Project #1 - Hot Byte Cookies

Hot Byte Cookies (HBC) is a bakery located close to GA Tech, known for late-night deliveries of hot cookies. A big part of its regular clientele are GA Tech students pulling all-night study sessions.

You will program an ordering system for HBC. Please only address the following requirements.

Process

The following process is reflected in the sample dialog between example customers and the system (see the provided starter Notebook). You can also review the example dialog for additional clarity on any of the below requirements.

When a customer comes into the store, they are presented with this menu display:

Menu Items	Sugar Cookie	Chocolate Cookie	Peanut Cookie	Menu Item Price
Burdell Bundle	5	5	5	30.0
Delicious Duo	3	3	0	9.0
Six Pack	2	2	2	12.0
Sugar Snack	6	0	0	6.0

Table 1: **Menu Items Table**

Once a customer has inspected the menu, they will be ready to place an order. For simplicity, we will assume a customer can place only a single order from the menu at a time (one order per interactive session).

As cookies are only baked once a day, there may be insufficient stock to fulfill an order. If this is the case, the order should not be taken, and the system will ask the customer to revise their initial order based on the available quantity. For example, if a customer wants to order 10 Six Packs and there are only enough cookies on hand to satisfy 5 Six Packs orders, the system will ask the customer to revise their order but limit the order quantity to 5 Six Packs.

After an order has been accepted, the system will inform the customer of the total price (which includes sales tax of 10%) and the order details will be recorded by adding a new record to the `orderRec` dictionary.

Data Structures

The `menu` dictionary is a collection of menu items. Each **menu item** reflects a specific **cookie mix** (cookie cardinality) of the various **cookie types** that we bake and sell.

Assume no new menu items will be added or deleted – but it is possible the cookie composition of a menu item may be adjusted over time (but there is no need to include specific code to enact this).

The prices of cookies can change (but there is no need to include specific code to enact this).

Cookies are not sold individually. As cookies are sold the associated stock amounts will be updated.

When an order is completed it will be recorded in the `orderRec` dictionary. The key to this dictionary is the `tranCode` field. You will notice the encoding of this field reflects the menu item ordered and also the transaction number (starting from 1). When adding a new record to the `orderRec` dictionary the system will need to establish the last recorded transaction number applicable to that menu item type, as the `tranCode` recorded for the current order must feature the next number.

Program Design

The order-taking dialog between the system and customer – including displaying the initial menu – will be controlled by code outside any function (something akin to a main function – but there is no need to include an actual main function in your code because we have not emphasized this concept, and even without a main function, the Python interpreter will automatically execute all your source code, starting at the top line).

Your program must have the following functions (and arguments).

Function	Description

`displayMenu(menu)`	• Displays the table of menu items.
	• Return is **None**.
	• Notes:
	  ○ Uses `PrettyTable` module (see **Addendum 3**).
	  ○ Do not hard-code the cookie mix, menu items, or menu prices (use `getMenuPrice()` to derive the latter).
	  ○ Hard-code the table's field names.
	  ○ Use PrettyTable's `field_names` attribute, & `add_row()` function.
	  ○ Menu items in the table must be sorted alphabetically (i.e., *Burdell Bundle* presented in the top row).

`getMenuPrice(menuItem)`	• Returns the total price of a specific menu item. • You can hard-code keys referring to the cookie types (e.g., Sugar).

`maxUnitsCanOrder(menu, chosenMenuItem, cookieInv)`	• Returns the maximum number of units that can be ordered for a chosen menu item, given the current level of inventory of each cookie type.
	• Notes:
	  ○ Create two lists:
	  ▪ cookie mix integer values (need).
	  ▪ inventory integer values (supply).
	  ○ One option is to then use a list comprehension to loop through both lists and compare the respective values (see **Addendum 1** for logic) to derive a list of capacity requirements for each cookie type within the cookie mix. You will need to accommodate the DIV/0 issue. Also see **Addendum 2** for additional noted on using List Comprehensions.
	  ○ Finally, pass the generated capacity list into min() to determine the maximum order amount for that specific menu item, given

	the current level of inventory for each cookie type within the cookie mix.
`decreaseInv(menu, chosenMenuItem, cookieInv, orderQty)`	• Updates inventory to reflect sales of cookies. • Do not hard-code cookie types. • Return is **None**.
`calcTranCost(chosenMenuItem, orderQty)`	• Returns the total cost of the purchase transaction.
`recordOrder(orderRec, chosenMenuItem, orderQty, tranCost)`	• Records completed orders. • New records must be added to the end of the data structure and feature the appropriate transaction header code (e.g., SS, BB, etc.) and number, given the chosen menu item. • Hard-code the transaction header codes (e.g., SS, BB, etc.). • Do not hard-code the time stamp. • Do not hard-code transaction numbers. • Return is **None**.

Table 2: **Function Definitions**

Both functions: `decreaseInv()` and `recordOrder()` update data structures. In this respect, there are three design options:

- Call a function that will modify the data structure. Do not pass the data structure (in this case, a dictionary) as an argument. No return value. Note: Hard-coding a reference to a specific global object within a function – while allowable – restricts function flexibility.
- Call a function that will modify the data structure. Pass the data structure to update (in this case, a dictionary) as an argument. No return value. [**Use this option**].
- Call a function that will modify the data structure. Pass the data structure to update (in this case, a dictionary) as an argument. Return the updated data structure.

Other Requirements

- Include all your code within a single Jupyter cell.
- Do not use the keyword **global**.
- Do not include any print statements within functions (except displayMenu(menu)).
- Functions should be located after the data structure. The order of functions should be the same as the order of functions listed in Table 2.
- Include substantial program comments to assist a reviewer in quickly understanding your code.
- Do not capture possible exceptions using try/except statements.
- If you prefer to use a different function naming convention (to the featured camelCase format) feel free to modify the format. But stick with the prescribed terms.

Testing

There are four testing scenarios:

1. There are enough cookies in inventory to satisfy the order. The order must be 5 *Six Packs*.
2. There are insufficient cookies in inventory to satisfy the order and the customer accepts the suggested revised order quantity. The order must be 8 *Sugar Snack*.
3. There are insufficient cookies in inventory to satisfy the order and the customer rejects the suggested revised order quantity. The order must be 3 *Burdell Bundle*.

Generate output reflecting the testing scenarios by copying your code to another cell and executing your program reflecting each test scenario. Then delete the copies of your code to reduce the clutter (to help to grade). **So, please include the generated output of each of the 3 testing scenarios within your submission** – but excluding the statement: "*watermark- do not include this output statement in your testing code output*".

Submission

Submit a Notebook with your code. At the top of the notebook add your name/s. You can elect to work with one other student.

Final Comments

This assignment stresses procedural programming. A more 'scalable' design would likely also reflect: an object-oriented approach (do not declare any classes), use of databases to store persistent data structures, and the use of web technologies to provide an advanced interface to the application.

Addendum 1: 'adequate inventory' logic

An order cannot be taken if there is insufficient cookie inventory for any cookie type that is needed for a particular menu item. For example: if you have no chocolate cookies left in stock, you will still be able to sell Sugar Snacks (as they do not have a chocolate cookie) but will not be able to sell the other menu items.

An approach to implementation of this logic (reflecting example stock & cookie mix scenarios mocked up in Excel) is shown below. Note: things get a more complex when there is no stock on hand for a particular cookie, or when a customer orders a menu item that does not contain all cookie types (to illustrate this, the cookie mix for Six Pack was assumed to change to include no chocolate cookies – see scenario #3 below).

		Sugar	**Choc**	**Peanut**
Assume=>	SixPack:	2	2	2
Assume=>	Inventory:	15	19	17
	Capacity:	7.5	9.5	8.50
Min() =>	7.5	7.5	9.5	8.50
Max can offer=>	**7**			

		Sugar	**Choc**	**Peanut**
Assume=>	SixPack:	2	2	2
Assume=>	Inventory:	10	19	17
	Capacity:	5	9.5	8.50
Min() =>	5.0	5	9.5	8.50
Max can offer=>	**5**			

		Sugar	**Choc**	**Peanut**
Assume=>	SixPack:	2	0	2
Assume=>	Inventory:	10	19	17
	Capacity:	5	#DIV/0!	8.50
Min() =>	5.0	5		8.50
Max can offer=>	**5**			

		Sugar	**Choc**	**Peanut**
Assume=>	SixPack:	2	2	2
Assume=>	Inventory:	10	0	17
	Capacity:	5	0	8.50
Min() =>	0.0	5	0	8.50
Max can offer=>	**0**			

Addendum 3: List Comprehension

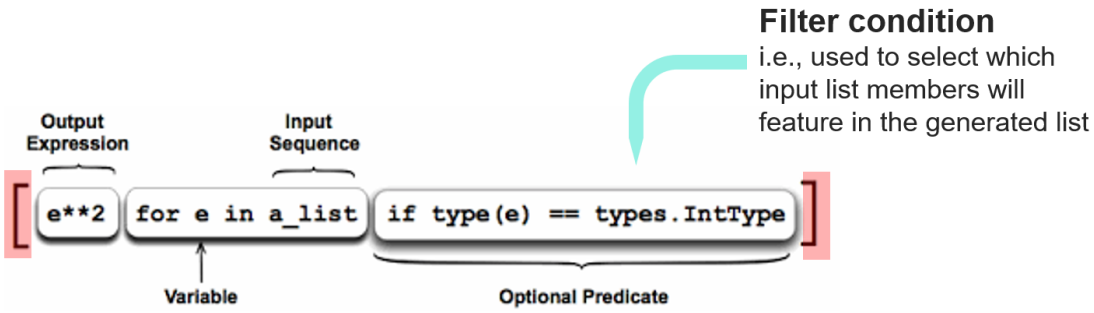
Conversion to a List:

list() converts a set of items to a list object.

List Comprehension:

Contains an expression, which is executed for each element (as defined by the for loop)

- Square brackets signifies that the output is a list



You can place the conditional at the end of the statement for simple filtering, but what if you want to change a member value instead of filtering it out? In this case, it's useful to place the conditional near the beginning of the expression

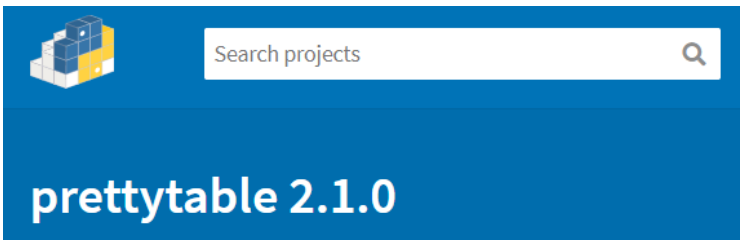
```
# change 'G' to an 'XX'
nameStr = 'George Burdell'   # str is an interator
GeorgesChrs = [ chr if chr != 'G' else 'XX'  for chr in nameStr ]
print(GeorgesChrs)
```

```
['XX', 'e', 'o', 'r', 'g', 'e', ' ', 'B', 'u', 'r', 'd', 'e', 'l', 'l']
```

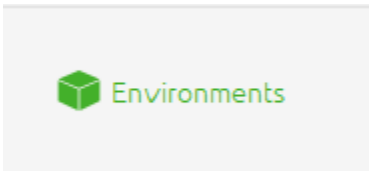
for modifying members

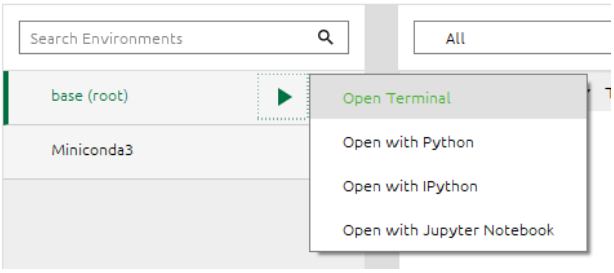
Addendum 3: prettytable

The prettytable module is not pre-installed in Anaconda.



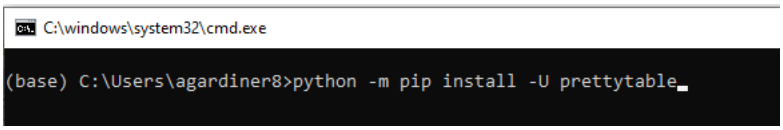
To install this module within Anaconda, open a terminal within Environments.



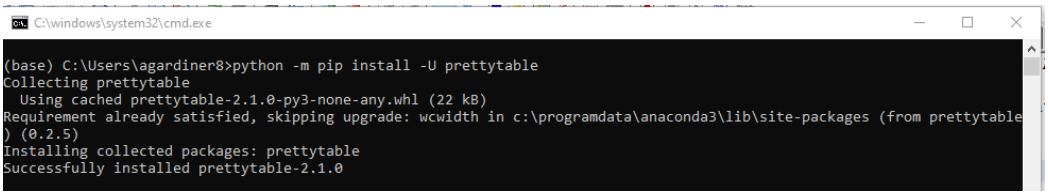


At the Terminal prompt, execute the following command:

> python -m pip install -U prettytable



Success looks like:



For help:

from prettytable import PrettyTable

help(PrettyTable)