Programming with Python Coursework 2020

Supermarket Self-Checkout System

Stefan Güttel, guettel.com

Overview

The aim of this coursework is to code a Python 3 module that simulates a supermarket cashier system (also known as POS system). The module allows a self-checkout customer to list the products on stock, add products from the stock to a shopping basket or remove products from it, list all the items in the basket, and produce a bill of the basket (applying two types of promotions). While there are several ways of solving this problem, in this project *you are required to use the data structures described below and follow the tasks as outlined.* First read and understand the whole description in this document before starting with the actual work.

General comments

How should I go about this coursework?

The best way to approach this coursework is, as usual, to first sit down with pen and paper and think about what data structures and functions should be required to represent and work on a supermarket shopping basket. Clearly, we will have a stock of products each of which has an associated price and amount of availability (for example, we may have 2.210 kg of bananas priced at £0.68 per kg available on stock). We want to be able to take a certain amount of product (bananas) from the stock and add it to a shopping basket. Likewise, we want to be able to take products out of the basket and put them back on stock. Finally, when we are done with our shopping we want to check out the basket and get a bill. This is the essence of this coursework.

In order to make life easier for you, below you will find nine concrete tasks to follow (Tasks 0-8). These tasks are ordered logically and in (approximately) increasing level of difficulty. The only exception is Task 7, which explains the main functionality of the module. You may look at Task 7 earlier.

What can/cannot be used for this coursework?

The Python knowledge contained in the course notes is sufficient to complete this project. While you are allowed to consult the web for getting ideas about how to solve a specific problem, the most straightforward solution may already be in the course notes. It is not allowed to copy-and-paste several lines of computer code from a (online) source without clearly indicating their origin. If you really need to use an online source, you must indicate this for example like so:

```
# the following three lines follow a similar code on
   # http://webaddress.org/somecode.htm as retrieved on 24/11/2020
```

Let's be very clear about **what is not allowed:**

You are NOT allowed to send/give/receive Python code to/from classmates and others.
This project is the equivalent of a standard exam and it counts 70% towards your final mark.
Consequently, the standard examination rules apply to this project.

Once your Python module is submitted via the Blackboard system, it will undergo plagiarism tests:

1.) The turn-it-in system checks for similarities in the codes among all students.

2.) Another Python program compares the syntax of all submitted codes.

Note that even if you are the originator (and not the one who copied), the University Guidelines require that you will be equally responsible for this case of academic malpractice and may lose all marks on the coursework (or even be assigned 0 marks for the overall course).

How is this coursework assessed?

There are several factors that go into the assessment.

- First of all, it will be checked whether you have followed the tasks and format specified below (using the prescribed function names, variable names, etc.).
- All functions of your code will be tested automatically by another computer program. The outputs are then compared to verified outputs of another implementation. (This also means that if you do not strictly follow the format specified below then some of these automatic tests will fail and you may lose marks.)
- Each function/line of code should be bug free. Functionality will be the main factor in the assessment.
- Your code should be robust to exceptional user inputs (using Exceptions). It should not be possible to crash the code even when a user provides an unexpected input.
- It is required that your module is properly documented, and that the module and every of its functions has a meaningful docstring. Each function must explain its own input arguments and returned values and their types. There is no room for misinterpretation. Check that `print(functionname.__doc__)` returns a proper docstring. Essentially, each function should be useable from the docstring alone, without reading the code.
- Further, marks will be given on code effciency. Have you solved a problem using 100 lines of code, although we have learned a very straightforward way which would only require 2 lines? You may lose marks in this case.

The rough split (subject to scaling) between the total marks is 65% for testing and manual inspection, 15% for documentation, and 20% for code efficiency.

Submission format and deadline

- The complete coursework can be completed and submitted as a single Python file `supermarket.py`.
- Click here to download a template module: `supermarket.py`.
- The submission will be via Blackboard and **the deadline is Friday, December 18, at 1pm.**

Important: You have almost four weeks for completing this coursework, but please do not wait until the end of the semester for getting started. *The deadline of Friday, December 18, at 1pm is strict and no exceptions can be made for fairness.* Aim to submit your coursework earlier than this so there is enough time to sort out eventualities. You can resubmit as often as you like, but only the last submission counts.

That's it, below are the nine tasks of the coursework...

Task 0. Understanding the data structures: `stock` and `basket`

Before you start with the actual coding, understand how to keep track of the products on stock and the products in the shopping basket. The fundamental data structures of your module will be two dictionaries `stock` and `products`. The `stock` dictionary has the following form (only showing three representative items):

```
stock = {
    '10005' : {
            'name' : 'Conference Pears Loose',
            'price' : 2.00,
            'unit' : 'kg',
            'promotion' : None,
            'group' : None,
            'amount' : 1.550
     },
     '10013' : {
            'name' : 'Emmental Slices 250G',
            'price' : 1.75,
            'unit' : 'pieces',
            'promotion' : 'get2pay1',
            'group' : None,
            'amount' : 9
     },
     '10015' : {
            'name' : 'Diced Beef 400G',
            'price' : 4.50,
            'unit' : 'pieces',
            'promotion': 'get4pay3',
            'group' : 4,
            'amount' : 14
    },
    # ...more products on stock...
}
```

Each key in the dictionary is a string referred to as `ident` (in the above example, the `ident` strings are `10005`, `10013`, and `10015`), which serves as a unique identifier for each product. Each product is again represented as a dictionary, with the keys `name`, `price`, `unit`, `promotion`, `group`, and `amount`. Here is a more detailed description of the corresponding values:

- `name` : a string representing the name of a product (e.g., `Granny Smith Apples Loose`)
- `price` : a floating point number representing the price (in Great Britain Pounds)
- `unit` : a string which is either `pieces` or `kg`, indicating whether the product is costed in pieces or per kilogram weight
- `promotion` : the value can either be `None` or a string `get2pay1` or `get4pay3`
- `group` : refers to the group of promotion and it can either be `None` or an integer (the latter only when `promotion` is `get4pay3`)
- `amount` : refers to the amount of a product being available on stock; it can either be an integer (in case `unit` takes the value `pieces`) or a floating point number (in case `unit` takes the value `kg`).

Example: The `stock` dictionary above indicates that we have 1.55 kg of pears on stock, with each kg priced at £2. There are no promotions on pears.

Now assume the customer adds 0.55 kg of pears to the shopping basket. Then we can simply represent the content of the basket using another dictionary of the form

```
basket = {
    '10005' : {
            'name' : 'Conference Pears Loose',
            'price' : 2.00,
            'unit' : 'kg',
            'promotion' : None,
            'group' : None,
            'amount' : 0.550
     },
}
```

This means we have essentially copied the value of `stock['10005']` to the basket dictionary and set the amount to the demanded value. At the same time, the amount of pears available on stock has been reduced to 1 kg. We can also add negative amounts of pears to the basket, in which case we are taking them out and adding them back to the stock.

Task 1. Load stock data: `loadStockFromFile(filename)`

Write a function `loadStockFromFile` which reads the products from a file and returns the dictionary `stock` in the form described above. The function accepts an *optional* string parameter `filename` corresponding to a file in the same directory as your Python module (i.e., there is no need to specify a path). If `filename` is not provided, the value `stock.csv` should be assumed. If a file cannot be found or opened, for whatever reason, the function should throw an appropriate exception.

Example file: To download the example **`stock.csv`** used here, right-click on the link and choose "Save target as..." or "Save as..." (or similarly, depending on your operating system). Save the file to the same directory as your Python module `supermarket.py`.

Each line in `stock.csv` is of the form

```
name|price|unit|promotion|group|amount
```

Ensure that the values in the sub-dictionaries representing each product have the correct type. For example, if `unit` takes the value `pieces`, then `amount` is an integer, otherwise it is a floating point number.

The `loadStockFromFile()` function should ignore "corrupted" lines in `stock.csv` which are not of the correct format. For example, the following line would not be of the correct format:

```
10014|Beef Steak Mince 600G|4.00|pieces|get4pay3|4|3.21
```

as the unit is `pieces` and the amount (the last value) is not an integer (3.21). This item should be skipped.

Also ensure that `None` values in `stock.csv` become proper `None` 's in the dictionary (of type `NoneValue`, not `str`).

Note: If for some reason you fail to implement the function `loadStockFromFile()`, or if you prefer to start working on other parts of the module without loading data from a file first, you may copy-and-paste the above `stock` dictionary with only three items to your module.

Task 2. Table of items on stock or in the basket: `listItems(dct)`

As both the `stock` and the `basket` dictionaries are of the same form, we only need to write one function which displays their items. Write a function `listItems(dct)` which takes as input a dictionary `dct` (which will be either `stock` or `basket`) and **returns** (not prints) a string corresponding to a nicely formatted table.

The products need to be sorted by increasing `ident` identifier (lexicographically). The prices need to be right-aligned and displayed with two decimal places. The kg amounts are displayed with one decimal place.

Example: If `s = listItems(dct); print(s)` is called, the returned string should be formatted similar to the following (your precise formatting and the items might differ):

```
 Ident | Product                              |   Price   |   Amount
-------+--------------------------------------+-----------+-------------
 10000 | Granny Smith Apples Loose            |   0.32 £ |    6 pieces
 10001 | Watermelon Fingers 90G               |   0.50 £ |   17 pieces
 10002 | Mango And Pineapple Fingers 80G      |   0.50 £ |    2 pieces
 10003 | Melon Finger Tray 80G                |   0.50 £ |   10 pieces
 10004 | Bananas Loose                        |   0.68 £ |    2.2 kg
 10005 | Conference Pears Loose               |   2.00 £ |    1.6 kg
```

Task 3. Search the stock: `searchStock(stock, s)`

Write a function `searchStock(stock, str)` which returns a dictionary `substock` of the same form as `stock`, but containing only those products whose `name` contains the string `s` as a substring. The search should be case-insensitive.

Example: A call to `searchStock(stock, 'beef')` should return a dictionary

```
substock = {
    '10015' : {
            'name' : 'Diced Beef 400G',
            'price' : 4.50,
            'unit' : 'pieces',
            'promotion' : 'get4pay3',
            'group' : 4,
            'amount' : 14
     },
     '10017' : {
            'name' : 'Brisket Beef 400G',
            'price' : 3.50,
            'unit' : 'pieces',
            'promotion' : 'get4pay3',
            'group' : 4,
            'amount' : 15
     }
}
```

Conveniently, because `substock` and `stock` are dictionaries of the same format, in your program you can use the same function `listItems(substock)` from Task 2 to display the search results.

Task 4. Add or remove items: `addToBasket(stock, basket, ident, amount)`

Write a function `addToBasket(stock, basket, ident, amount)` which adds `amount` units of the product with identifier `ident` from the stock to the basket. The function should return a message `msg`, which can either be `None` or a string. There are four possible scenarios this function must handle:

1) If `amount` is positive and there are sufficiently many units of the product on stock, then `amount` units will be taken out and added to the basket. See the above example in Task 0, where we have added 0.550 kg of pears from the stock to the basket. In this case the function will return `msg = None`, i.e., no message.

2) If `amount` is positive but there are less units of the product on stock, then only this many units will be added to the basket (hence leaving 0 units of the product on stock). The function will return a message of the form `msg = 'Cannot add this many {unit} to the basket, only added {amount} {unit}.'`, where the appropriate replacements are performed on the string.

3) If `amount` is negative and there are sufficiently many units of the product in the basket, then `amount` units will be taken out and added back onto the stock. If it happens that the amount of product remaining in the basket is zero, then the corresponding product should be deleted from the `basket` dictionary. In this case the function will return `msg = None`, i.e., no message.

4) If `amount` is negative but there are less units of the product in the basket, then the product will be removed from the basket and added back to the stock. The function will return a message of the form `msg = 'Cannot remove this many {unit} from the basket, only removed {amount} {unit}.'`, where the appropriate replacements are performed on the string.

Hint: Recall that dictionaries behave similarly to lists and you can simply modify `stock` and `basket` inside a function. The changes will apply to the dictionaries available outside module. Therefore your `addToBasket` function does not need to return updated `stock` and `basket` variables, only the message `msg`. The function should update the values of `stock` and `basket`, which are provided as arguments to it, directly.

Task 5: Prepare for checkout: `prepareCheckout(basket)`

This is a very quick task (it can be done in two lines of code): Write a function `prepareCheckout(basket)` that loops through each item in the `basket` and adds a key `amountPayable` taking the same value as the corresponding `amount` value. This will be required later (in Task 8) because with promotions you may pay for an amount of a product that is less than the actual amount in the basket.

Example: Assume that our `basket` dictionary is of the following form:

```
basket = {
    '10013' : {
            'name' : 'Emmental Slices 250G',
            'price' : 1.75,
            'unit' : 'pieces',
            'promotion' : 'get2pay1',
            'group' : None,
            'amount' : 7
     },

    '10015' : {
            'name' : 'Diced Beef 400G',
            'price' : 4.50,
            'unit' : 'pieces',
            'promotion' : 'get4pay3',
            'group' : 4,
            'amount' : 5
     },
     '10017' : {
            'name' : 'Brisket Beef 400G',
            'price' : 3.50,
            'unit' : 'pieces',
            'promotion' : 'get4pay3',
            'group' : 4,
            'amount' : 3
     },
}
```

Then after calling `prepareCheckout(basket)` we want `basket` to be the following:

```
basket = {
    '10013' : {
            'name' : 'Emmental Slices 250G',
            'price' : 1.75,
            'unit' : 'pieces',
            'promotion' : 'get2pay1',
            'group' : None,
            'amount' : 7,
            'amountPayable' : 7
     },

    '10015' : {
            'name' : 'Diced Beef 400G',
            'price' : 4.50,
            'unit' : 'pieces',
            'promotion' : 'get4pay3',
            'group' : 4,
            'amount' : 5,
            'amountPayable' : 5
     },
     '10017' : {
            'name' : 'Brisket Beef 400G',
            'price' : 3.50,
            'unit' : 'pieces',
            'promotion' : 'get4pay3',
            'group' : 4,
            'amount' : 3,
            'amountPayable' : 3
     },
}
```

That's it. At this stage we will not worry about applying promotions, and only come back to them later. The next two steps are easier and should be done first.

Task 6: Produce a bill `getBill(basket)` (without promotions)

Write a function that returns, as a string, a nicely formatted bill for the `basket`. The total amount to be paid is given by the sum of `amountPayable*price` for each product in the basket. The returned string should also include the total amount to be paid.

Example: For the basket in Task 5 the returned string from `getBill(basket)` should look similar to

```
Here is your shopping bill:

Product                              |   Price   |   Amount   |  Payable
-------------------------------------+-----------+------------+----------
Emmental Slices 250G                 |    1.75 £ |   7 pieces |   12.25 £
Diced Beef 400G                      |    4.50 £ |   5 pieces |   22.50 £
Brisket Beef 400G                    |    3.50 £ |   3 pieces |   10.50 £
-------------------------------------+-----------+------------+-----------
TOTAL:                                                            45.25 £
```

Task 7: The `main()` function

The `main()` function uses all the above functions to mimic a self-checkout system which allows the customer to list the products on stock, add (remove) products from the stock to their shopping basket, list the items in the basket, and display a bill for the basket.

You may want to use the following general form of the module (also available for download: `supermarket.py`):

```
"""
Add a short description of the module.
"""

import csv # you may use this for loadStockFromFile()

# feel free to import other modules or define your own functions

# Task 1
def loadStockFromFile(filename):
    """
    TODO: include a docstring for each function
    """
    stock = {}
    # TODO: load the stock items from file
    return stock

# TODO: functions for the other tasks can go here

# Task 7
def main():
    """
    TODO: include a docstring
    """
    stock = loadStockFromFile('stock.csv')
    basket = { }
    print("*"*75)
    print("*"*15+" "*10+"WELCOME TO STEFAN EXPRESS"+" "*10+"*"*15)
    print("*"*75,"\n")

    while True:
        s = input("Input product-Ident, search string, 0 to display basket, 1 to check out: ")
        # TODO: complete the code

if __name__ == '__main__':
    main()
```

As you can see, in the main function we will ask the user for a string `s`. Depending on this input, the following things can happen:

- If `s` is `"0"`, then your function should output a string like "`Your current shopping basket:` ", followed by a use of the `listItems(basket)` function to display the contents of the shopping basket. If the basket is empty, an informative message should be displayed. After the basket content has been listed, the function asks for user input again.
- If `s` is `"1"`, then `prepareCheckout(basket)` and `getBill(basket)` are used to display the bill for the shopping basket. Following this, a goodbye message will be displayed (e.g., "`Thank you for shopping with us!` ") and the function stops.
- If `s` corresponds to an `ident` key in the dictionary `stock`, then the user wants to add/remove the corresponding product to/from the basket. The function hence asks "`How many units ({}) do you want to add to your basket?` " (where `{}` can be either `pieces` or `kg`), and inputs the amount (say `nr`). If the amount `nr` is not an integer or a floating point number (depending on what `unit` the product has), the function we will ask for a corrected input. The function then calls `msg = addToBasket(stock, basket, s, nr)` which will invoke the function we have defined in Task 4. If the returned `msg` is not `Null`, the program displays `msg` to the user (recall, `msg` is a string representing a warning that `nr` was to large positive or negative). Afterwards the program asks for user input again.
- If `s` is anything else, then we assume the user wants to search the stock and call `substock = searchStock(stock, s)` and use `listItems(substock)` to display the search results. Before listing the results, an appropriate description is given, and following it we again ask for user input. For example:

```
Input product-Ident, search string, 0 to display basket, 1 to check out: beef

There were 2 search results for 'beef':

 Ident | Product                              |   Price   |   Amount
-------+--------------------------------------+-----------+-------------
 10015 | Diced Beef 400G                      |    4.50 £ |   14 pieces
 10017 | Brisket Beef 400G                    |    3.50 £ |   15 pieces

Input product-Ident, search string, 0 to display basket, 1 to check out:
```

Task 8: Applying promotions `applyPromotions(basket)`

Once you have completed all previous tasks and your program is working flawlessly you can attempt this final task!

Write a function `applyPromotions(basket)` that adjusts the value of `amountPayable` for each item in the basket. There are two types of promotions, indicated by the value of the `promotion` key associated with each product.

- **Get 2 Pay 1:** If a product has a `promotion` string equal to `get2pay1`, then every second instance of this product in the basket is free. For example, if `amount` is equal to 3, then with this promotion type we will set `amountPayable` to 2.
- **Get 4 Pay 3:** If a number of products has a `promotion` string equal to `get4pay3` and the same integer value for `group` (i.e., they are in the same promotions group), then out of 4 instances only 3 items have to be payed for. Our shop tries to maximize its profit by only giving away the cheapest products for free.

To illustrate, here is the result of `applyPromotions` being applied to the `basket` in Task 5:

```
basket = {
    '10013' : {
            'name' : 'Emmental Slices 250G',
            'price' : 1.75,
            'unit' : 'pieces',
            'promotion' : 'get2pay1',
            'group' : None,
            'amount' : 7,
            'amountPayable' : 4
     },

    '10015' : {
            'name' : 'Diced Beef 400G',
            'price' : 4.50,
            'unit' : 'pieces',
            'promotion' : 'get4pay3',
            'group' : 4,
            'amount' : 5,
            'amountPayable' : 5
     },
     '10017' : {
            'name' : 'Brisket Beef 400G',
            'price' : 3.50,
            'unit' : 'pieces',
            'promotion' : 'get4pay3',
            'group' : 4,
            'amount' : 3,
            'amountPayable' : 1
     },
}
```

Note that we only pay for 4 Emmentals, because 3 out of 7 Emmentals in our basket qualify for the "Get 2 Pay 1" promotion. Likewise, we have a total number of 8 items qualifying for the "Get 4 Pay 3" promotion and sharing the same promotions group `4`, hence 2 of these items should go for free. As our shop is greedy, these free items will be "`Brisket Beef` " (which is cheaper than "`Diced Beef` "), hence `amountPayable` has been reduced from `3` to `1`.

Finally, adjust the `getBill` function from Task 6 to return a string displaying the applied promotions:

```
Here is your shopping bill:

Product                              |   Price   |   Amount   |  Payable
-------------------------------------+-----------+------------+----------
Emmental Slices 250G                 |    1.75 £ |   7 pieces |   12.25 £
    Promotion get2pay1               |   -1.75 £ |   3 pieces |   -5.25 £
Diced Beef 400G                      |    4.50 £ |   5 pieces |   22.50 £
Brisket Beef 400G                    |    3.50 £ |   3 pieces |   10.50 £
    Promotion get4pay3               |   -3.50 £ |   2 pieces |   -7.00 £
-------------------------------------+-----------+------------+----------
TOTAL:                                                            33.00 £

**********     THANK YOU FOR SHOPPING AT STEFAN EXPRESS!     ***********
```

Hint: You can easily infer which products have been subject to promotions by comparing `amount` and `amountPayable`.