MCS 260 – Introduction to Computer Science – Fall 2020 – Emily Dumas

Project 2 – Due Fri Oct 9 at 6:00pm Central time

1. Overview

This document describes a computer program you must write in Python and submit to Gradescope.

For this project, the program will simulate the operation of a vending machine that sells snacks, accepting coins and dispensing products and change.

In an actual vending machine, a program running on an embedded computer accepts input from buttons (e.g. a numeric keypad) and devices such as coin acceptors, and its output consists of signals that control a display, actuate motors to dispense products, etc.. However, for this project you will build a *simulation* where all of the necessary input signals are replaced by keyboard input from a user, and all output signals and actions are indicated by printing text to the terminal.

2. Preview

Here is a sample session of using the kind of vending machine simulator you are going to write. This example is meant to convey the general idea. A longer sample input and output for the program can be found in Section 7. Red boxes indicate keyboard input from the user.

```
CREDIT: $0.00
>inventory
0 Nutrition nuggets $1.00 (5 available)
1 Honey nutrition nuggets $1.20 (5 available)
2 Almonds $18.00 (4 available)
3 Nutrition nugget minis $0.70 (10 available)
4 A carrot $4.00 (5 available)
5 Pretzels $1.25 (8 available)
CREDIT: $0.00
>3
MSG: Insufficient credit
CREDIT: $0.00
>quarter
CREDIT: $0.25
>quarter
CREDIT: $0.50
>quarter
CREDIT: $0.75
>3
VEND: Nutrition nugget minis
RETURN: nickel
CREDIT: $0.00
>
```

3. Operational specification

This section describes how your program must operate.

The program will be given one command line option, which is the name of a text file containing the inventory. The format of this file is described below (Section 3.1).

The program will read this file to determine the starting inventory of snacks. It will then begin the simulation, in which it reads user commands and responds accordingly. The required commands are described in Section 3.4.

3.1. **Inventory.** Before starting the simulation, your program must open, read, and close the text file specified in the first command line argument after the script name.

This file will consists of 6 lines, each of which describes one of the six snacks available for purchase. The format of a line is

```
stock,price,name
```

where `stock` is the number of the snack available at the beginning of the simulation, `price` is the price of the snack in cents (which will be a multiple of 5), and `name` is a string not containing the character "," that describes the snack.

The order of the snacks is important, because when ordering from the machine, a snack is indicated by its 0-based index; thus snack 2 means the third line of the inventory file.

Here is a sample inventory you can use for testing:

```
6,125,Cheesy dibbles
10,115,Oat biscuits
12,75,Sugar rings
5,150,Celery crunchies
6,205,Astringent persimmon
10,95,Almond crescents
```

This inventory file is available for download from
`https://dumas.io/teaching/2020/fall/mcs260/project2/sample_inventory.txt`
This inventory indicates, for example, that snack 2 is Sugar rings, of which there are initially 12 available, each of which has a cost of $0.75.

3.2. **The simulation.** After reading the inventory, your program should enter a loop (the *command loop*) in which it does the following, repeatedly, in order:

- Print `CREDIT:` followed by the total amount of credit currently deposited in the machine, printed on the same line. The credit is initially $0.00. It should always be printed as a dollar sign, followed by a dollar amount with two digits after the decimal point.
- Print a prompt >
- Wait for one line of user input
- Perform the action associated with the user input, which may involve additional output (see Section 3.4)

Note that during the simulation, you need to keep track of the credit (the total amount of money deposited to the machine) and display it on each iteration of the loop. The remaining stock of each snack must also be tracked, as this will change as a result of purchases and restocking.

3.3. **Control logic overview.** The next section describes the commands your simulation must accept from the user. This section is a high-level description of the underlying principles of operation; for full details see Section 3.4.

The simulated vending machine allows the user to insert coins to add credit. If the credit already equals or exceeds the price of the most expensive snack in the inventory, any coin inserted will be immediately returned.

The user can select a snack by number (0–5), and if the credit currently in the machine equals or exceeds the price of the snack, then the snack is dispensed. Any change (the difference of the credit and the price) is dispensed as coins.

Finally, a maintenance worker can specify that one of the snacks is being restocked, i.e. more of that snack has been loaded into the machine. Restocked snacks are then available for purchase.

3.4. **Commands.** The simulation must support the following commands:

- `quarter` - simulates deposit of one quarter ($0.25). If the current credit is less than the most expensive item in the inventory (*including* any items that may be out of stock), the coin is accepted and credit increases by $0.25. Otherwise, the coin is rejected by printing the line

  ```
  RETURN: quarter
  ```

  to indicate that the quarter just deposited is returned.
- `dime` - simulates deposit of one dime ($0.10). The logic is the same as the quarter command, except that if the dime is not accepted, the line to be printed is

  ```
  RETURN: dime
  ```
- `nickel` - simulates deposit of one dime ($0.05). The logic is the same as the quarter command, except that if the nickel is not accepted, the line to be printed is

  ```
  RETURN: nickel
  ```
- `inventory` - display the current inventory in the format of 0-based index, followed by name, followed by price, and then a parenthetical statement of the number available, in this format:

  ```
  0 Cheesy dibbles $1.25 (6 available)
  1 Oat biscuits $1.15 (10 available)
  2 Sugar rings $0.75 (12 avilable)
  3 Celery crunchies $1.50 (5 available)
  4 Astringent persimmon $2.05 (6 available)
  5 Almond crescents $0.95 (10 available)
  ```
- Any of the digits `0`, `1`, `2`, `3`, `4`, `5` - this is a request to purchase the snack whose 0-based index in the inventory is the given integer. The action depends on the current credit and stock:
  - If the current credit is sufficient to purchase that snack, and if the stock of that snack is positive, then the machine dispenses the snack followed by change. Dispensing the snack is simulated by printing
    ```
    VEND: Name of snack
    ```
    where "Name of snack" is replaced by the name specified in the inventory. Then, returning change is simulated by printing lines such as
    ```
    RETURN: quarter
    RETURN: dime
    RETURN: nickel
    ```
    so that each line corresponds to one coin that is returned. The process for making change is subject to additional rules, described in Section 3.5.
    After a successful purchase and dispensing of change, the stock of that item decreases by one, and the credit is set to $0.00.
  - If the stock of the requested item is zero, the following line is printed:

```
MSG: Out of stock
```

The credit is unchanged, and the loop begins again immediately. (For example, if the credit was also insufficient for that item, no message is printed to that effect.)

- If the stock of the requested item is positive, but the current credit is NOT sufficient to purchase that snack, then the following line is printed:

```
MSG: Insufficient credit
```

The credit is unchanged.

- `restock` - add to the inventory of one snack. This command never appears on a line by itself, and is always followed by a space and then two integers separated by spaces. The first integer is the 0-based index of a snack, and the second is the number of additional items loaded. The effect of this command is to immediately increase the inventory of that snack. The current credit is not changed. For example, `restock 3 18` means that the inventory of snack 3 should be increased by 18.
- `return` - a request to return all currently-deposited credit. The credit should be returned to the user in the same way that change is returned after a successful purchase (see Section 3.5 for detailed rules).
- `exit` - exit the program.

3.5. **Coin return rules.** The specifications above include two situations in which coins need to be dispensed to the user:

- After a purchase, to give change
- In response to the `return` command, to return the current credit

In each case, simulated coins are dispensed by printing lines such as

```
RETURN: quarter
RETURN: dime
RETURN: nickel
```

each of which corresponds to a single coin.

The sequence of coin return lines must begin with quarters, followed by dimes, and lastly nickels.

Change must be given using the largest coins possible, so for example it is never permissible to give two or more dimes and one or more nickel, because the same change could be made with the number of quarters increased by one. For the purposes of this assignment, the machine never runs out of coins.

The following "greedy" approach will dispense coins meeting these requirements:

(1) Dispense quarters until the remaining amount to return is less than \$0.25.
(2) Dispense dimes until the remaining amount to return is less than \$0.10.
(3) Dispense nickels until the remaining amount to return is zero.

Note that unlike most real-world vending machines, these rules mean that the `return` command may give back a different set of coins than the user deposited. For example, after depositing five nickels, the `return` command would return a single quarter.

3.6. **Efficiency.** Your program must be able to complete 50 commands in less than 30 seconds, not counting the time a user takes to enter the commands. This is an extraordinarily generous time budget, as a typical solution will take at most 0.01 seconds to complete 50 commands. It is almost certain that you will not need to pay attention to the speed of any operation in your program.

However, if you perform tens of millions of unnecessary calculations in the command loop, or do something else unusual that makes your program slow to respond to commands, then the autograder will run out of time in testing your program. In this case you will not receive credit.

4. Source code specification

This section describes how your program must be written and submitted.

Your program must be contained in a single file named `vend.py`. If you submit multiple files to gradescope (not recommended), then there must be exactly one file ending with `.py`, and that file must be named `vend.py`. Gradescope will not test any Python program other than `vend.py`.

The program must be able to run with Python 3.6 (which is consistent with all programming practices taught in this course).

The only module you are allowed to import is `sys`.

As with all Python source files you submit in MCS 260, this one must have a file-level docstring, and every function must have a descriptive docstring.

The file-level docstring must be the first line of the file, and it must be followed by these comments:

- The following comment line, verbatim: `# MCS 260 Fall 2020 Project 2`
- A comment line consisting of your full name
- A comment line beginning with `# Declaration:`, and then containing a full sentence that says, in your own words, that you are the sole author of the program you are submitting and that you followed the rules from the course syllabus in preparing it. (This part can span multiple lines.)

Here is an example of an acceptable start of a source file for a project submitted by a student named Srinivasa Ramanujan:

```
"""Simulate a snack vending machine"""
# MCS 260 Fall 2020 Project 2
# Srinivasa Ramanujan
# Declaration: I, Srinivasa Ramanujan, am the sole author of this code, which
# was developed in accordance with the rules in the course syllabus.

def read_inventory(fn):
    """Load inventory and return as a dict"""
    ...
```

Your source code will be evaluated on the basis of readability. You are expected to use descriptive variable names for the most important variables (e.g. `credit` as opposed to `c`). Single-letter names are permissible for the variable of a for loop.

Comments should be included whenever the intent of a line is not immediately apparent. Comments on every line would be excessive, but it is expected that this project will involve at least 5 lines that contain descriptive comments. Judging the correct comment density involves an element of subjectivity. If you are unsure about whether your code has enough comments, just ask. (You can ask about this by submitting your code to gradescope and then emailing the instructor, or by including source code directly in an email to the instructor.)

In the review of your source code, the ability of the reviewer to understand the way your code works is important. It is theoretically possible to write an extremely complicated program that meets the operational specifications but which is impossible for a human to understand. In the code review scoring, code that is very difficult to understand may be subject to a penalty. If you are unsure about the level of understandability of your code, ask.

5. Acceptable use policy and plagiarism

The program you submit will be run by an autograder as part of its evaluation. Submitting a program that attempts to sabotage, circumvent, or examine the autograder, or which is designed to bypass course policies in any other way, is a violation of the ACCC acceptable use policy. Such violations will be reported for possible disciplinary action.

Giving or receiving aid to other students on projects in MCS 260 is prohibited. Submitting code of which you are not the sole author (for example, adapting work of another student or from a textbook or online resource) constitutes plagiarism and will be reported for possible disciplinary action.

6. Suggestions

6.1. **Starting point.** Look for sample programs from recent lectures that do things similar to parts of this project, and consider using that code as a starting point for writing your project.

6.2. **Break the simulation into smaller tasks and use functions.** For example, your project should probably have a function `read_inventory(fn)` which reads the inventory from a file with name `fn` and returns it. We recommend using a list of dicts as the return type, processing the inventory into a structure like

```
[
  {"name": "Cheesy dibbles", "stock": 6, "price": 125},
  {"name": "Oat biscuits", "stock": 10, "price": 115},
  {"name": "Sugar rings", "stock": 12, "price": 75},
  {"name": "Celery crunchies", "stock": 5, "price": 150},
  {"name": "Astringent persimmon", "stock": 6, "price": 205},
  {"name": "Almond crescents", "stock": 10, "price": 95}
]
```

Similarly, you may want to write a function `dispense_change(cents)` which will determine and print the coin types to properly dispense `cents` cents.

6.3. **Use cents.** Since the inventory prices are given in units of cents, you should probably just store all monetary quantities as ints in units of cents. In some places you'll need to divide by 100 to get the dollar amount for printing, which is fine. Using ints for cents is compatible with a general piece of programming advice: Never use floats for monetary values.

6.4. **Test using hard-coded inventory.** You may want to work on the command loop before you write the function to load the inventory. To make your program easier to test, we recommend replacing the use of the `read_inventory()` function with a static assignment during the early stages of your development. For example, while the final program will probably call

```
inventory = read_inventory(sys.argv[1])
```

before you have written `read_inventory`, you may want to develop the command loop by creating a test inventory with something like

```
inventory = [ {"name": "Cheesy dibbles", "stock": 6, "price": 125}, ... ]
```

6.5. **Start with purchase logic, handle restocking and change return later.** When developing your program, we suggest starting with a version that doesn't dispense proper change, and instead just prints a total amount that would be dispensed.

Similarly, the coin deposit and purchase commands should probably be developed and tested first. Later you can add the restocking and coin return commands. Finally, you can handle loading the inventory from a file.

In general it is extremely helpful to make a program that does *something* in the general direction of the project as early as possible, so that you can test, correct, and refine it until it meets all project specifications.

6.6. **Submit early and often.** There is no limit on the number of times you can submit your project, as long as all submissions are before the deadline. Only the last submission will count toward your score on Project 2.

The immediate availability of an autograder report on any submission you make is one of the most valuable tools you have to develop a program that meets the specifications in this document. When your submission fails any of the tests, be sure to review the autograder output to find out why. Usually, this will help you to find the part of your program that needs to be changed.

It can even be useful to submit a program you know is incomplete, just to see if it passes any tests, and to see the format of the results of the failed tests.

7. Sample input and output

In this example, the sample inventory from Section 3.1 was used.

Text inside red boxes is the keyboard input, everything else is output from the program.

```
CREDIT: $0.00
> inventory
0 Cheesy dibbles $1.25 (6 available)
1 Oat biscuits $1.15 (10 available)
2 Sugar rings $0.75 (12 available)
3 Celery crunchies $1.50 (5 available)
4 Astringent persimmon $2.05 (6 available)
5 Almond crescents $0.95 (10 available)
CREDIT: $0.00
> dime
CREDIT: $0.10
> nickel
CREDIT: $0.15
> quarter
CREDIT: $0.40
> quarter
CREDIT: $0.65
> quarter
CREDIT: $0.90
> quarter
```

```
CREDIT: $1.15
> quarter
CREDIT: $1.40
> quarter
CREDIT: $1.65
> quarter
CREDIT: $1.90
> quarter
CREDIT: $2.15
> quarter
RETURN: quarter
CREDIT: $2.15
> 4
VEND: Astringent persimmon
RETURN: dime
CREDIT: $0.00
> inventory
0 Cheesy dibbles $1.25 (6 available)
1 Oat biscuits $1.15 (10 available)
2 Sugar rings $0.75 (12 available)
3 Celery crunchies $1.50 (5 available)
4 Astringent persimmon $2.05 (5 available)
5 Almond crescents $0.95 (10 available)
CREDIT: $0.00
> restock 4 15
CREDIT: $0.00
> inventory
0 Cheesy dibbles $1.25 (6 available)
1 Oat biscuits $1.15 (10 available)
2 Sugar rings $0.75 (12 available)
3 Celery crunchies $1.50 (5 available)
4 Astringent persimmon $2.05 (20 available)
5 Almond crescents $0.95 (10 available)
CREDIT: $0.00
> dime
CREDIT: $0.10
> dime
CREDIT: $0.20
> dime
CREDIT: $0.30
> 4
MSG: Insufficient credit
CREDIT: $0.30
> return
```

```
RETURN: quarter
RETURN: nickel
CREDIT: $0.00
> exit
```

8. Revision history

- 2020-09-28 Added missing first line of output in example; added note about efficiency
- 2020-09-24 Initial publication