"Shopping Cart" Further Exploration Challenges

This document provides optional project challenges for students seeking a greater level of difficulty.

Functionality Challenges

Configuring Sales Tax Rate

You'd like to share your code with stores in other locations as well, but different municipalities use different sales tax rates. Instead of hard-coding the sales tax rate, allow the user to configure it via environment variable using a ".env" file approach.

> HINT: use The `os` Module in conjunction with The `dotenv` Package

> HINT: you might need to use the `float()` function to convert the environment variable value to a float datatype, so you can perform numeric calculations with it

Handling Pricing per Pound

Add a new product called "Organic Bananas" to the products inventory. Assign it a price of `0.79`, but add another attribute called something like `price_per` to indicate the item is priced per "pound". Update all the other product dictionaries to match the new structure, indicating they are priced per "item".

When running the program, if the clerk inputs the identifier of the bananas (or any other item that is priced by pound), the program should ask the clerk to input the number of pounds (e.g. `2.2`), then the program should calculate the price accordingly.

Writing Receipts to File

Instead of, or in addition to, displaying a receipt at the end of the checkout process, the program should write the receipt information into a new ".txt" file saved in a new "receipts" directory inside the project repository. The clerk's printer-connected computer should be able to actually print a paper receipt from the information contained in this file.

Each text file should be named according to the date and time the checkout process started (e.g. "/receipts/2019-07-04-15-43-13-579531.txt", where the numbers represent the year, month, day, 24-hour-style hour, minute, second, and milliseconds/microseconds, respectively).

> HINT: consult the notes on file management for examples of how to write to file in Python

> NOTE: exclude these receipt files from being tracked in version control by using a "receipts/.gitignore" file with the following contents:

```
# data/.gitignore
```

```
# see: https://stackoverflow.com/a/5581995/670433

# ignore all files in this "data" directory:
*

# except this ".gitignore" file:
!.gitignore
```

Sending Receipts via Email

Instead of, or in addition to, displaying a receipt at the end of the checkout process, the program should prompt the checkout clerk or the customer to indicate whether the customer would like to receive the receipt by email. And if so, it should prompt the checkout clerk or the customer to input the customer's email address, and then it should send the receipt information to the customer by email. The clerk's network-connected computer should be able to send these emails.

At the very least, the email should display the checkout timestamp and the total price. But ideally it should contain all the receipt information described in the basic requirements.

> HINT: leverage the email-sending capabilities of the `sendgrid` package, and optionally use Sendgrid email templates to further control the formatting of email contents

Integrating with a Google Sheets Datastore

Instead of using a hard-coded `products` variable or a "products.csv" file as the application's datastore, use this provided products Google Sheet document instead.

> HINT: leverage the capabilities of the `gspread` package

Integrating with a Barcode Scanner

Assemble a handful of real products that have barcodes.

Ask to borrow the professor's barcode scanner during class or office hours, or feel free to purchase one from Amazon.

Connect the barcode scanner to your computer's USB port, and practice scanning a few of the items, and record the resulting product identifiers.

Adapt the program's `products` variable (or CSV file or Google Sheet, or whatever datastore you're using) to reflect the real products and their identifiers. For example:

```
# shopping_cart.py
# ...
products = [
    {"id": "99482452704", "name": "Organic Black Beans", "price": 0.99},
```

```
    {"id": "99482434182", "name": "Organic Almonds Roasted Unsalted",
"price": 7.33},
    {"id": "99482418939", "name": "Jug of Spring Water", "price": 0.99},
    {"id": "898248001107", "name": "Siggi's Strawberry Yogurt", "price":
1.45},
    {"id": "898248001114", "name": "Siggi's Peach Yogurt", "price": 1.45},
    {"id": "290295004269", "name": "Whole Foods Guacamole - Small", "price":
6.50},
    {"id": "012000161155", "name": "LIFE Water", "price": 2.15},
]
# ...
```

> NOTE: for real life products, we'll probably want to change the identifier values to strings, in case any contain alphabetic characters or leading zeros (e.g. "LIFE water" identifier above)

After modifying the product list, try re-running the program, using the barcode scanner to scan the real items. It should work!