Criterion B: Design Overview

To answer Jerry’s need, my product will be **a Python module** with functionalities of collecting and processing stock data.

**1. Database organization**

**Data source: Yahoo Finance**

I will collect all data from Yahoo Finance (finance.yahoo.com). Below is robots.txt of Yahoo Finance (finance.yahoo.com/robots.txt). Robots.txt is a protocol which specifies what content on the site should and should not be acquired by Web Spyder programs. Yahoo Finance’s robots.txt implies that I can use my own Web Spyder to collect data from it, so my Web Spyder will be legal.

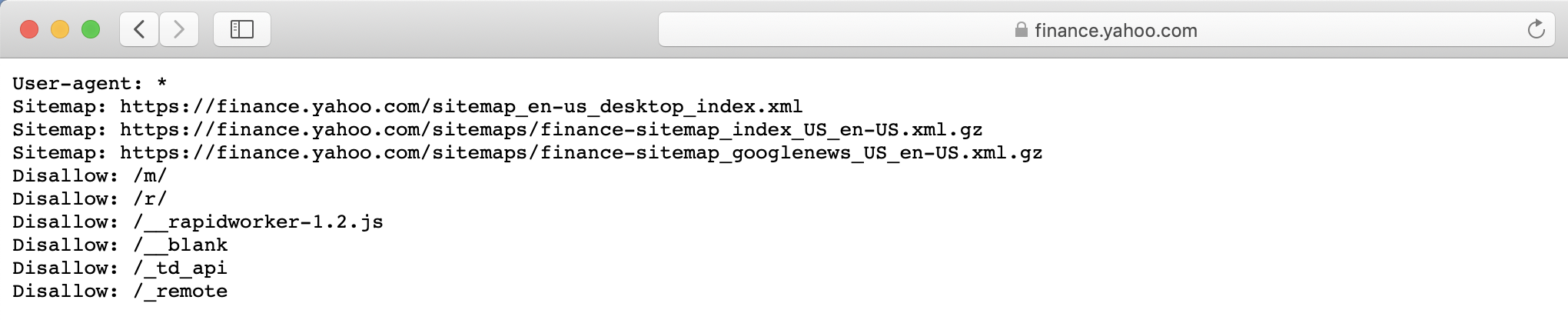


Figure 1: Robots.txt of Yahoo Finance

Yahoo Finance has very comprehensive data for all companies in US stock market. Every company has 8 webpages for multiple categories of data, and my **Web Spyder** will target these pages and download stock data accordingly. Most data are numbers (with decimals “,”, which need to be eliminated) with columns being dates and index being factor names. Some missing data are “-“ so these should be replaced by zero. Figure below is a webpage with data in Yahoo Finance (red block is the target of web Spyder).

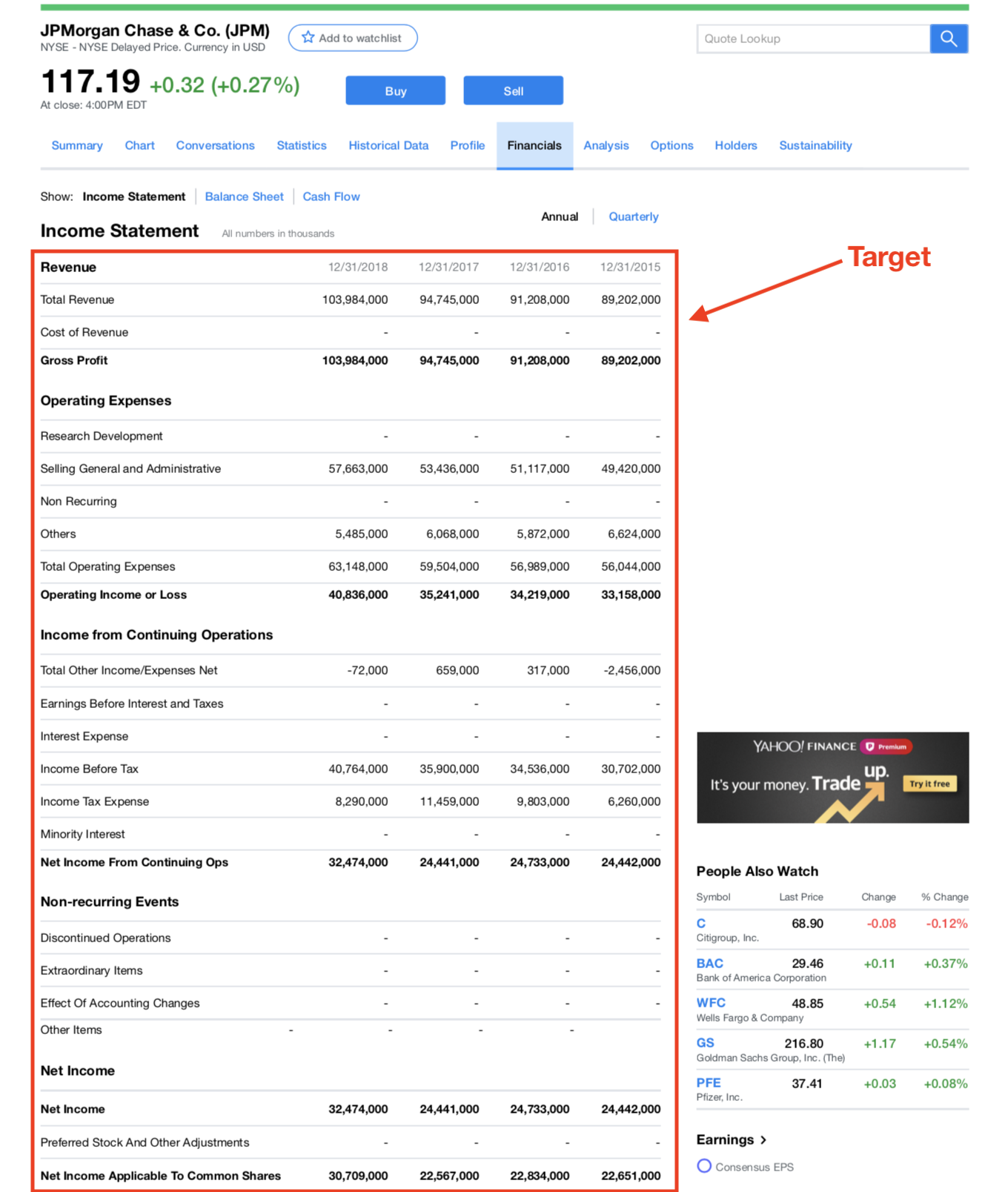
****

Figure 2: Example of a data page in Yahoo Finance (Web Spyder’s target)

**Database structure (self-designed)**

Since I decided not using external databases, I designed my own database with structure below. **Operations** have functions to interact with database, which are similar to sheet operations in MySQL. Blue and green blocks will be folders stored in Jerry’s PC, and yellow blocks will be csv sheets.

There are over 500 stocks in database, such as AMZN (Amazon), AAPL (Apple Inc.), BABA (Alibaba Group.), and JPM (JP Morgan Chase & Co.) (following the order in figure below). Each of them has a unique folder (represented by green blocks) under the JAQK Database folder. Each stock has 23 data sheets with CSV format (represented by yellow blocks), such as income (income statement), balance (balance statement), cash flow (cash flow statement), and so on. “General” in figure below is the folder that holds stock lists (retrieved from Yahoo Finance and nasdaq.com) used in measurements of famous index of US stock market, such as NYSE (New York Stock Exchange), NASDAQ (National Association of Securities Dealers Automated Quotations), and S&P100 (Standard Pour 100).

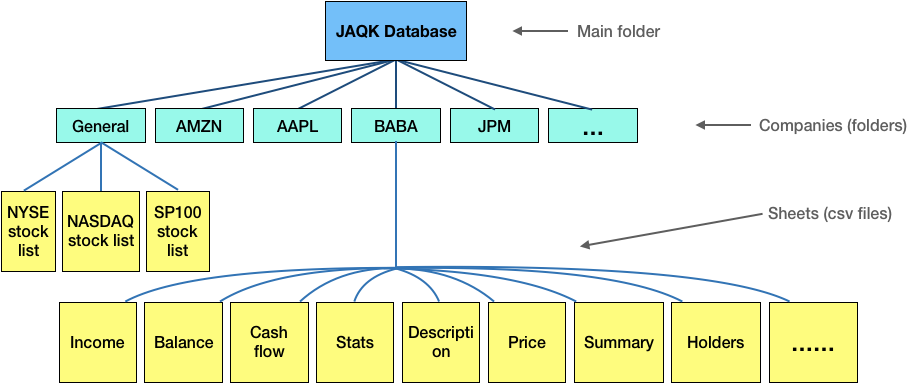


Figure 3: Structure of database

For each sheet in the database, columns will be years with format MM/DD/YYYY, index will be factors in strings (such as “Total Revenue”, “Net Income”, “Operating Cash Flow”, etc.), values will be numbers in floats. Figure below illustrates draft design of the data sheet.

**Data structure**

All data in the database are CSV sheets**.** Feature of sheet include:

* File name: all file name will be in format “{Company Name}\_{Sheet name}.csv”. For example, cash flow statement for Apple Inc. will have file name: AAPL\_cash\_flow.csv
* Columns: most index are years in strings (with format of MM/DD/YYYY).
* Index: all columns in data sheets are name of factors in strings.
* Data: all data are in numbers in floats.

Draft design of data sheet is below:



Figure 4: Draft design of the data sheet

**2. System organization**

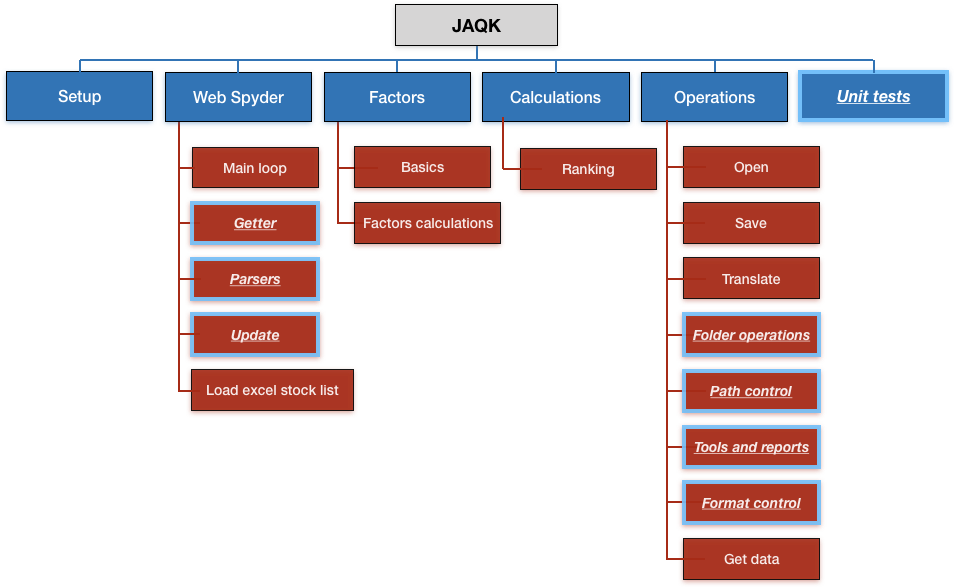


Figure 5: Structure chart of my module

To increase extensibility of codes and to meet Jerry’s expectations, I arrange functionalities with hierarchy like above. **JAQK**,the gray block, is the name of my Python module. Blue blocks, including **Setup, Web Spyder, Factors, Calculations, Operations, and Unit tests**,are wrap up of different groups of functionalities, and each satisfies one of Jerry’s requirements. Red blocks are specified functionalities under each sub module. There are other auxiliary functions below each red block, and will be explained in Criterion C. Details for each colored block will be explained below. (**Bolded words** in context means they are one of the blocks’ name described above.)

Blocks with light blue edges are internal functions to which Jerry won’t have access. Other blocks can be accessed by Jerry.

**Setup: initialize database**

Since PyPI doesn’t accept files with format .csv or .txt when compiling and uploading the package, I will convert all data sheets that are necessary to initialize database to Python script files before uploading my module to PyPI. After Jerry downloads my module from PyPI, he can call **Setup** to convert these ‘.py’ files back to CSV data sheets, which initialize database.

1. CSV sheets to .py files (which I will do before uploading):

For this conversion, I will manually change the postfix of data sheet from ‘xxx.csv’ to ‘xxx.py’. This has the effect shown below. (Data sheet below was manually downloaded from Google Finance)

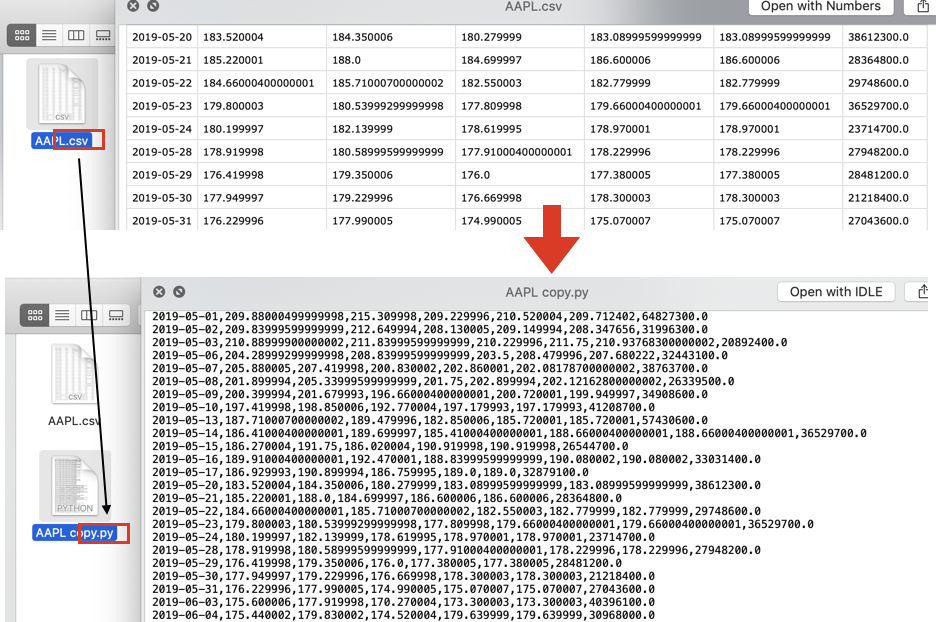


Figure 6: Illustration of converting csv to python script

2. .py files to CSV sheets (which Jerry will use):

After Jerry calls **Setup**, the program will first pop out a GUI component that asks for a path to hold the database (refer as “setup path” below), then the program will read original ‘.py’ scripts and save them to the chosen path with CSV format.

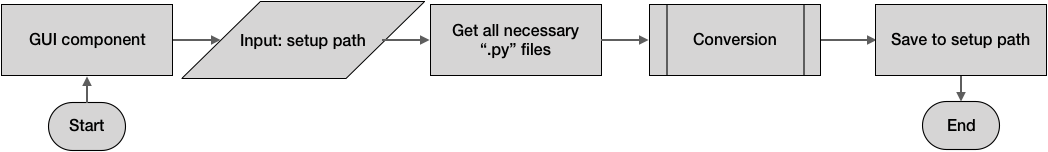


Figure 7: Designed flow chart for **Setup**

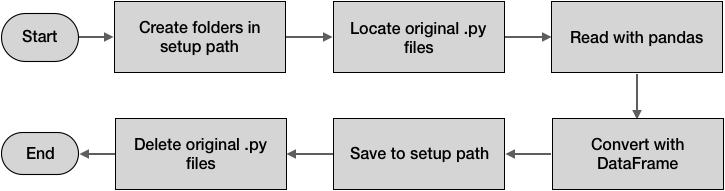


Figure 8: Designed flow chart for conversion

**Setup** also enables Jerry to freely choose a database path, to which all data will be downloaded, so Jerry can manually check data sheets within. To simplify the process of choosing path, a GUI will be used for selecting the target folder.

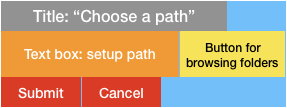


Figure 9: Layout for choosing setup path using GUI

**Web Spyder: data collection**

This module downloads data from Yahoo Finance and perform updates. Its main functionality is data collection. **Web Spyder** downloads section of data needed from the webpage and formats data into a data sheet with correct columns, index, and formatted values. Below is an illustration of the draft sketch of **Web Spyder**’s effect in downloading and formatting data.

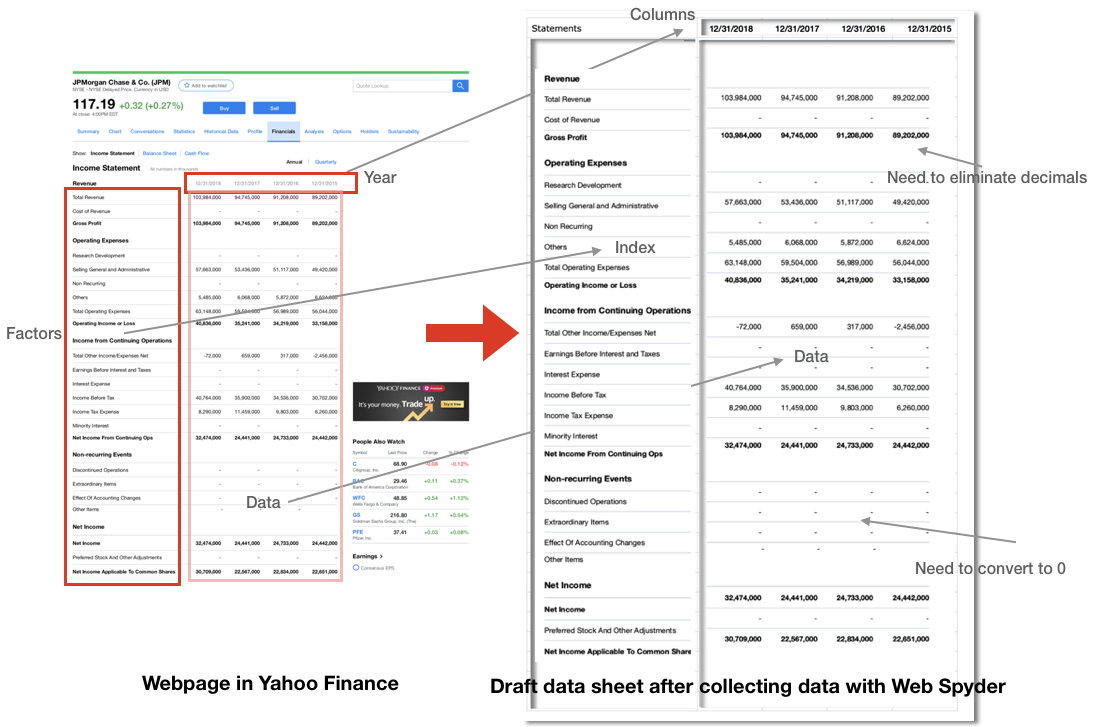


Figure 10: Illustration of the effect of **Web Spyder**

1. Main loop

The main loop is an asynchronous (non-blocking) loop which can speed up the Web Spyder. It connects getter, parsers, sheet saver. The changing parameter in the main loop is URL. The main loop will process with batches (e.g. requesting data of 64 companies simultaneously) in order to fully while not overly utilize internet bandwidth. For each company in the batch, **Getter** will send requests and pass result HTML texts to **Parsers**, which extract information wanted and format the columns and index of sheets, and the result will be saved into the folder of that company in the database.

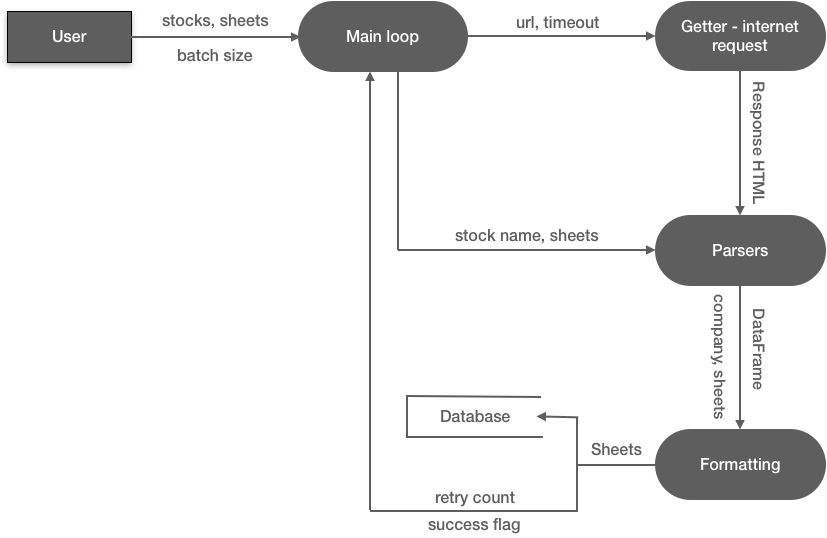


Figure 11: Designed data flow for Web Spyder main loop

2. Getter: sending internet request

This is hidden from Jerry. **Getter** requests HTML text from a URL. URL will be concatenated using strings, and some keywords will be replaced with needed ones for each request. Figure below shows how Yahoo Finance’s URL is composed. “Quote” is where category of sheets is specified, which includes “stats”, “summary”, and “financials”, etc. “JPM” is where company names are specified, for each request it will be replaced with a different company’s name, such as “AAPL”, “AMZN”, etc. “cash-flow” is where sheet name is specified, each company has 23 sheets so this will be changed frequently.

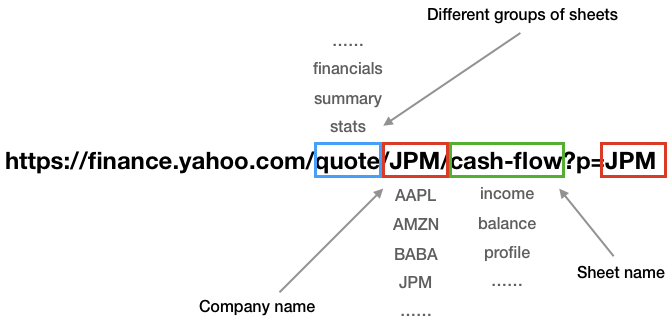


Figure 12: Example of URL concatenation of Yahoo Finance

Internet requests will be executed asynchronously to speed up the Spyder program. It can retry after possible failures (internet failure, permission denied, etc.). “Build request header” is a technique to increase the success rate of internet requests.

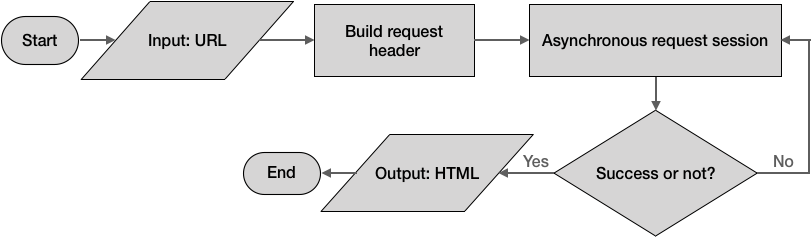


Figure 13: Designed flow chart for Getter (internet request)

3. Parsers: decoding request result

This is hidden from Jerry. Each webpage has its unique parser. Parsers decode HTML texts in order to extract wanted information, such as years, factor names, numbers, etc. Figure below illustrates the effect of parser. Pseudocode below is the general structure of parsers. Different webpage has different parsing and formatting rules.



Figure 14: Illustration of parser’s work

Pseudocode of **Parsers**:

HTML\_TEXT = GETTER(URL) // **Getter**’s internet request result

TABLES = HTML\_TEXT.CSS\_select(some\_parse\_rules) // extract information using CSS selection

RESULT = [] // list of results

loop for TABLE in TABLES

DATA = TABLE.get\_result() // retrieve data, data are usually in 1-D list

RESULT.append(DATA)

End loop

RESULT\_FORMATTED = RESULT.format(some\_formatting\_calls) // formatting

output RESULT\_FORMATTED // a 2D array that will be passed to **saver** in main loop

4. Update

**Update** is necessary because stock data are frequently changing. With this function Jerry can enjoy the newest data from stock market, which will assist his quantitative analysis. **Update** utilizes the same main loop as above for downloading data. Since Jerry’s stock list is a lot shorter than ALL US stocks (500 compared to 7000), **Update** needs to identify and update only stocks Jerry wants, rather than all 7000 US stocks.

Figure below is the data source of companies that have been updated in a given day. (<https://finance.yahoo.com/calendar/earnings>). Companies’ names in the red block are the target of **Web Spyder**, which retrieves these names, checks which of these are in Jerry’s stock list, and retrieves data of those companies that Jerry wants.

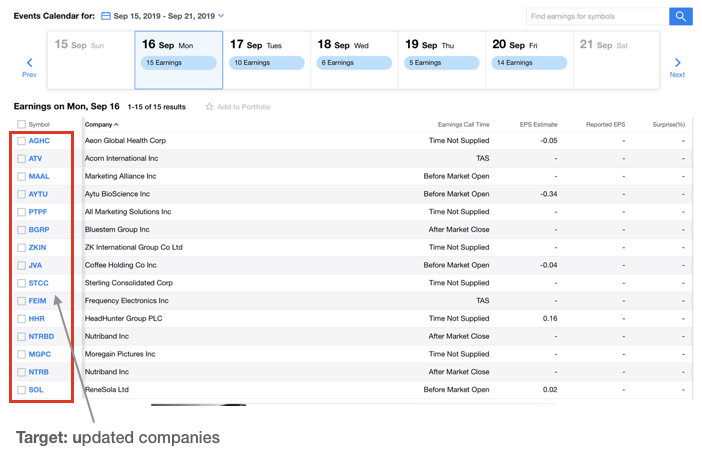


Figure 15: Example of updated company list

Figure below is data flow chart of **Update**. It checks which days does Web Spyder needs to search possible updates, which companies that have been updated during these days, and which of those are in Jerry’s stock list.

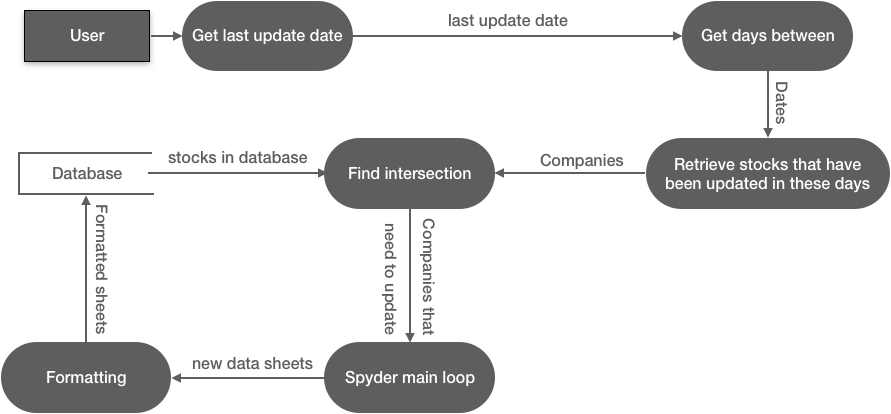


Figure 16: Designed data flow for database update

5. Load stock list **(it’s a PDF actually)**

Since Jerry gave me a list of stocks in excel (.xls) with several sheets (see Appendix B), this function reads each sheet in his excel file and concatenate sheets into one continuous CSV sheet in Python. For Jerry’s convenience, I will use a GUI for choosing the path of excel.

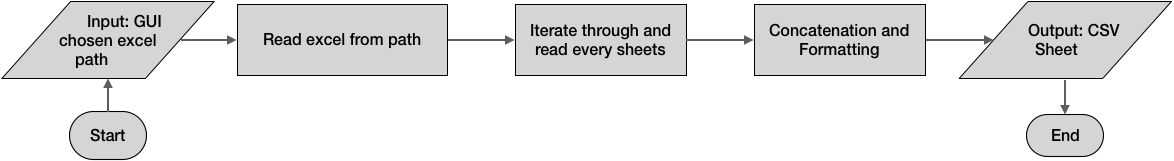


Figure 17: Designed flow chart for reading excel stock list

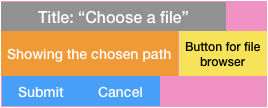


Figure 18: The layout of GUI: choosing excel path

**Factors: calculate new factors**

This module calculates important factors that are not in the stock data sheets collected from Yahoo Finance. It’s similar to **Calculation** that they both involve calculations, but **Factors** onlycalculates new factors for a single company while **Calculation** performs algorithms on a group of companies.

1. Factors calculations

This takes input of data sheets and calculate new factors based on the data sheets. Figure below shows an example of a factor: Free Cash Flow. FCF is not on any sheets that will be downloaded from Yahoo Finance, but it is one of the best fundamental indicators of a company. The calculation involves two factors that are in the cash flow statement, so **Factors Calculations** provides calculation of FCF by using these two factors.

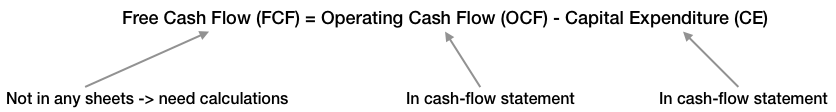


Figure 19: Example of a factor that needs **Factor Calculation**

Most calculations follow this:

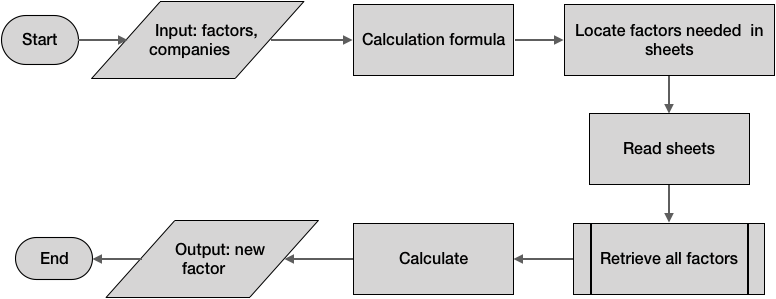


Figure 20: Designed flow chart for factors calculations

2. Basics: calculations of macro index

This takes in all factors data and calculate the macro data of US stock market. It follows similar process as above. Differences will be explained in C.

**Calculations: algorithms**

This sub module provides some important algorithms for quantitative analysis. It executes quantitative analysis algorithms on data in database. **Calculations** doesn’t include calculation of new factors (such as FCF described above); rather, it focuses on calculation over the entire database.

1. ranking: factors

Rank companies according to a factor. This include finding best and worse company, percentage position of a company, one factor’s top percentile companies (e.g. finding the top 20% companies in Net Income), and so on. Intermediate results will be saved and updated in order to speed up future calls. All calculations share the process below. Different ranking have different calculations in “Numpy calculations”. Explanation of flow chart is below.

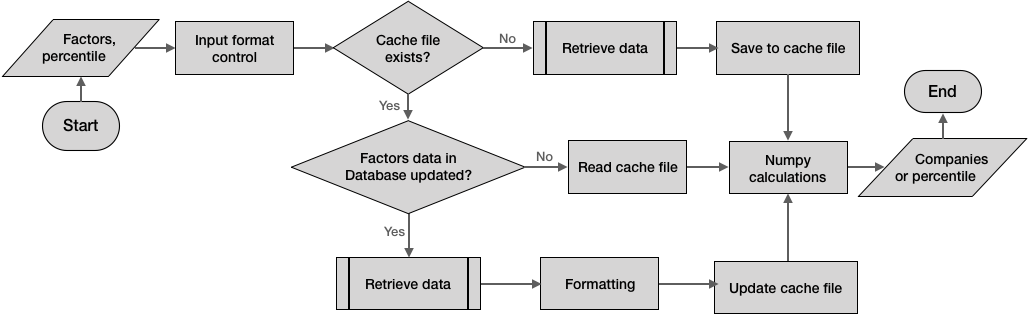


Figure 21: Designed flow chart for factor ranking calculation

“Retrieve data” traverses all companies and collects needed factors data for each company, This means it will open over 500 sheets before outputting the result, so some speed-up techniques will be used. “cache file” here refers to a saved CSV sheet containing the result of previous “retrieve data” call. Future calls of ranking will read only this “cache file” instead of going through 500 sheets in “retrieve data”. This can eliminate intensive file I/O. If database has been updated, “cache file” then need to be updated as well; to do this, the program identifies which companies have been updated and call “retrieve data” only on these companies, and the newly collected data will be concatenated to the old “cache file”.

Pseudocode of “Retrieve data” is below.

Retrieve\_data(FACTOR, need\_update\_companies)

FACTOR\_DIC = {Factor1:sheet1, Factor2:sheet2, …} // stores location of newly calculated factors

If FACTOR in FACTORS\_LIST then // FACTORS\_LIST contains all factors in all data sheets

PATH = Path\_Control[FACTOR]

Else then

FLAG = True

FROM\_WHICH\_SHEET = FACTOR\_DIC[FACTOR] // from which sheet

End if

RESULT = [] // empty list of arrays to save result

loop for Company in need\_update\_companies

if FLAG is True then // needs to call **Factors** for calculation

execute(“from **Factors** import {FROM\_WHICH\_SHEET}”) // call function in **Factors**

ARRAY = **Factors**(FACTOR) // retrieve factor

else then // FACTOR in sheets of database

SHEET = **Open**(PATH, Company) // open data sheet in database

ARRAY = **Get\_factor**(SHEET, FACTOR) // **get\_factor** function in **Operations**

end if

RESULT.append(ARRAY)

End loop

RESULT\_ARRAY = numpy.array(RESULT) // convert to 2D array

RESULT\_DATAFRAME = pandas.DataFrame(RESULT\_ARRAY) // convert to sheet

RESULT\_DATAFRAME.columns = [columns] // set columns of output result

Output RESULT\_DATAFRAME

End Retrieve\_data

“Result\_dataframe” is then passed on to be formatted (in “formatting”) before being saved to the cache file (in “Update to cache file”). With this factor data will be collected and, if factor data have been updated, result of updated factors will be updated to cache sheets. This ensures the speed and accuracy of future calls of ranking functions.

**Operations**

**Operations** acts as interactions between functionalities and the database, similar to SQL syntax in SQL databases. Operations include: Open, Save, Translate, Folder operations, Path control, Tools and reports, Format control, Get data, each dedicates to one unique interaction. All functions in my module will use only operations in this module. Other python script files will call them using relative import.

1. Open

Open data sheets in database. Below is opener for companies data sheets. Considering possible input errors and I/O errors make the opener robust. “Setup flag” indicates whether **Open** is used by **Setup**, which needs to open files in site-packages directory of the IDE rather than the chosen setup path (e.g. Jerry’s desktop or downloads).

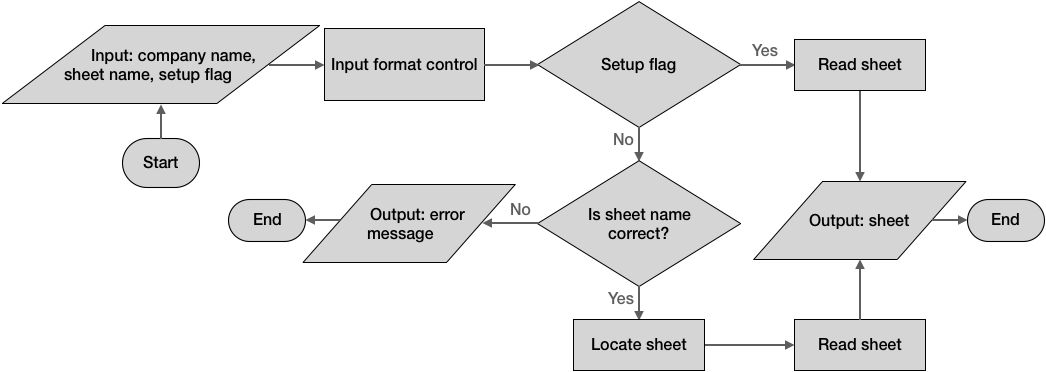


Figure 2: Designed flow chart for opening data sheets of companies

2. Save

**Save** will include saving a sheet into the company’s folder in database and into a path chosen by Jerry (which enables him to save results from calculations in his PC). Jerry can only use saver to his chosen path, and saver to the database will remain internal for **Web Spyder** and updates (of “cache files”) in **Calculations**.

A. Save to database

This is hidden from Jerry. All saving-to-database works will be done exclusively by this. It is mainly used by **Web Spyder**,which uses this saver to save result collected from Yahoo Finance. This saver can also deal with saving data in **Update**: **Update** needs to concatenate new data to old data sheets rather than write over the old ones (this ensures sufficient data for Jerry to use).

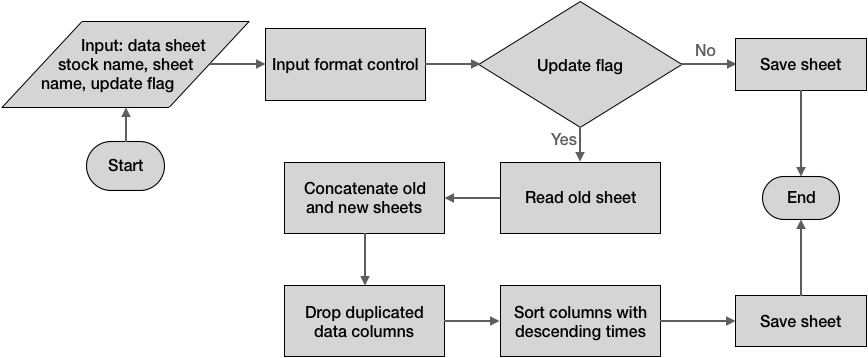


Figure 23: Designed flow chart for saving sheets to database

B. Save to user chosen path

A GUI is used for Jerry to choose a path he wants. It can save to different formats, including excel, csv, txt, and so on.

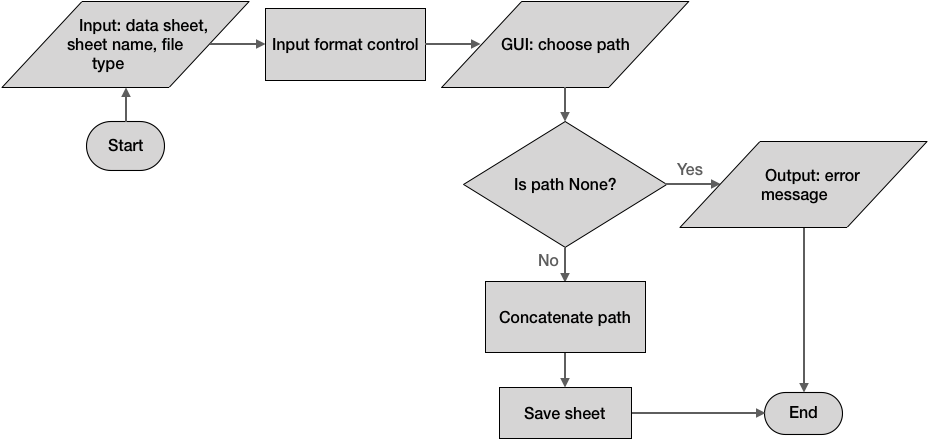


Figure 24: Designed flow chart for saving sheets to the path Jerry chosed

3. Translate

Translate description of companies and factors into any languages (Chinese for Jerry) using Baidu Translate API (http://api.fanyi.baidu.com/api/trans/product/index). This translation API is free and supports multiple language translation. I have used this API for long time in other project and the result of translation shows high accuracy. Translation will require internet connectivity.

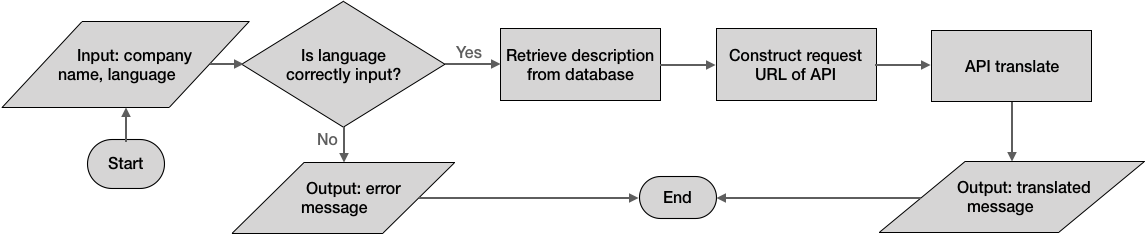


Figure 25: Designed flow chart for translating description using API

4. Folder operations

**Folder operations** contains creating folder, checking existence of files, deleting folders, and so on. This is hidden from Jerry.

A. Check existence: a sheet exists or not

Check if a file (data sheet) exists or not. If the file exists, returns True, vice versa. Before collecting data in **Web Spyder**’smain loop, this function checks if the data sheets that are going to be collected exists of not; this design reduces internet requests (those collected won’t be collected again).

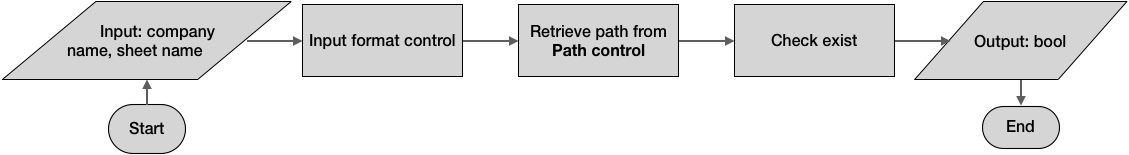


Figure 26: Designed flow chart for checking existence of data sheet

B. Create folder

Create folder before downloading data of a company. All data sheets of a company will be downloaded only to the folder designated to that company. **Setup** will change the database location (directory path), so **create folder** needs to create folder in the new database location.

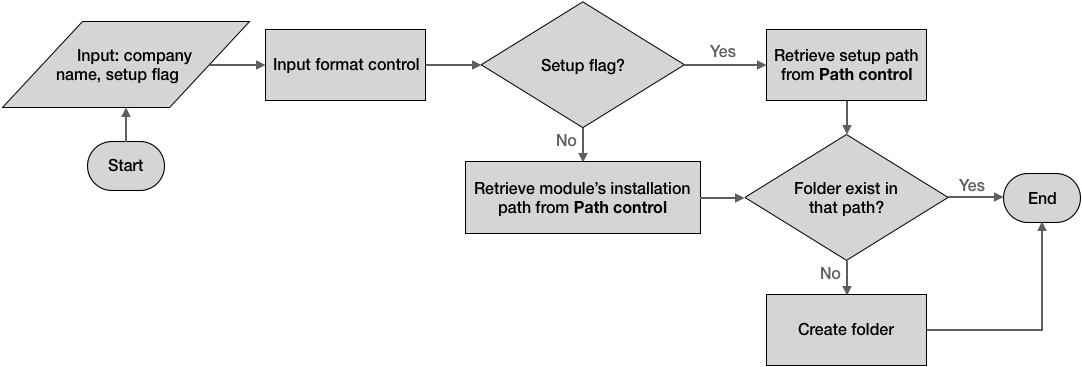


Figure 27: Designed flow chart for creating folder

5. Path control

**Path control** stores paths of database, modules, files, and factors. It acts like an index module in other database or an index page in a website. Functions in other modules call functions within to retrieve paths needed. **Path control** has a dictionary containing all paths of factors, modules, and so on. Pseudocode of **Path control** is below:

PATH\_DIC // dictionary containing all paths

TARGET = ‘Total Revenue’ // Other functions want to retrieve path of ‘Total Revenue’, a factor in database

RESULT = PATH\_DIC[TARGET]

output RESULT

6. Tools and reports

Report database status (file and company counts), clear database, and so on. Jerry can use this to know if database has sufficient data sheets or not.

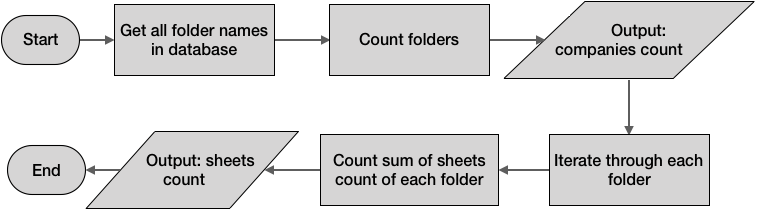


Figure 28: Designed flow chart for counting folders and sheets in database

7. Format control: clean numbers

**Format control** eliminates or converts decimals, money digits (B, M, K), and different formats of years in data sheets.

Since numbers in data sheets have decimals between numbers and have “-“ for factors that are unrecorded. These numbers can’t be evaluated in calculations since they will be regarded as strings in Python rather than floats or integers. **Format control** identifies these characters and converts to correct values: “,” to None, “-“ to 0. With this, calculations on numbers can be evaluated and become easier. I will use NumPy vectorization to map the conversion rules to entire data sheet.

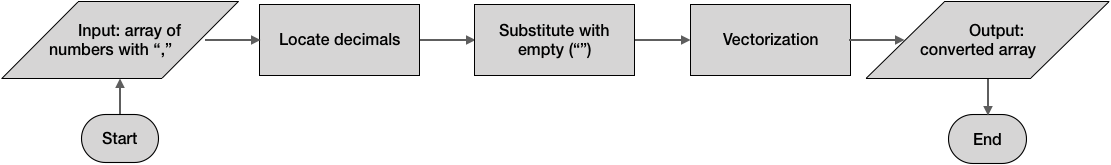


Figure 29: Designed flow chart for eliminating decimals in numbers

Similarly, money signs will cause trouble in calculations, since “34K” can’t be automatically evaluated as 34000 in calculation. **Format control** also identifies and converts these money signs to the correct values for multiplication (“34K” to 34\*1000).

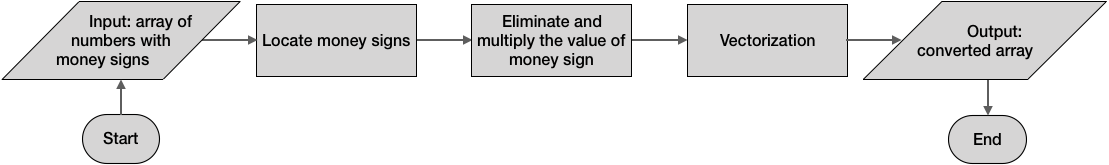


Figure 30: Designed flow chart for converting money signs

8. Get data

This is a higher level wrap up of **Open** and **Format control.** Jerry can use it to get formatted results of multiple factors of multiple companies and years. With this, Jerry and his teammates won’t need to locate, retrieve, and format data by themselves.

A. Get sheets: full data sheets

This retrieves the full data sheets from database. Note that “open\_file” already handles with locating sheet and reading sheet, but it has other functionalities since it is used by other functions such as **Get** and **Setup**; on the contrary, **Get sheets** perform a single task of opening data sheets, which makes this function more straightforward and easy to use.



Figure 31: Deigned data flow of get sheets function

B. Get factors

This can take three input parameters (companies, factors, years) and format according to different combination of inputs. However, the output can be either a 2-D sheet or 1-D array, so some input format controls are necessary (to limit that only two of three inputs can have multiple values). Columns and index names are also different with different input, which are processed in “Formatting output” below. Jerry can call this function to collect factors of one or more companies without other calling and formatting efforts.

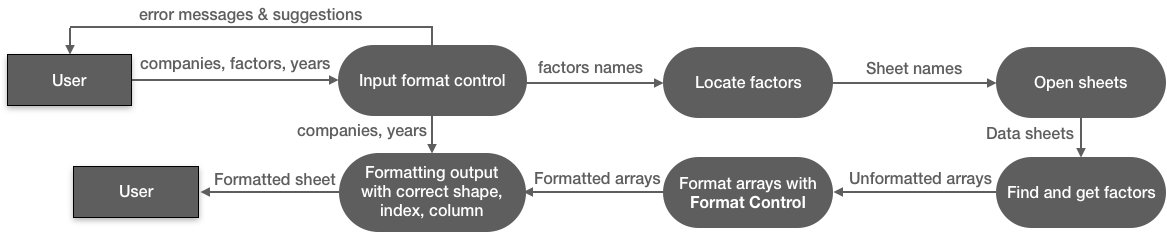


Figure 32: Deigned data flow of getting factors function

**Unit tests**

Each sub-module (blue block) above has one test class, and each function above (red block) has some test cases. **Unit tests** is a main test file that imports all test classes in sub-modules and runs them simultaneously. This ensures the integrity of functionalities after any code changes, and Jerry can use this to diagnose the functionalities in my module. Details of each test classes in sub-modules will be explained in Criterion C.

**3. Development plan**

Since my module involves three main functionalities: **Web Spyder**, **Calculations**, and **Operations**, I will develop and finish them in different stages.

Stage 1: data collection

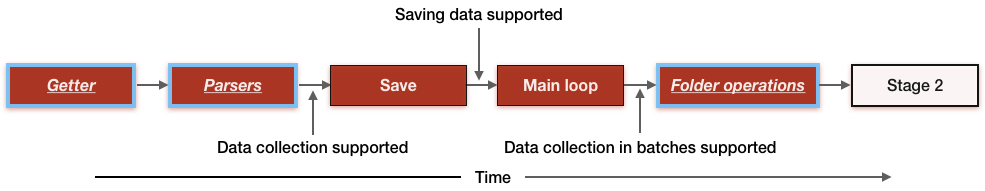


Figure 33: Plan for development stage 1

With **Getter** and **Parser**, data collection from a single webpage will be supported. **Save** enables the collected data being saved to database. With **Main loop** and **Save**, data collection is fully supported. **Operations** and **Web Spyder** will be developed in this stage. This will take two weeks.

Stage 2: update

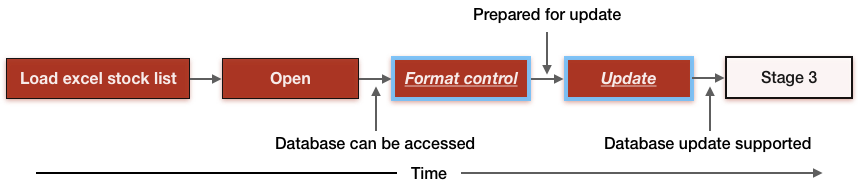


Figure 34: Plan for development stage 2

With **Open**, the program can automatically open sheets in database. **Format control** enables formatting data sheets, which will be used in **Update**. **Update** enables updating data in database. **Operations** and **Web Spyder** will be developed in this stage. This will take one week.

Stage 3: setup database

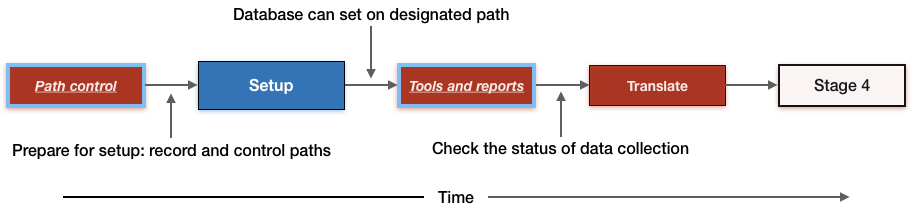


Figure 35: Plan for development stage 3

**Path control** helps to manage paths of database, functions, and sub modules, which is necessary in **Setup**. With **Setup**, Jerry can choose a path for the database. **Tools and reports** check the status of database. **Translate** will be developed in this stage. **Web Spyder, Operations,** and **Setup** will be developed in this stage. This will take one week.

Stage 4: calculations

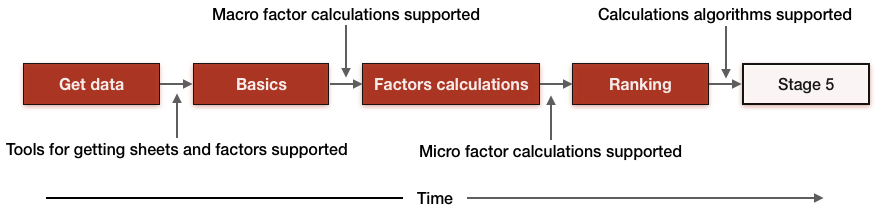


Figure 36: Plan for development stage 4

**Get data** provides higher level wrap up of Open developed above; it enables Jerry to get formatted results of factors and sheets. **Basics** give macro (market level) factor calculations, and **Factors calculations** give micro (company level) factor calculations. **Ranking** enables calculations in my product. Factors, Calculations, This will take two weeks.

Stage 5: Unit testing

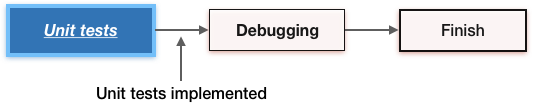


Figure 37: Plan for development stage 5

**Unit tests** will be developed last, because it needs to test all functionalities. It will make sure all success criteria and other functionalities are unaffected after future code changes. This will take one week.

**4. Upload to PyPI**

Uploading to PyPI (Python Package Index) needs some extra work. According to PyPI’s official documentation (https://packaging.python.org/tutorials/packaging-projects/), I will need to write a “setup.py” script, a readme.md file, and a license file. (Hierarchy shown below.) After this, I will need to execute some commands in terminal (command line in Mac), which will perform upload. I will follow the example code in PyPI official documentation after developing all functionalities described above. Below is a draft illustration of the hierarchy of the package:

JAQK (uploaded package)/

jaqk (my module)/

\_\_init\_\_.py

Web Spyder/

……

Calculations/

……

……

setup.py

LICENSE

README.md

Figure 32: Illustration of package (for uploading) hierarchy

**Testing plan**

All test plan in success criterions will be implemented in **Unit tests** module. Test classes within each module will include tests below. **（全部打包一次性搞定，自动化test）**Test classes have other tests to test utility functions but not within success criterions, so they are not listed below.

|  |  |  |
| --- | --- | --- |
| **Installation and setup** | To be tested | **Installation using “pip install” is supported and no other steps are needed.** |
| Test plan | Upload module to PyPI then execute “pip install JAQK” in terminal/cmd, then import JAQK in Python IDE and call **Unit tests** module. |
| Anticipated result | Upload is successful. Executing “pip install JAQK” automatically and successfully downloads the module and any dependencies. Import JAQK in IDE is successful, and **Unit tests** module reports success in running all important functions. |
| To be tested | **Setup can convert setup data and initialize database.** |
| Test plan | Call setup() after import JAQK, choose a setup path in the GUI. Manually go to the chosen path, see if needed folders are existing. Open each folder in the chosen path, randomly open some data sheets and compare them with original ones, see if all data are correct. Go to my module in Python IDE’s site-packages directory, see if all original data sheets with “.py” have been deleted. |
| Anticipated result | GUI successfully pops out, selecting path is successful. All data sheets are converted correctly and moved to the chosen path with correct database organization. Original sheets have been deleted. |
| **Web Spyder** | To be tested | **An Update function to update data of companies in database, but this should expand on original data rather deleting original ones.** |
| Test plan | Manually identify the date when database was last updated, then go to Yahoo Finance and manually find companies that need to be updated by **Update**. Go to database and randomly choose three companies, make a copy of those companies’ folders. Call update() and wait for result. Check if last updated date and companies that need to be updated are correctly identified. Go to database, find those sampled companies, then compare all data sheets of those companies; check if new data appear starting from the left of the sheet (descending order of time), and check if old data are not rewritten. |
| Anticipated result | **Update** correctly identifies last updated dates, identifies dates that need to be searched, and collects companies that need to be updated within these days. Checking database shows successful concatenation of new data to the old ones without hurting old data. |
| To be tested | **A Get function that can automatically collect ALL the factors and ALL financial data of companies provided by Jerry from Yahoo Finance.** |
| Test plan | (Finish **Setup** first.) Manually identify few stocks that have very comprehensive record of data (JPM, AMZN, etc.) and those that have limited data recorded (Shanghai Airport etc.), pass them in as parameters of **Get** (with this **Get** only collects these companies). After collection, go to setup chosen path and manually go through all data sheets within, compare them with data in Yahoo Finance webpage. Inspect if columns, index, and data are correct in all these data sheets. Call database status report to see the company and sheet counts match the manually counted numbers. |
| Anticipated result | **Get** automatically collects and saves data sheets into setup chosen path. Data sheets are saved to correct folder and are complete. Columns, index, and data are all correct. Database status report counts correctly when comparing to manual counting. |
| To be tested | **Progress of functions above should be reported.** |
| Test plan | Test along with two tests above. When calling **Update** and **Setup**, wait for result and manually watch if the progress is reported for both functions. |
| Anticipated result | Progress is reported with estimated remaining time and current progress. |
| **Calculations** | To be tested | **Some basic calculations should be available: factors, ranking, etc.** |
| Test plan | (After downloading all data in the database in setup path.) Make a copy of the database, delete 90% of data in the copy (50 companies remaining), call **ranking** on these remaining companies in the copy, wait for result. When result returns, manually go to database and inspect these companies to see if the result match the data in database. Use different input parameters (companies, factors, percentage, etc.) and repeat steps above. Delete the copy after the test. |
| Anticipated result | **Ranking** works successfully, and the result is correct. |
| To be tested | **Calculation don’t take longer than 1 second.** |
| Test plan | Activate timer in code, then call **ranking**, wait for speed result. Record the time needed, then repeat steps above for 5 times, record accordingly and manually calculate the average time. |
| Anticipated result | Speed timer successfully calculates time. Average time is less than 1 second for giving out results. |
| **Operations** | To be tested | **Open can handle possible errors and open sheets in database.** |
| Test plan | Manually open some data sheets in the database. Pass their company names as parameters in “**open\_file**” call. Manually compare the result from **Open** and the sheet in database. Try different input parameters and repeat steps above.  Then give **Open** different but incorrect parameters, including incorrect company names, sheet name, and format of those two, etc. Read the error message and compare them with the ones that were designed to pop out. |
| Anticipated result | **Open** gives correct sheet with correct format when calling with correct parameters. With incorrect parameters, **Open** can detect them and give correct error messages. |
| To be tested | **Save can save data sheets to database as well as to a chosen path.** |
| Test plan | Manually open a data sheet using **Open**, then choose a folder and path it in as parameters of **save** call. After execution, go to the folder and see if the data sheet has been correctly (index, columns, data, sheet name) saved. Try different parameters (company name, sheet name, etc.) and repeat steps above.  Then, for manual check: call **Saver** (which saves to a chosen path), choose a correct path and input a correct file format (CSV etc.); for unit testing, automatically pass in the path (mimicking choosing path) and then proceed as above. After saving, go to the folder and check if data sheet is correctly saved. Repeat the step above with incorrect path and file format (.dmp, .zip, etc.), see if warning massage pops out. |
| Anticipated result | **Save** successfully saves the data sheet into correct folder for all different parameters. **Saver** that saves to a chosen path successfully saves the data sheet into the GUI chosen path. |
| To be tested | **Translate can translate description of companies into Chinese.** |
| Test plan | Get descriptions from three companies (randomly) using **Open**, and pass them into **translate.** Set parameter “language” as Chinese (zh), Japanese (jpn), and Arabic (arb). Record the translated result, manually go to Google Translate to translate the translated texts back to English, and manually read and compare result with original description.  Then, go through steps above with poor and no internet connection, see if error message pops out. |
| Anticipated result | English text that’s translated back was similar (in meanings) as original descriptions. Chinese translation is correct (since I speak Chinese). When internet is poor, **Translate** successfully and correctly detects it and gives error messages. |
| To be tested | **Folder operations can create and check existence of sheets.** |
| Test plan | Open database in setup path. Call **create** with input of a folder name, then go to database and see if new folder has been created in correct directory. Delete that folder after test.  Call **check** with input of a sheet that exists in database, record result. Then call with a sheet that doesn’t exist in database, record result. Repeat steps above with different sheet names. Then, input incorrect parameters (incorrect formats) and see if error message appears. |
| Anticipated result | **Create** successfully create a folder in the database with correct name, and delete that folder after this test. **Check existence** successfully returns Boolean values on sheets, and it successfully detects the incorrect input parameters and gives correct error message. |
| To be tested | **Path control can locate factors in database and can output correct path of each module, sub module, and function.** |
| Test plan | Manually select some factors in data sheets and record. Pass them in to **Path control** and compare the result (which will be a sheet name) to the record. Try with different factors in data sheets and repeat steps above. Try with incorrect input, such as non-existing factors and factors with wrong format (e.g. a list instead of a string), see if error has been detected and reported.  Manually record paths of some functions and sub modules, then call **Path control** to retrieve location of these recorded functions and sub modules. Compare the result from **Path control**. Try with incorrect input, such as non-existing functions, see if error has been detected and reported. |
| Anticipated result | **Path control** gives correct location for all factors tested. For incorrect input parameters, errors are successfully detected and alerted.  **Path control** gives correct paths for functions and sub modules tested. For incorrect input parameters, errors are successfully detected and alerted. |
| To be tested | **Tools and reports can report database status (file and company counts).** |
| Test plan | Create a copy of database. Manually, count the number of folders using commond+I (for Mac, the command will give info of the database folder). Call database count, compare the result given and the manually counted.  Leave on company and delete all others in the copy. Count the number of sheets under the remaining company, then copy the company folder for 50 times. Call database count, compare the result given and the multiplication result of 50 and the number of sheets.  Delete the copy after this test. |
| Anticipated result | Database status gives correct counts of data sheets and folders in the database. Deletion is successful. |
| To be tested | **Format control can clean non-number characters in numbers (comma, money sign, etc.) and format output of Get data.** |
| Test plan | Call **Format control** functions and input arrays with decimals (like 23,000) and money sign (B, M, K). Compare results given and anticipated result (23,000 to 23000, 34K to 34000, etc.). Try more than 5 times with different unclean values.  Call **Get data** with correct and different combinations of input parameters (companies, factor names, years, etc.), see if the results are formatted (column and index; data will be check in **get data** test case). |
| Anticipated result | All arrays are correctly formatted (e.g. 23,000 to 23000, 34K to 34000). All calls of **Get data** gives correct format needed. |
| To be tested | **Get data provides functions to retrieve data in database and give formatted results. This include getting data sheets, factors, and so on.** |
| Test plan | Call **get sheets** with correct input parameters (company, sheet). Compare the result with the data sheet in the database, and see if the result have correct data, columns, index, and so on.  Call **get factors**, try different combinations of years, company names, and factors names, see if either the result or the error message is successfully and correctly outputted with correct formats. Try with incorrect combinations (incorrect non-included company names, formats, years, factors, etc.) for the call and examine if error has been detected and reported. |
| Anticipated result | **Get** **sheets** and **get** **factors** give correct result with correct data, correct format, and correct index and columns for all combinations of inputs. Errors are successfully and correctly detected and reported in concise languages. |