binary file).