READ ME

Summary: The following document will explain the various Python documents/modules and provide instructions for its use. This script uses Django predominantly.

First, to run the script, make sure you have Python installed (ideally the most recent version), at least version 3.4. Please refer to the following website, and it will provide different options based on your operating system:

https://www.python.org/downloads/

This distribution should also install pip (Python's package installer). To check if pip was installed, enter the following command in Command Prompt/Terminal:

python -m ensurepip --upgrade

Second, you will need to download the Python package "Django". In Command Prompt/Terminal, enter the following command

python -m pip install Django

Sometimes, normally for windows (depending on the version you first installed), you may need to replace "python" with "py". For further assistance, please refer to https://docs.djangoproject.com/en/4.0/topics/install/

Now that we have the required packages installed, download the Github respository:

https://github.com/josephouchie/Fetch-Rewards-Assessment-Joseph-Ouchie.git

Navigate (change directory) to the **fetch_assessment** file's directory in Command Prompt/Terminal:

cd … > fetch_assessment

WARNING: There should be nothing else in this directory, as there is another **fetch_assessment** folder in a directory with the folder **fetch_points** and the files *manage.py* and *db.sqlite3*

To execute the script, enter following command in the **fetch_assessment** file's directory:

python manage.py runserver

Then open a new tab in your web browser and go to **localhost:8000** (http://127.0.0.1:8000/) to view the webpage, using your computer as its server.

To execute the Unit Test script file (...\fetch_assessments\fetch_points\tests.py), enter the following command:

python manage.py test fetch_points

If you wish to reset the database, you'll need to do the following:

- Delete the db.sqlite3 file (in the fetch_assessment folder) and the 0001_initial.py file (in the Migrations folder)
- The run the following commands in Command Prompt/Terminal:
  - python manage.py makemigrations
  - python manage.py migrate

Navigating the "website"

The top-left has the form fields where the user can enter transactions on the top (the Transactions form) or spend their points (the Spending form) below it.

The top-right of the website shows the total points (which, in theory, is what the customer would be seeing) and below it shows the total points associated with each payer (which, in theory, the customer should NOT be seeing). This is not hard-coded to reject negative numbers, as it accepts the calculated balances in the *calculations.py* file. It relies on proper front-end usage (which in deployment would have front-end validation to reject improper entries).

The bottom shows the "Transactions Ledger" which displays all the transactions in the database (as a result of entries entered in the Transactions form and entries generated as a result of the Spending form) in the form "Payer: @@@ , Points: ### , Timestamp: YYYY-MM-DDTHH:MM:SSZ"

- Transactions form:
  - The "Payer" entry field is very sensitive to characters, so please avoid excess spaces and keep the capitalization standardized
    - i.e "Dannon" will be considered a different entity than "DANNON" and "dannon" is different than " dannon" (notice the extra space!)
    - The maximum characters allowed is 50 (this was arbitrarily chosen)
  - You are also able enter negative numbers when making a transaction (for purposes of seeding the database. Please do not enter a negative number that is more than what that payer has in their balance).
    - Do not press Submit while this entry field is blank (it will reject letters though).
  - Enter the "Timestamp" in the following format: YYYY-MM-DDTHH:MM:SSZ.
    - Since this accepts a text response, it will accept any alphanumeric format that you choose (including blank!), which may disrupt the calculations when executing the program since the calculation "filters" based on the left-most alphanumeric!
    - The maximum characters should be 20.
  - When you have entered all three fields (Payer, Points, Timestamp) press "Submit" to enter the transaction into the database
    - Please double-check the transaction data before entering, as this action cannot be undone without resetting the database!

- Spending form:
  - The "Points" field is another Integer form that also accepts negative numbers.
    - There is no current try/catch validation for this, so please do not enter a negative number or leave it blank when you press submit!

  WARNING: If you attempt to enter an amount that exceeds the total points available, nothing will happen, except that it will clear the entry fields!
  - Enter the "Timestamp" in the following format: YYYY-MM-DDTHH:MM:SSZ.
    - Since this accepts a text response, it will accept any alphanumeric format that you choose (including blank!), which may disrupt the calculations when executing the program since the calculation "filters" based on the left-most alphanumeric!
    - The maximum characters should be 20.
  - When you have entered both fields (Points, Timestamp) press "Submit" to enter the transaction into the database
    - Please double-check the transaction data before entering, as this action cannot be undone without resetting the database!

About the Program Files

Within the **fetch_assessments** folder:

The folder **fetch_assessment**s contains the required *.py* files to run the rest of the program. These are generated by default in Django, with a couple of changes to account for program-specific code: "fetch_points.apps.FetchPointsConfig" should be within the “INSTALLED APPS” in the *settings.py* file

- *path('',include('fetch_points.urls'))* should be included in the *urlpatterns* list in the *urls.py* file

Within the **fetch_points** folder:

- Under the **templates** folder, you should find the *home.html* file. This is the HTML file that is rendered to the request (i.e. opening the website).

- *apps.py*
  - This is the application file that is passed to the **fetch_assessment** folder to run the script of this user-defined program

- *admin.py*
  - This is used to define Administrators (Super Users) for the website who, upon log in, are able to make overall changes to content on the website. It was not used for purposes of this assignment.

- *models.py*
  - This defines the format of our entries in our database (each format is define by each class)
  - The only type of entry stored in our database will be a transaction, which has the fields:
    - “payer” in text format
    - “points” in integer format
    - “timestamp” in text format
      - I understand that you may have wanted a form of **datetime** format, but I couldn’t write an elegant solution that would store it in the format YYYY-MM-DDTHH:MM:SSZ

- *views.py*
  - This module has the main function *add_spend(request)*, where “request” is the request from the website in the form of a POST.
  - When a transaction is entered and sent from the website, the *request.POST.get()* retrieves the information based on the associate POST “name” (i.e. payer, points, timestamp), this then gets saved as a transaction in the database
  - When a spending attempt is entered, the same retrieves the spending point amount and timestamp enters that information to the calculations.py module (see calculations.py)
  - After each transaction or spending attempt, the total balances for each payer is recalculated
  - The function final returns a *render()* HTTP response to the *home.html* file passing the total balances and transaction data as “context”. This allows the webpage to display the transactions ledger and calculated/updated balances for each payer.

- *urls.py*
  - This file (different than the one in the **fetch_assessment** folder) generates the path to the main function *add_spend()* in the *views.py* file

- *calculations.py*
  - This file has two main functions:
  - *spend(sp_points,sp_timestamp)*
    - This function is used to sort the transactions by the latest date and assign spent points to “use up” the latest transaction (First-In, First-Out)
    - “sp_points” is the spend points entered (as a positive number) and “sp_timestamp” is the timestamp of the spending attempt
    - The function first retrieves the total balances for each payer and calculates the total points available out of all the payers combined.
    - Then, it check to see if the spend point is less than the total points available
      - This prevents overspending available points, by returning an “Error!” text value. However, this text value wasn’t used in the website for the sake of time constraints, but I would’ve liked it to be passed back to the *views.py* module to initiate an error pop-up message on the website.
    - The function then calculates the total negative values (debits) in the Transactions database and then it assigns previous debits to the latest transactions
      - This determines which transactions were already spent!
    - Once these previous debits are exhausted, the function then determines if the remainder of the transaction is less, equal to, or greater than the current spending amount (“sp_points”)
      - If the remainder is less than current spending amount, then a debiting (negative) transaction tuple is appended to the transactions list and it reiterates
        - The tuple uses the payer of the current Transaction in the database as the payer for this new transaction in the following format:
          - (payer, debiting value, sp_transaction)
      - If the remainder is greater than or equal to the current spending amount, then a final debiting (negative) transaction is appended to the transactions list and it breaks the “for” loop
    - The function then converts the tuple to a dictionary using the *Counter()* function and proceeds the sum the total debiting values associated to each payer
      - Without this, each exhausted transaction would be tied to a corresponding negative transaction(s), which wouldn’t affect the total balances but it creates unnecessary entries that could be aggregated.
    - The function finally returns a “Done!” text value, which isn’t used anywhere else in the script.
  - *find_balances()*
    - This aggregates the total points associated with each payer
    - First it filters the Transactions database for each 'payer' and sums the points for each transaction entered for that payer
    - the *.annotate()* creates a QuerySet (a list of nested dictionaries, with each entry having the following format: {'payer': total}) which is returned by the function

- *tests.py*
  - This is the Unit Test file that contains 14 tests (each test defined as a function beginning with “test_”
  - Each individual test generates its own temporary database using the alias “default”, which is subsequently destroyed after the tests are ran.

- The test validate using the function self.assertEqual(<Variable>, <Value>). This tests to see if the value associated with the <Variable> is equal to the value (text, integer, or alphanumeric) directly entered in place of <Value>.
- 14 tests:
  - *test_og_spend*
    - Test the original situation outlined in the instruction document
  - *test_multiple_trans_sequential*
    - Test to see if the program reacts properly to multiple transactions of the same 'payer' in a row.
    - In this case, “DANNON” had three positive transactions in row prior to the spending attempt.
  - *test_multiple_spend_before*
    - Test to see how the program reacts to multiple spending periods before the new spending attempt
    - In this case, “DANNON” had two spending (negative) transactions prior to the spending attempt.
  - *test_multiple_spends*
    - Test to see how the program reacts to two new spending attempts in a row
  - *test_liquidate_before*
    - Test to see how the program reacts to a 'payer' being liquidated (i.e. balance = 0) before the new spending attempt
    - In this case, “DANNON” get liquidated prior to the spending attempt
  - *test_multiple_liquidate_before*
    - Test to see how the program reacts to multiple 'payers' being liquidated before the new spending attempt
    - In this case, “DANNON” and “UNILEVER get liquidated prior to the new spending attempt
  - *test_multiple_liquidate*
    - Test to see how the program reacts to multiple 'payers' being liquidated during the new spending attempts
      - The original (1st) test case liquidates “UNILEVER” in the new spending attempt, so there was no need to write a separate test case for a single liquidation.
    - In this case, “DANNON” and “UNILEVER get liquidated during the new spending attempt
  - *test_all_liquidate*
    - Test to see how the program reacts to all 'payers' being liquidated in a single new spending attempt
    - All payer balances should be equal to zero after the spending attempt!
  - *test_overspend*
    - Test to if the program rejects overspending points
    - The “new_spend_status” should equal “Error!”, since this is the return message from the *calculations.spend()* function for overspending attempts
    - Also tests to see if the total point balances are still the same (since nothing should have happened)
  - *test_overspend_no_entries*
    - Test to if the program rejects overspending points with no previous entries entered
    - The “new_spend_status” should equal “Error!”, since this is the return message from the *calculations.spend()* function for overspending attempts
    - This doesn’t test the total balances, since there shouldn't be any!

- *test_trans_after_one_liquidate*
  - Test to see how the program reacts a new transaction (gaining points) after that 'payer' was liquidated
  - Since UNILEVER was previously liquidated in the original spending attempt (of 5000 points), UNILEVER gets an additional 1000 points after this spending period
  - The 3 transactions as a result of the original spending attempt are NOT checked since these were checked in the original (1st) test case
- *test_trans_after_multiple_liquidate*
  - Test to see how the program reacts new transactions (gaining points) after multiple 'payers' were liquidated
  - In this case, the new spending attempt eliminates both UNILEVER and MILLER COORS by adjusting the spending amount to 10300, which would yiled the following transactions:
    {‘payer’: DANNON, ‘points’: -100, ‘timestamp’: 2022-01-31T10:00:00Z}
    {‘payer’: UNILEVER, ‘points’: -200, ‘timestamp’: 2022-01-31T10:00:00Z}
    {‘payer’: MILLER COORS, ‘points’: -10000, ‘timestamp’: 2022-01-31T10:00:00Z}
  - This test specifically validates if the new (positive) transactions are entered correctly
- *test_trans_after_all_liquidate*
  - Test to see how the program reacts to new transactions (gaining points) after all 'payers' get liquidated in the new spending attempt
  - In this case, all three payers will each have a positive transaction to re-seed their balances
  - This test specifically validates if the new (positive) transactions are entered correctly
- *test_multiple_trans_and_spend*
  - Test to see how the program multiple (3) transactions (gaining points) and multiple (3) spending attempts
  - The test leverages the original transactions and spending attempt, then it proceeds to make a new (positive) transaction for each payer
    - After the first new (positive) transaction, it is followed by two sequential spending attempts, which is then followed by two new sequential (positive) transactions
  - This test validates all transactions, including the original transactions