CODELAB I

ASSESSMENT 2: Utility App

Tutor: Ms. Lavanya Mohan

Programming Fundamentals

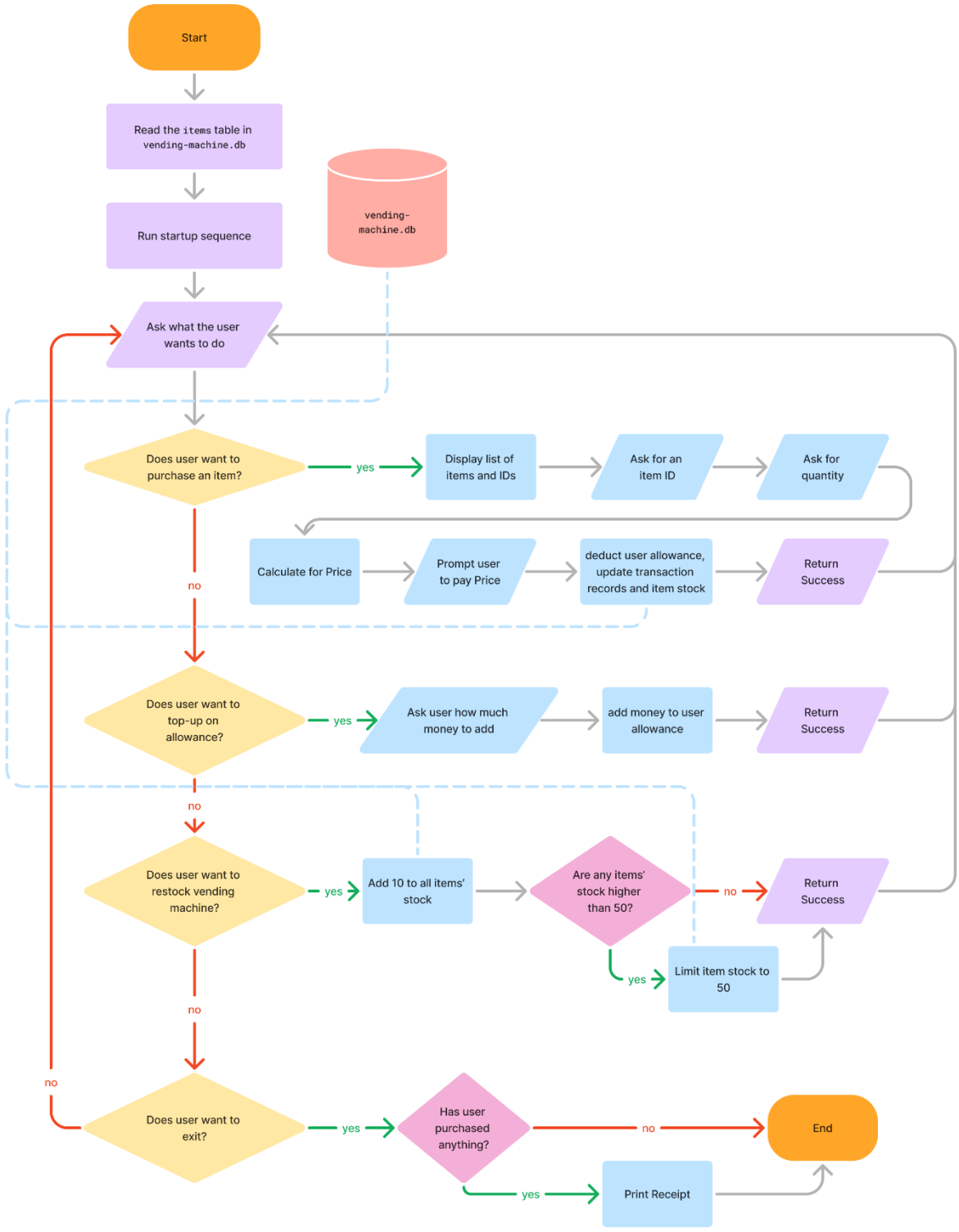
Student's Name	Vince Matthew C. Caballero
Student I.D.	04-24-027
GitHub Repository Name	creative-computing
GitHub Repository Link	https://github.com/beansebrrr/creative-computing/tree/main/introduction-to-programming/assessments/A2-vending-machine
YouTube Link	https://youtu.be/5Q-jxSGUMmU

Specifications

For this assessment, I have been tasked to develop a program that emulates a vending machine. I implemented the basic features including: a wide selection of items that can be purchased, each with their own unique I.D.; a way to select the item, ask how many they wish to buy, and dispense the correct items; and lastly, a prompt to insert enough money and it returns the appropriate change. Additionally, I added extra features such as: a startup sequence for the program, a menu interface for an organized view, an allowance system which limits what the user can buy, and a stock system on all the items.

Most importantly, I made sure that adding a new item into the vending machine is as simple as updating the database and I wouldn't need to fidget with the code any further.

System Flowcharts



This program has three main processes: **purchase**, **allowance top-up**, and **item restock**. They have been set up so that the user can run any of the functions as they please. This program also takes care of a lot of unwanted occurrences which will be discussed in detail under Walkthrough.

Walkthrough

Before I began developing the vending machine, I first thought about what I want to add into the program. I knew I needed the core components such as: a selection of items with unique identifiers for each of them; a simple money-insert system; and the purchasing sequence. Extra features I decided to implement include item stocks, a set allowance for the user, and a receipt that displays when the user exits the program.

Lastly, my number one objective in implementing this program is to make adding new items an easy task. I used a SQLite3 database to store the data of the vending machine's items and storage; I want to make sure that if new items are added into the database, the program will work as intended.

The database

I began with the item selection or menu. As mentioned above, I made use of SQLite3 to create a database of the items. I only needed one table which I named `items`. This table contains four columns:

- `id` serves as the unique identifier for each item. This will be what the user may input should they wish to purchase an item from the vending machine. This also increments automatically, meaning it doesn't have to be touched if we wish to add more items to the database.
- `name` is exactly what it is. It's the name of the item that the user can buy!
- `stock` refers to the quantity of an item that's available at the moment. If the stock reaches 0, the item cannot be purchased until the vending machine is restocked.
- `price` is the amount of cash that will be deducted from the user's balance if they purchase one of its items.

Global variables

There aren't that many global variables in this program. However, they are worth mentioning.

```
# Global Variables.
ALLOWANCE = 50.00
TOTAL_SPENT = 0
TRANSACTIONS = []

# Get list of valid IDs, will be used later on.
with sqlite3.connect(database) as temp_conn:
    global valid_ids
    temp_cursor = temp_conn.cursor()
    temp_cursor.execute("SELECT id FROM items;")

    valid_ids = [num[0] for num in temp_cursor.fetchall()]
```

Stock and restock

Every item has their own *stock*. Meaning there is a limited quantity of said item. Every time a purchase is made, the item's stock is updated in the database. Once the item's stock reaches 0, there will be no way of purchasing the item until a restock is done.

```
"""Add 10 to the stock of each item with a limit of 50."""
def restock():
    print(r"""
.-------.___
| ||||| |[_o\_
| ^^^^^ |-`  )
'-()------()-
""", flush=True)

    sleep(0.5)
    typewriter("The restock truck has arrived!")

    conn.execute("UPDATE items SET stock = stock + 10;")
    conn.execute("UPDATE items SET stock = 50 WHERE stock > 50;")
    conn.commit()

    sleep(1)
    typewriter("The vending machine has been restocked!")
    sleep(1)
```

`restock()` is a simple function which updates the database by adding 10 to all of the items' stock, and if any of the stock go beyond the limit of 50, its values are updated to said limit.

Allowance system

The allowance system is the amount of money that the user may start with. On startup, the user will have **AED 50.00** to spend how ever they want. If they try purchasing something that's above their budget, the program warns the user and brings them back.

```
"""Increase the ALLOWANCE of the user. Has a
little message if the user is a little greedy."""
def allowance_top_up():
    global ALLOWANCE

    print("You call your dad to give you more money...\n", flush=True)
    sleep(0.25)
    print_typewriter("Dad", "So, how much do you need?")
    sleep(0.5)
    top_up = get_float(">>> ")
    # Magic number.
    if top_up == 0:
        return

    if top_up >= 70:
        print_typewriter("Dad", "Don't you think that's a little too much?", end=" ")
        sleep(0.35)
        typewriter("Well, here you go anyways...", 0.025)
    else:
        print_typewriter("Dad", "Here you go buddy.", 0.025)

    sleep(0.5)
    ALLOWANCE += top_up
    sleep(1)
```

This may be frustrating for some user, so if ever they wish to increase their allowance, I added a simple function called `allowance_top_up()` which lets the user get as much allowance as they wish. The function simply takes input from the user on how much money they'd like, and then it will be added to the `ALLOWANCE` variable. Although there is no punishment for asking too much, I did add a little easter egg if the user asks for more than **AED 70.00** at once.

Receipts

The receipts keep track of all the purchases that the user has made during the entire session. When the user exits the program after making a purchase, a receipt will be printed out to the terminal. The two functions that make this up are `update_receipt()` and `print_receipt()`.

```
"""Keep track of purchased items."""
def update_receipt(name, quantity, price):
    global TRANSACTIONS

    # If item had already been bought previously, update old record
    for transaction in TRANSACTIONS:
        if name == transaction["name"]:
            transaction["quantity"] += quantity
            transaction["price"] += price
            return
    # Otherwise add new record
    else:
        TRANSACTIONS.append({
            "name" : name,
            "quantity" : quantity,
            "price" : price,
        })
```

`update_receipt()` takes the item's `name`, `quantity` purchased, and its total `price` as arguments. When the user purchases an item, the details are appended to the global variable TRANSACTIONS as a dict. But if for example the user had previously bought 3 Coca-colas and purchases one more, the existing record will be updated to show that the user purchased 4 Coca-colas.

```
"""Print record of transactions"""
def print_receipt():
    # Emulates a loading sequence
    print_typewriter("Printing your receipt", "...", separator="", delay=0.5)

    # Print all transactions into a receipt
    print("\n+-------------+ RECEIPT +-------------+\n")
    for transaction in TRANSACTIONS:
        typewriter(f"{transaction["quantity"]} {transaction["name"]}(s) : AED
{transaction["price"]:,.2f}")

    sleep(0.25)

    typewriter(f"\nTotal: AED {TOTAL_SPENT:,.2f}\nDate of Purchase: {date.today()}")
    print("\n+-------------------------------------+\n")
```

`print_receipt()` takes all the contents of `TRANSACTIONS` and prints it out into an organized receipt. It also displays the `TOTAL_SPENT` during the session, as well as the date of purchase. When the user exits the program through the `menu()`, it runs `exit_program()`, which gives the program a chance to print the receipt if and only if a transaction has been made.

```
"""Initiate a cool sequence before exiting"""
def exit_program():
    # Emulate a loading sequence
    clear_terminal()
    print_typewriter("The vending machine must go", "...", separator="", delay=0.5)

    # Only prints receipt if the user had bought anything
    if len(TRANSACTIONS) > 0:
        print_receipt()
    else:
        clear_terminal()
    sleep(0.5)
    quit()
```

Navigation

```
"""Is the first thing the user is greeted to."""
def start_screen():
    dialogues = [
        "Walking across the street, you make your\nway to something you haven't seen
here\nbefore...",
        "It's a new vending machine! Beside it is\na manual. You read its contents:",
        "\"Step 1: Enter the ID of the item you\nwish to purchase,\"",
        "\"Step 2: Enter how many you want to\nbuy.\"",
        "\"Step 3: Please be patient, the machine\nwill dispense your items shortly.\"",
        "\"Step 4: Enjoy your food and/or drinks!\"",
        "\"P.S.: If ever you need to go back or\nleave, just type the magic number '0'\"",
        "After reading the instructions, you take\nanother look at the machine...",
        "*Hmm... what should I get today?*"
    ]
    # Iterate through the dialogues
    for dialogue in dialogues:
        clear_terminal()
        print(TITLE)
        typewriter(dialogue)
        sleep(0.2)
        print("\n[Click Enter or type 'SKIP']")
        # Skip dialogue
        if input(">>> ").lower() in ["skip", "0"]:
            break
```

I wanted to make sure that navigating through the program is as easy-to-grasp as I could possibly make it. At startup, the program runs `start_screen()` that displays a sequence of text which will (hopefully) guide the user and teach them how to navigate through the program.

```
def main():
    # Only show when program first runs
    start_screen()

    while True:
        # Main interface
        clear_terminal()
        print(f"""{TITLE}
 [P] Purchase.
 [A] Top-up allowance.
 [R] Restock on goods.
 [0] Magic Number (exit & get receipt).
+--------------------------------------+
Amount spent so far: AED {TOTAL_SPENT:,.2f}
Allowance: AED {ALLOWANCE:,.2f}
""")
        match input(">>> ").upper():
            case 'P':
                make_purchase()
            case 'A':
                allowance_top_up()
            case 'R':
                restock()
            case '0':
                exit_program()
            case _:
                pass
```

I also implemented a menu in `main()` which makes it much easier to organize all of the possible functions of this vending machine program.

```
# Magic number.
if item_id == 0:
    break
```

Lastly, I have my own implementation of a “go back” button. This is what I call the Magic number (it’s `0`). Anytime that the user types `0` in a function where it’s looking for input, the program can accept `0` to “go back” or to “leave the current menu.” These bits of code are found everywhere in `vending-machine.py`, just look for the comment `# Magic number`. and they’re quite easy to spot.

The purchase sequence

The purchase sequence is the bulkiest part of the program, given that it is the main function of a vending machine. Thus, I've broken down this sequence into multiple parts. When the user wishes to purchase something, the `make_purchase()` function is run.

```
"""Initiate a transaction"""
def make_purchase():
    while True:
        clear_terminal()
        view_items()

        print(f"\nTransaction #{len(TRANSACTIONS)+1}")
        while True:
            item_id = get_id()
            # Magic number.
            if item_id == 0:
                break

            item = get_item_info(item_id)
            if item["stock"] < 1:
                print(f"{item["name"]} is not in stock!")
                continue
            elif item["price"] > ALLOWANCE:
                print("You don't have enough money to even buy one!")
                continue

            quantity = get_quantity(item)
            # Magic number.
            if quantity == 0:
                continue

            # Proceed with buying item.
            if buy(item, quantity) == True:
                break
        # Ask user if they want to do another purchase.
        # Also, magic number.
        if item_id == 0 or not get_bool("Would you like to buy something else? (y/N): "):
            break
```

This function bundles all of the first few steps in buying an item, which is selecting what item to buy and how many. Before beginning a transaction, the program displays all of the items inside the vending machine with `view_items()`.

I. View items

```
"""Print a table from vending-machine.db."""
def view_items():
    # Retrieve all contents of SQLite database
    cursor.execute("SELECT * FROM items;")
    _ = cursor.fetchall()
    data = [dict(row) for row in _]
    # Make all prices a floating-point value
    for row in data:
        row["price"] = round(float(row["price"]), 2)
    # Print table
    table = tabulate(data, headers="keys", floatfmt=".2f", tablefmt="psql")
    print(table)
```

First off, it fetches all the contents of the database as a *list* of *dicts* under the variable `data`. Next, the program changes all of the items under the `price` column to a *float*. SQLite deals with

numbers differently, which meant that if I tried entering a float like `5.00`, SQLite would simplify it as `5`. For formatting (and aesthetic) reasons, I had to add the `.00` by myself.

Lastly, it prints the contents of the database. I didn't like the varying width of each row when it does a basic loop through a print statement—making the text a little annoying to read—which is why I decided to import the **tabulate** module. With this, the contents of the database can be formatted into a table before printing.

Now we move on to the transaction.

Item selection

Firstly, the program retrieves the desired item's ID with the function `get_id()`, which the item's information is then read with `get_item_info()`. If the item is found to be out of stock, the program will loop back to `get_id()`; otherwise, the program will carry on to `get_quantity()`. Lastly, the program will proceed to `buy()` an X amount of the item. If the purchase is successful, The user will be asked if they would like to purchase anything else. If the user types yes, It will initiate another purchase. Otherwise, the program will go back to `main()`.

```
"""Only accept IDs found in valid_ids"""
def get_id():
    # Prompt for a valid ID
    while True:
        item_id = get_int("Enter item's ID: ")
        if item_id in valid_ids or item_id == 0:
            return item_id
        print("Invalid ID.\n")
```

`get_id()` prompts the user for an *integer*. If the user enters a number which isn't present in the `valid_ids` *list*, the program will prompt the user again until a valid id (or the magic number `0`) is entered.

```
"""Return a dict of the item's information"""
def get_item_info(item_id):
    cursor.execute("SELECT * FROM items WHERE id = ?;", (item_id,))
    return dict(cursor.fetchone())
get_item_info() will take the item's ID as an argument, and returns the row which corresponds to
the provided ID. It then returns a dict of the item's information.
"""Ask for quantity"""
def get_quantity(item):
    while True:
        # Ask how many to buy
        quantity = get_int(f"How many {item["name"]} would you like to purchase? ")
        # Stops user from buying if insufficient stock
        if quantity < 0:
            print(f"You cannot buy {quantity} {item["name"]}(s).\n")
        elif item["stock"] < quantity:
            print(f"There are not enough {item["name"]} in stock.\n")
        else:
            return quantity
```

`get_quantity()` takes the *dict* `item` as an argument. It will prompt the user how many items would they like to purchase. However, it will reject the user's input if either the user inputs a negative integer or there aren't enough of the item in stock to buy that many.

Checkout

```
"""Is the checkout part. This is the
chonkiest function in the program"""
def buy(item, quantity):
    global ALLOWANCE, TOTAL_SPENT

    while True:
        # Calculate price.
        price = round((item["price"] * quantity), 2)
        if price <= ALLOWANCE:
            break
        # If too expensive, reprompt for quantity and recalculate price.
        print(f"You need AED {price:,.2f}, but you only have AED {ALLOWANCE:,.2f}.\n")
        quantity = get_quantity(item)
        # Magic number.
        if quantity == 0:
            print("Transaction terminated.\n")
            return

    # Prompt user to pay to machine
    cash_paid = pay(price)
    # Magic number.
    if cash_paid == 0:
        print("Transaction terminated.\n")
        return

    # Update database
    conn.execute(f"UPDATE items SET stock = stock - {quantity} WHERE id = ?;", (item["id"],))
    conn.commit()

    # Update globals
    ALLOWANCE -= price
    TOTAL_SPENT += price
    update_receipt(item["name"], quantity, price)

    # Purchase successful
    print("\n+---+ Purchase Successful! +---+\n")
    sleep(0.25)

    print_typewriter("The machine returned", f"AED {(cash_paid - price):,.2f}", delay=0.075)
    sleep(0.5)
    print_typewriter("Your new allowance", f"AED {ALLOWANCE:,.2f}", delay=0.075)
    return True
```

Now the program finally moves on to the checkout phase. First, the price of the items are calculated right away. If in case the price goes beyond the user's `ALLOWANCE` and they can't pay for it, the user will be sent back to `get_quantity()`. The program then moves on to `pay()`. If this part is done successfully; the item's stock will be updated in the database, the user's `ALLOWANCE` and `TOTAL_SPENT` will be deducted and increased respectively, and this record will update the list `TRANSACTIONS` through `update_receipt()` (more about this under Receipts). Finally, it will inform the user of a successful transaction and displays the user's change (`cash_paid - price`) and the updated `ALLOWANCE`.

```
"""Prompts user to insert money until they add enough"""
def pay(price):
    typewriter(f"\nPlease pay at least AED {price:,.2f} in the machine.")
    sleep(0.75)

    while True:
        # Ask user to insert money
        cash = get_float("Insert money >>> ")

        # Reprompts for cash if not enough money
        # Also allow user to input 0
```

```
        if cash > ALLOWANCE and cash != 0:
            print(f"You only have AED {ALLOWANCE:,.2f}.")
        elif cash < price and cash != 0:
            print("You did not put enough money.")
        else:
            return cash
```

`pay()` takes the required price as an argument which the function will ask the user to pay with "cash". It will not accept the user's input and prompt them again if they pay an amount that surpasses their `ALLOWANCE`, as well as if the user doesn't put enough money to cover the `price`. If all the conditions are met, the function will return `cash`.

Miscellaneous functions

Scattered throughout the code are some unfamiliar functions which I haven't explained yet. These are functions which are not specific to the vending machine program and serve as "helper functions". These make the program a little less cluttered and easier to read. All of these can be found at the bottom of the file under a commented section called `Miscellaneous Functions`.

Three of these functions deals with user input. Python's `input()` function, though useful, returns values as just *strings*, and is not ideal when the program expects a different data type. Which is why below, I've used code I learned from **CS50x** which takes user input and only returns the appropriate data type. Namely, I implemented `get_int()`, `get_float()`, and `get_bool()`.

```
"""only allows an int input"""
def get_int(prompt) -> int:
    while True:
        try:
            return int(input(prompt))
        except ValueError:
            pass
```

`get_int()` starts with prompting the user for an input, after which it will try to convert it into an *int*. If it succeeds to do so, it will return the converted value. Else, the function will loop back and prompt the user again until it works.

```
"""only allows a float input"""
# Rounded to 2 decimals since we're dealing with money
def get_float(prompt) -> float:
    while True:
        try:
            return round(float(input(prompt)), 2)
        except ValueError:
            pass
```

`get_float()` works in a similar vein to `get_int()`. However, instead of ending by just returning the *float*, I made sure that **the returned value is also rounded to 2 decimal places**. Because this program deals with money, it wouldn't be ideal if the user can input AED 3.14159265 into their balance.

```
"""only allows yes or no inputs"""
def get_bool(prompt) -> bool:
    while True:
        _ = input(prompt).lower()
        if _ in ["y", "yes", "t", "true", "1"]:
            return True
        elif _ in ["n", "no", "f", "false", "0"]:
            return False
```

The `get_bool()` function works differently from the previous two. Instead of converting strings into a bool, the function prompts the user for **yes** or **no**. Actually, the function accepts a number of strings that are in similar vein with **yes** or **no** such as **true** or **false**, **y** or **n**, **t** or **f**, and even **1** or **0**. I also eliminated case-sensitiveness by always converting the user's input to lowercase.

These next two functions are variations of printing text, purely for aesthetics: `typewriter()` and `print_typewriter()`.

```
"""emulate a typewriter effect when printing text"""
def typewriter(text, delay=0.01, end="\n"):
    for char in text:
        print(char, end="", flush=True)
        sleep(delay)
    print(end, end="")
```

`typewriter()` prints text letter-by-letter with a configurable delay between each letter being printed. It loops through the string and prints them one by one, with `time.sleep()` adding a delay between each print. At the end of the loop, a newline is printed by default but can be changed by adding the argument to the function.

```
"""character dialogue, with name and text"""
def print_typewriter(printed, typewritten, delay=0.025, separator=": ", end="\n"):
    print(printed, end=separator, flush=True)
    sleep(0.25)
    typewriter(typewritten, delay=delay, end=end)
```

`print_typewriter()` combines the `print()` and typewriter() function together. In this program, it has a few uses. But most importantly, it emulates character dialogue. It first prints the text printed and appends a "`: `" by default. For example, it would output something like "`Juliana: `". It then calls `typewriter()` to print Juliana's dialogue.

The last miscellaneous function is `clear_terminal()` which does exactly what it says. It clears up all of the text in the terminal. This is used to clear a lot of the clutter, especially since this program occupies a lot of vertical real estate.

```
"""clears the text from the console"""
def clear_terminal():
    system("cls" if name == "nt" else "clear")
```

This function runs a terminal command using `os.system()`. It executes `cls` if the `os.name()` is `nt` being Windows; and otherwise runs `clear` for macOS or Linux systems.

Reflection

In developing the vending machine program, I focused a lot on ease-of-use and intuitiveness—making sure that anything that the user can do has been accounted for. I solved as many unwanted outcomes as I could find, polished the look and feel, and did my best to not complicate the experience. I think I've done well in that regard.

However, I do believe that there is room for improvement, especially on cleanliness of code. As it stands, I am not completely satisfied with how some areas of the code are a bit cluttered, and I know I could do better. I'm also sure there are other solutions to the hundreds of hurdles I faced, and the only way I can find them is to continue learning what I don't know yet, hone the skills I already have, and use them in ways a creative mind would. This little project made me truly realize there are many ways to tackle a problem in programming, and the only limit is the programmer's imagination (and the power of their CPU).

Appendix

```
"""
Vince Matthew C. Caballero
Assessment 2: Vending Machine
"""

from datetime import date
from os import name, system
from pathlib import Path
from time import sleep
import sqlite3

# This might need to be pip install-ed.
from tabulate import tabulate

# Get database directory
root_dir = Path(__file__).resolve().parent
database = root_dir/"vending-machine.db"

# Check if database exists
if database.exists() == False:
    print("Error: No such file as vending-machine.db! Please put vending-machine.py and vending-machine.db in the same folder.")
    quit()

# SQLite connection.
conn = sqlite3.connect(database)
conn.row_factory = sqlite3.Row
cursor = conn.cursor()

# Global Variables.
ALLOWANCE = 50.00
TOTAL_SPENT = 0
TRANSACTIONS = []

# Get list of valid IDs, will be used later on.
with sqlite3.connect(database) as temp_conn:
    global valid_ids
    temp_cursor = temp_conn.cursor()
    temp_cursor.execute("SELECT id FROM items;")

    valid_ids = [num[0] for num in temp_cursor.fetchall()]

# I'm gonna use this a few times, but this
# won't be manipulated.
```

```
TITLE = r""" __      __                  _ _
 \ \    / /__ _ __    __| (_)_ __   __ _ _
  \ \ / / _ \ '_ \ /  _` | | '_ \ / _` |
   \ V /  __/ | | | (_| | | | | | (_| |
    \_/ \___|_| |_|\__,_|_|_| |_|\__, |
  |  \/  | __ _  ___| |__ (_)_ __ |___/
  | |\/| |/ _` |/ __| '_ \| | '_ \ / _ \
  | |  | | (_| | (__| | | | | | | |  __/
  |_|  |_|\__,_|\___|_| |_|_|_| |_|\___|

+--------------------------------------+"""

def main():
    # Only show when program first runs
    start_screen()

    while True:
        # Main interface
        clear_terminal()
        print(f"""{TITLE}
 [P] Purchase.
 [A] Top-up allowance.
 [R] Restock on goods.
 [0] Magic Number (exit & get receipt).
+--------------------------------------+
Amount spent so far: AED {TOTAL_SPENT:,.2f}
Allowance: AED {ALLOWANCE:,.2f}
""")
        match input(">>> ").upper():
            case 'P':
                make_purchase()
            case 'A':
                allowance_top_up()
            case 'R':
                restock()
            case '0':
                exit_program()
            case _:
                pass

"""Initiate a transaction"""
def make_purchase():
    while True:
        clear_terminal()
        view_items()

        print(f"\nTransaction #{len(TRANSACTIONS)+1}")
        while True:
            item_id = get_id()
            # Magic number.
            if item_id == 0:
                break

            item = get_item_info(item_id)
            if item["stock"] < 1:
                print(f"{item["name"]} is not in stock!")
                continue
            elif item["price"] > ALLOWANCE:
                print("You don't have enough money to even buy one!")
                continue

            quantity = get_quantity(item)
            # Magic number.
            if quantity == 0:
                continue

            # Proceed with buying item.
            if buy(item, quantity) == True:
                break
        # Ask user if they want to do another purchase.
```

```
        # Also, magic number.
        if item_id == 0 or not get_bool("Would you like to buy something else? (y/N): "):
            break

"""Is the checkout part. This is the
chonkiest function in the program"""
def buy(item, quantity):
    global ALLOWANCE, TOTAL_SPENT

    while True:
        # Calculate price.
        price = round((item["price"] * quantity), 2)
        if price <= ALLOWANCE:
            break
        # If too expensive, reprompt for quantity and recalculate price.
        print(f"You need AED {price:,.2f}, but you only have AED {ALLOWANCE:,.2f}.\n")
        quantity = get_quantity(item)
        # Magic number.
        if quantity == 0:
            print("Transaction terminated.\n")
            return

    # Prompt user to pay to machine
    cash_paid = pay(price)
    # Magic number.
    if cash_paid == 0:
        print("Transaction terminated.\n")
        return

    # Update database
    conn.execute(f"UPDATE items SET stock = stock - {quantity} WHERE id = ?;",
(item["id"],))
    conn.commit()

    # Update globals
    ALLOWANCE -= price
    TOTAL_SPENT += price
    update_receipt(item["name"], quantity, price)

    # Purchase successful
    print("\n+---+ Purchase Successful! +---+\n")
    sleep(0.25)

    print_typewriter("The machine returned", f"AED {(cash_paid - price):,.2f}",
delay=0.075)
    sleep(0.5)
    print_typewriter("Your new allowance", f"AED {ALLOWANCE:,.2f}", delay=0.075)
    return True

"""Ask for quantity"""
def get_quantity(item):
    while True:
        # Ask how many to buy
        quantity = get_int(f"How many {item["name"]} would you like to purchase? ")
        # Stops user from buying if insufficient stock
        if quantity < 0:
            print(f"You cannot buy {quantity} {item["name"]}(s).\n")
        elif item["stock"] < quantity:
            print(f"There are not enough {item["name"]} in stock.\n")
        else:
            return quantity

"""Prompts user to insert money until they add enough"""
def pay(price):
    typewriter(f"\nPlease pay at least AED {price:,.2f} in the machine.")
    sleep(0.75)

    while True:
        # Ask user to insert money
        cash = get_float("Insert money >>> ")
```

```
182.
183.         # Reprompts for cash if not enough money
184.         # Also allow user to input 0
185.         if cash > ALLOWANCE and cash != 0:
186.             print(f"You only have AED {ALLOWANCE:,.2f}.")
187.         elif cash < price and cash != 0:
188.             print("You did not put enough money.")
189.         else:
190.             return cash
191.
192. """Increase the ALLOWANCE of the user. Has a
193. little message if the user is a little greedy."""
194. def allowance_top_up():
195.     global ALLOWANCE
196.
197.     print("You call your dad to give you more money...\n", flush=True)
198.     sleep(0.25)
199.     print_typewriter("Dad", "So, how much do you need?")
200.     sleep(0.5)
201.     top_up = get_float(">>> ")
202.     # Magic number.
203.     if top_up == 0:
204.         return
205.
206.     if top_up >= 70:
207.         print_typewriter("Dad", "Don't you think that's a little too much?", end=" ")
208.         sleep(0.35)
209.         typewriter("Well, here you go anyways...", 0.025)
210.     else:
211.         print_typewriter("Dad", "Here you go buddy.", 0.025)
212.
213.     sleep(0.5)
214.     ALLOWANCE += top_up
215.     sleep(1)
216.
217. """Add 10 to the stock of each item with a limit of 50."""
218. def restock():
219.     print(r"""
220. .-------.___
221. | ||||| |[_o\_
222. | ^^^^^ |- `  )
223. '-()------()-
224. """, flush=True)
225.
226.     sleep(0.5)
227.     typewriter("The restock truck has arrived!")
228.
229.     conn.execute("UPDATE items SET stock = stock + 10;")
230.     conn.execute("UPDATE items SET stock = 50 WHERE stock > 50;")
231.     conn.commit()
232.
233.     sleep(1)
234.     typewriter("The vending machine has been restocked!")
235.     sleep(1)
236.
237. """Keep track of purchased items."""
238. def update_receipt(name, quantity, price):
239.     global TRANSACTIONS
240.
241.     # If item had already been bought previously, update old record
242.     for transaction in TRANSACTIONS:
243.         if name == transaction["name"]:
244.             transaction["quantity"] += quantity
245.             transaction["price"] += price
246.             return
247.     # Otherwise add new record
248.     else:
249.         TRANSACTIONS.append({
250.             "name" : name,
251.             "quantity" : quantity,
```

```
252.             "price" : price,
253.         })
254.
255. """Only accept IDs found in valid_ids"""
256. def get_id():
257.     # Prompt for a valid ID
258.     while True:
259.         item_id = get_int("Enter item's ID: ")
260.         if item_id in valid_ids or item_id == 0:
261.             return item_id
262.         print("Invalid ID.\n")
263.
264. """Return a dict of the item's information"""
265. def get_item_info(item_id):
266.     cursor.execute("SELECT * FROM items WHERE id = ?;", (item_id,))
267.     return dict(cursor.fetchone())
268.
269. """Print a table from vending-machine.db."""
270. def view_items():
271.     # Retrieve all contents of SQLite database
272.     cursor.execute("SELECT * FROM items;")
273.     _ = cursor.fetchall()
274.     data = [dict(row) for row in _]
275.     # Make all prices a floating-point value
276.     for row in data:
277.         row["price"] = round(float(row["price"]), 2)
278.     # Print table
279.     table = tabulate(data, headers="keys", floatfmt=".2f", tablefmt="psql")
280.     print(table)
281.
282. """Is the first thing the user is greeted to."""
283. def start_screen():
284.     dialogues = [
285.         "Walking across the street, you make your\nway to something you haven't seen
here\nbefore...",
286.         "It's a new vending machine! Beside it is\na manual. You read its contents:",
287.         "\"Step 1: Enter the ID of the item you\nwish to purchase,\"",
288.         "\"Step 2: Enter how many you want to\nbuy.\"",
289.         "\"Step 3: Please be patient, the machine\nwill dispense your items shortly.\"",
290.         "\"Step 4: Enjoy your food and/or drinks!\"",
291.         "\"P.S.: If ever you need to go back or\nleave, just type the magic number '0'\"",
292.         "After reading the instructions, you take\nanother look at the machine...",
293.         "*Hmm... what should I get today?*"
294.     ]
295.     # Iterate through the dialogues
296.     for dialogue in dialogues:
297.         clear_terminal()
298.         print(TITLE)
299.         typewriter(dialogue)
300.         sleep(0.2)
301.         print("\n[Click Enter or type 'SKIP']")
302.         # Skip dialogue
303.         if input(">>> ").lower() in ["skip", "0"]:
304.             break
305.
306. """Initiate a cool sequence before exiting"""
307. def exit_program():
308.     # Emulate a loading sequence
309.     clear_terminal()
310.     print_typewriter("The vending machine must go", "...", separator="", delay=0.5)
311.
312.     # Only prints receipt if the user had bought anything
313.     if len(TRANSACTIONS) > 0:
314.         print_receipt()
315.     else:
316.         clear_terminal()
317.     sleep(0.5)
318.     quit()
319.
320. """Print record of transactions"""
```

```
321. def print_receipt():
322.     # Emulates a loading sequence
323.     print_typewriter("Printing your receipt", "...", separator="", delay=0.5)
324.
325.     # Print all transactions into a receipt
326.     print("\n+-------------+ RECEIPT +-------------+\n")
327.     for transaction in TRANSACTIONS:
328.         typewriter(f"{transaction["quantity"]} {transaction["name"]}(s) : AED
{transaction["price"]:,.2f}")
329.
330.     sleep(0.25)
331.
332.     typewriter(f"\nTotal: AED {TOTAL_SPENT:,.2f}\nDate of Purchase: {date.today()}")
333.     print("\n+-------------------------------------+\n")
334.

335. """+-------------+ MISCELLANEOUS FUNCTIONS +-------------+"""
336.
337. """only allows an int input"""
338. def get_int(prompt) -> int:
339.     while True:
340.         try:
341.             return int(input(prompt))
342.         except ValueError:
343.             pass
344.
345. """only allows a float input"""
346. # Rounded to 2 decimals since we're dealing with money
347. def get_float(prompt) -> float:
348.     while True:
349.         try:
350.             return round(float(input(prompt)), 2)
351.         except ValueError:
352.             pass
353.
354. """only allows yes or no inputs"""
355. def get_bool(prompt) -> bool:
356.     while True:
357.         _ = input(prompt).lower()
358.         if _ in ["y", "yes", "t", "true", "1"]:
359.             return True
360.         elif _ in ["n", "no", "f", "false", "0"]:
361.             return False
362.
363. """emulate a typewriter effect when printing text"""
364. def typewriter(text, delay=0.01, end="\n"):
365.     for char in text:
366.         print(char, end="", flush=True)
367.         sleep(delay)
368.     print(end, end="")
369.
370. """character dialogue, with name and text"""
371. def print_typewriter(printed, typewritten, delay=0.025, separator=": ", end="\n"):
372.     print(printed, end=separator, flush=True)
373.     sleep(0.25)
374.     typewriter(typewritten, delay=delay, end=end)
375.
376. """clears the text from the console"""
377. def clear_terminal():
378.     system("cls" if name == "nt" else "clear")
379.
380. main()
381.
```