Mini Project Report

### 

### PREPARED BY

Wang Liang Yi

* Contributed to Program Testing, Source Code Analysis and Report

Jethro Phuah An Ping

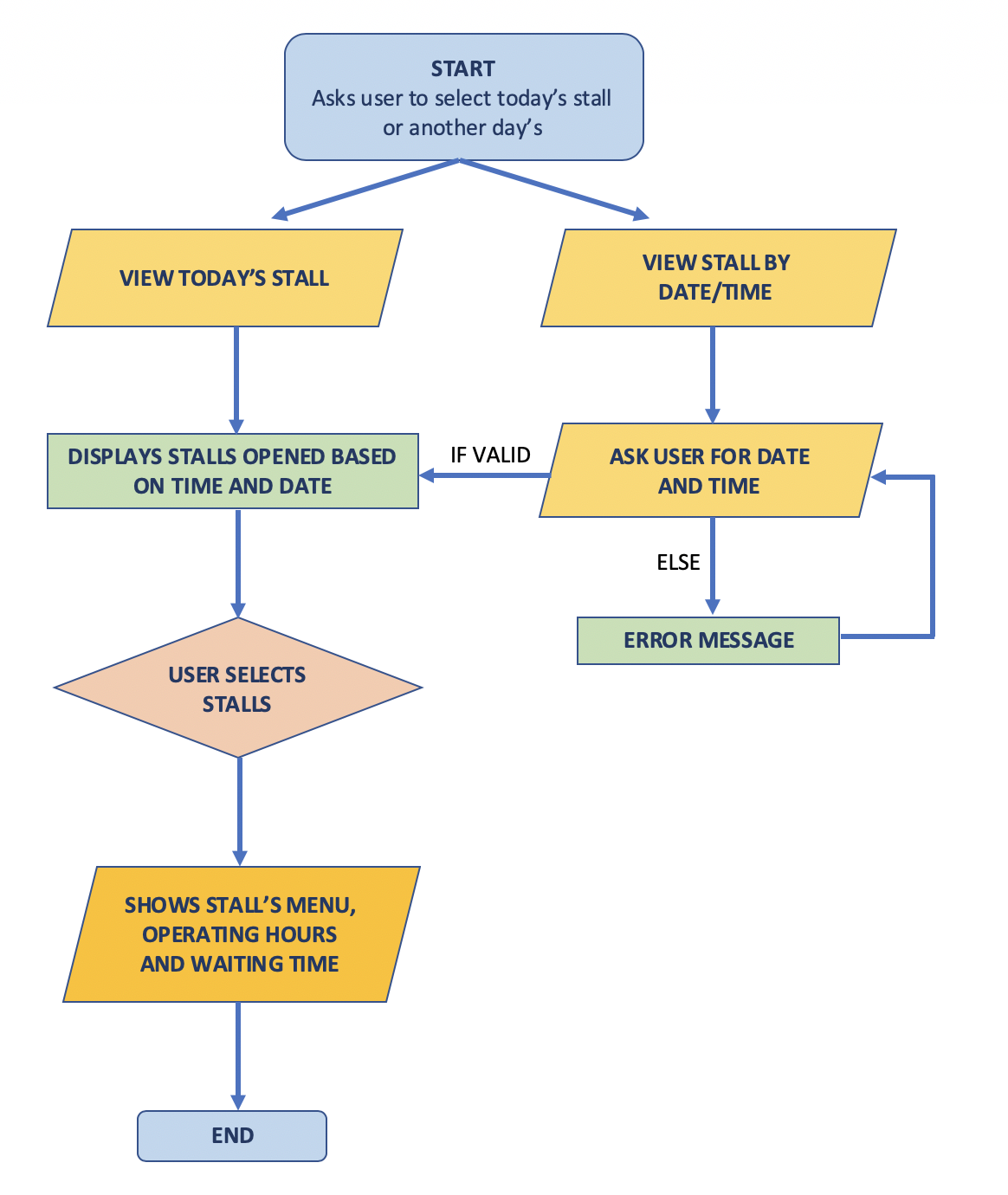
* Contributed to Backend Data Storage & other module, Source Codes, Window Designs and Report

Brandon Chen Yun Xin

* Contributed to Frontend Design, Source Codes and Report

Algorithm Design

# 1. Algorithm Design: Top level flow chart



*Figure 1*

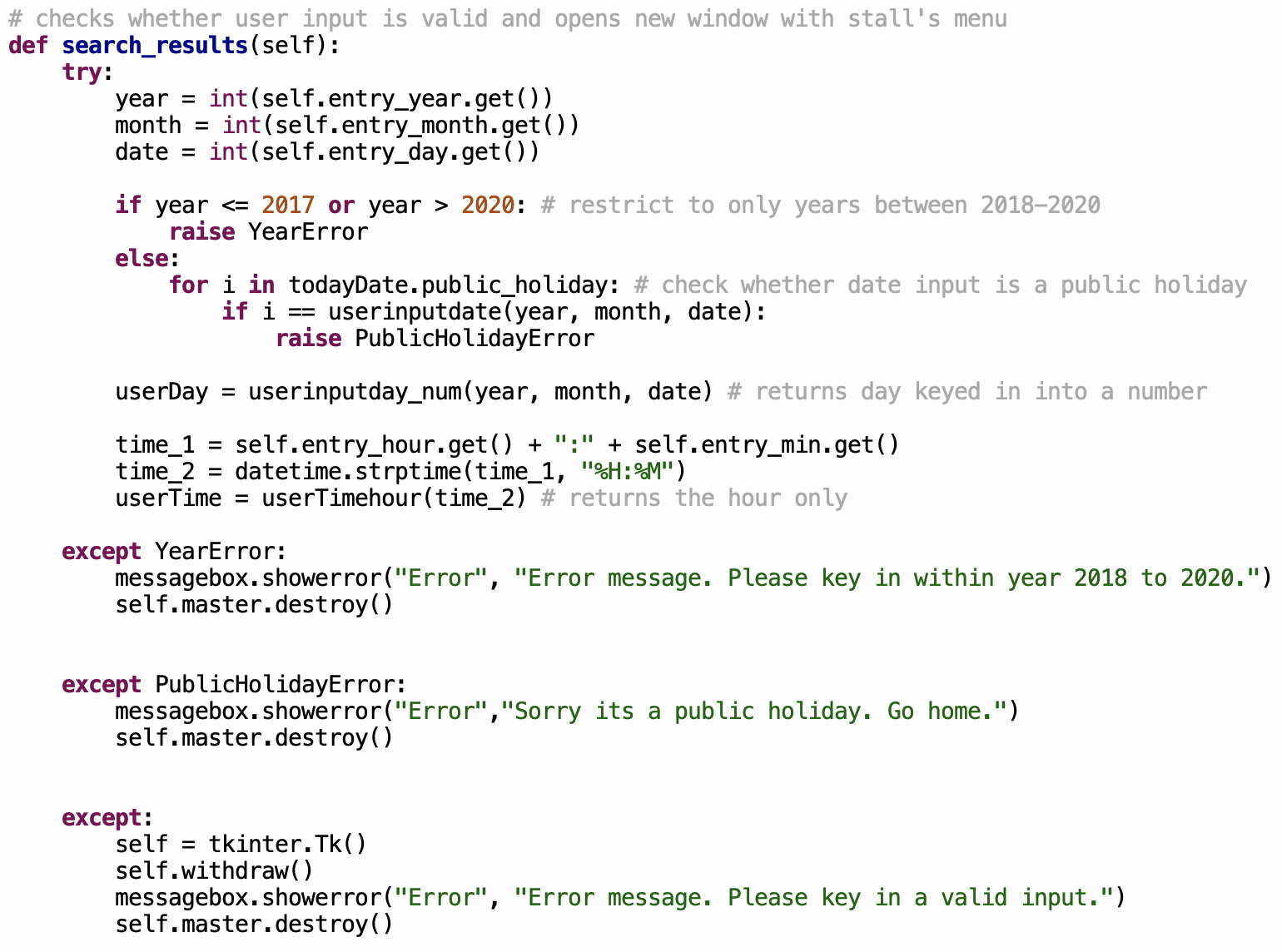
# 

# 

# 2. Algorithm Design: Brief description of important functions

2.1 Get user date and time (***search\_results()***)

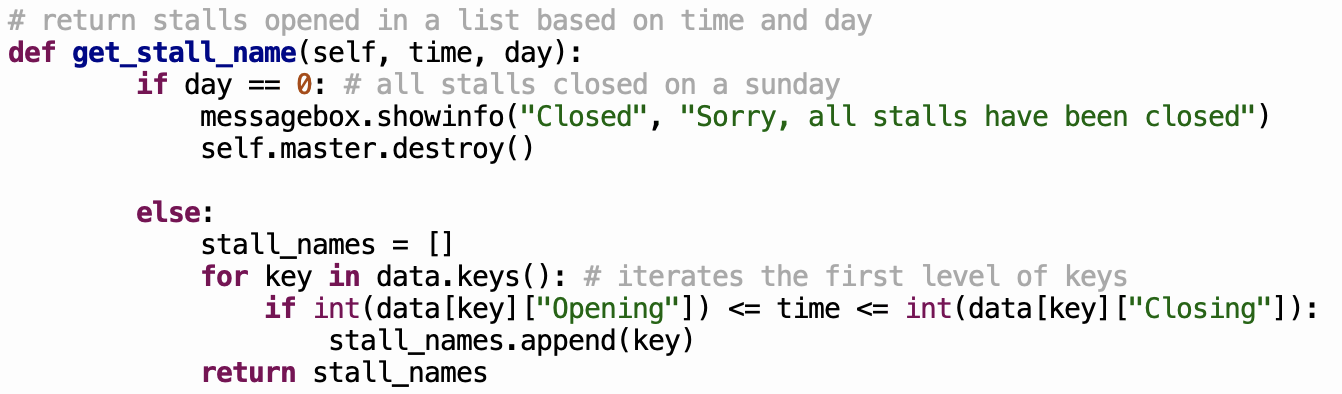
Input box gets user’s input, which is then checked using a try-except block and returns an error message if it’s invalid as seen in *Figure 2*.



*Figure 2*

2.2 Check which stalls are open (***get\_stall\_name()***)

After getting date and time from system or user, function checks for stores that are open and returns a list. Function contains an if-else loop (*Figure 3*) whereby the parameters, time and day, will be passed through. If day == 0, it means that it is a Sunday and therefore, an error message box will appear. If day is any other numbers from 1 to 6, a list of stall names will be generated using a for loop to search for stalls that are opened within the given time frame.



*Figure 3*

2.3 Check which menus are available (***get\_menu()***)

After getting day and time from system or user, program checks for special menus that are available and returns a dictionary of food items as seen in *Figure 4.*

*Figure 4*

2.4 Calculate waiting time (***calculate\_waitingTime()***)

Calculated based on *number of people in the queue* x average *time taken per person*.

# 

*Figure 5*

# 3. Algorithm Design: System’s function requirements

|  |  |  |
| --- | --- | --- |
| **System’s functions requirements** | | **Explanation** |
| A. | Store and display stall information | *NorthSpineCompiled.json* file stores all the following information in a simple nested dictionary:  a) Stall name  b) Operating hours  c) Menu  d) Price of food  Example:    Taking the above section as an example, we have:  a) “Indian Food” stored as a string to be the key to another dictionary.  b) “Opening” and “Closing” is the key to the operating hours stored as integers.  c) Menu” is the key to another dictionary which stores the food items as a string.  d) “Nasi Briyani” is the key to a list with the following format [*price, opening hour, closing hour, day*] |
| B. | Store and display stall menus |
| C. | Display stall information and menus based on current system date and time | After selecting the store, the menu is presented to the user based on the date and time given by the system or user. |
| D. | Display stall information and menus based on user defined date and time |
| E. | Calculate estimated waiting time for the stall by asking user to enter the number of people in the queue | An option to calculate waiting time is available at the bottom of the page. |
| F. | Allow to check the operating hours for all stalls | It’s displayed after the stall is chosen, on the menu page. |

# 4. Algorithm Design: Non-complexity

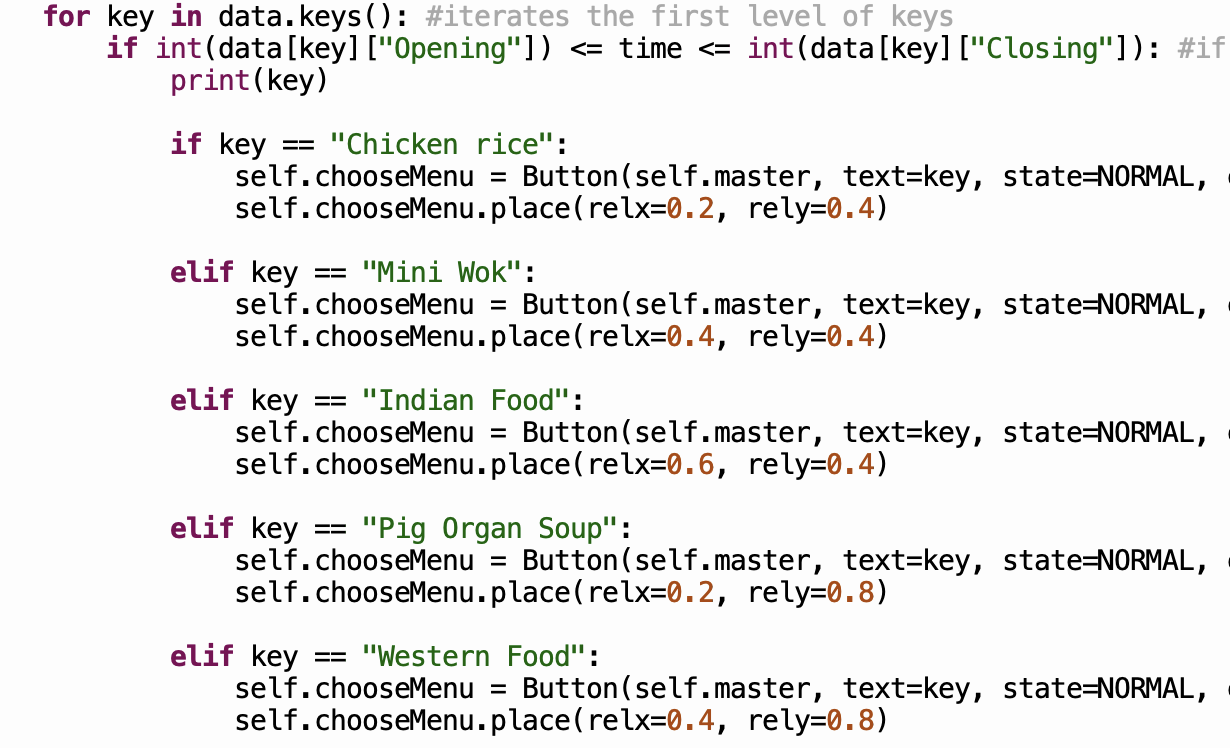
Our functions support a single purpose each. Taking our function ***get\_stall\_names*** as an example in Figure 6, we gave it the simple purpose of retrieving the stall names from our json file and returning the stall names of the stalls opened in a list based on the time and day that the user wants.



*Figure 6*

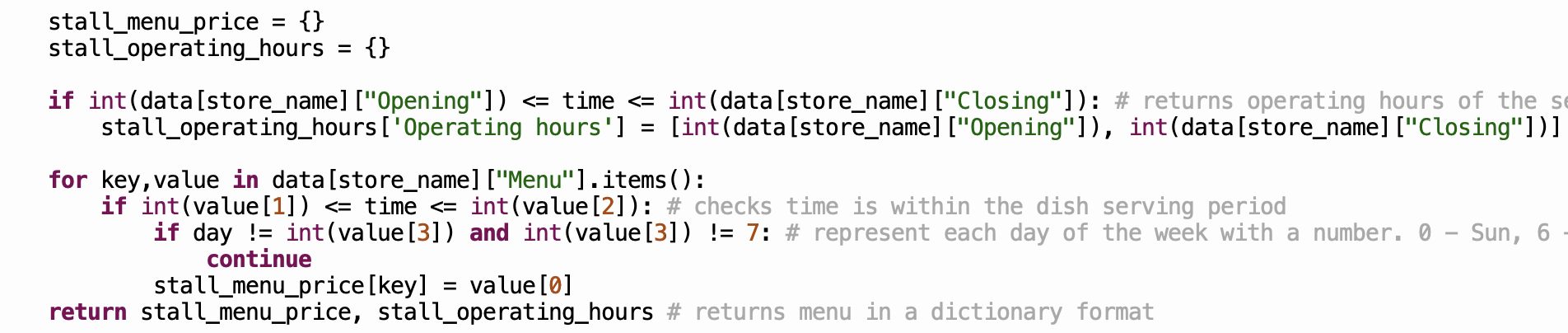
We also made our program more scalable. For example, in retrieving the menu for a certain stall, we switched from if-else statements (*Figure 7*) which had a time complexity of O(n) to indexing directly into a dictionary (*Figure 8*) which is O(1).

Before:



*Figure 7*

After:



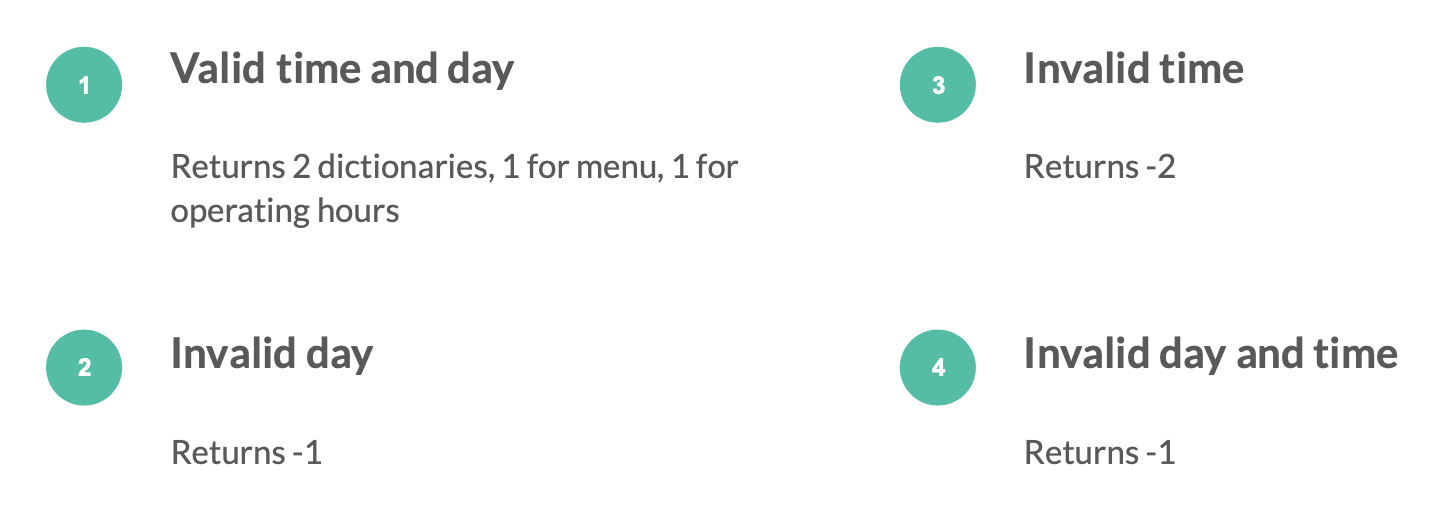
*Figure 8*

Program Testing

# 1. Methodology

We used mainly unit testing and this was automated in our Test Cases.py file. Focus was on testing the 3 important functionalities (mentioned above): getting the right user input for day and time in our required format; displaying the respective stores that are open; showing the correct menu and special menus if any. This can be seen from Figures 9-12 where we also stated the respective output. Figure 13 & 14 is a screenshot of our Test Cases.py file.

# 2. Test user input for date and time

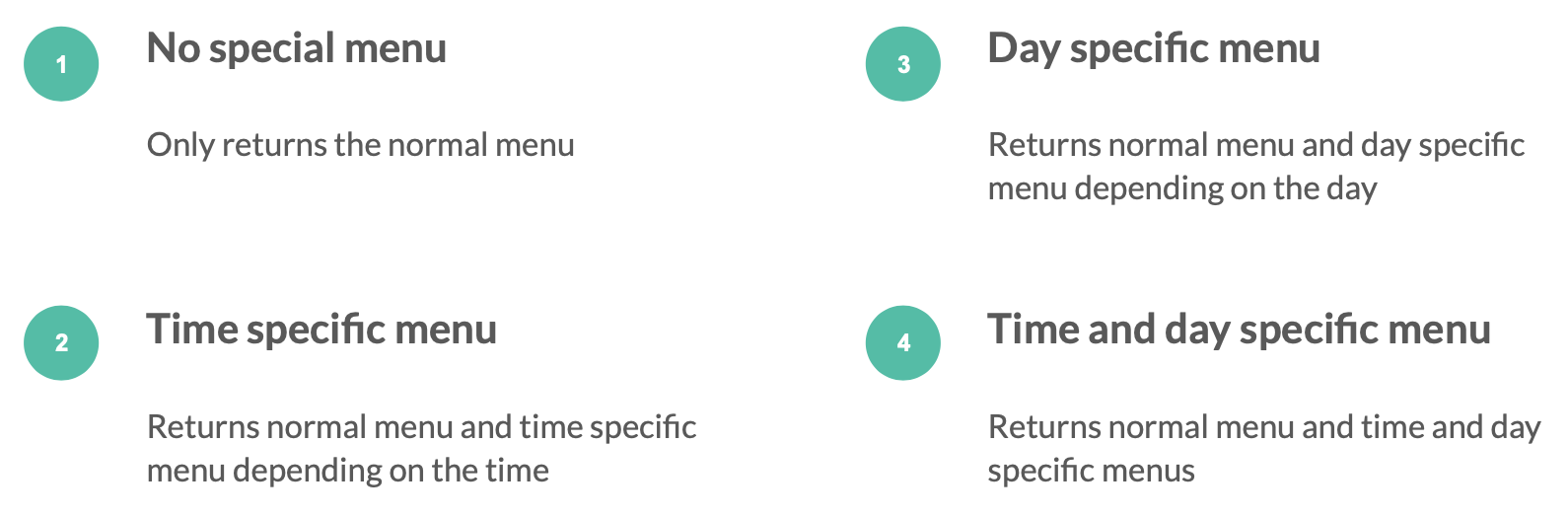


*Figure 9*

# 3. Test for opening stores

*Figure 10*

# 4. Test for correct menu



*Figure 11*

# 5. Test for valid number of people in queue

*Figure 12*

# 6. Unit-testing

# 

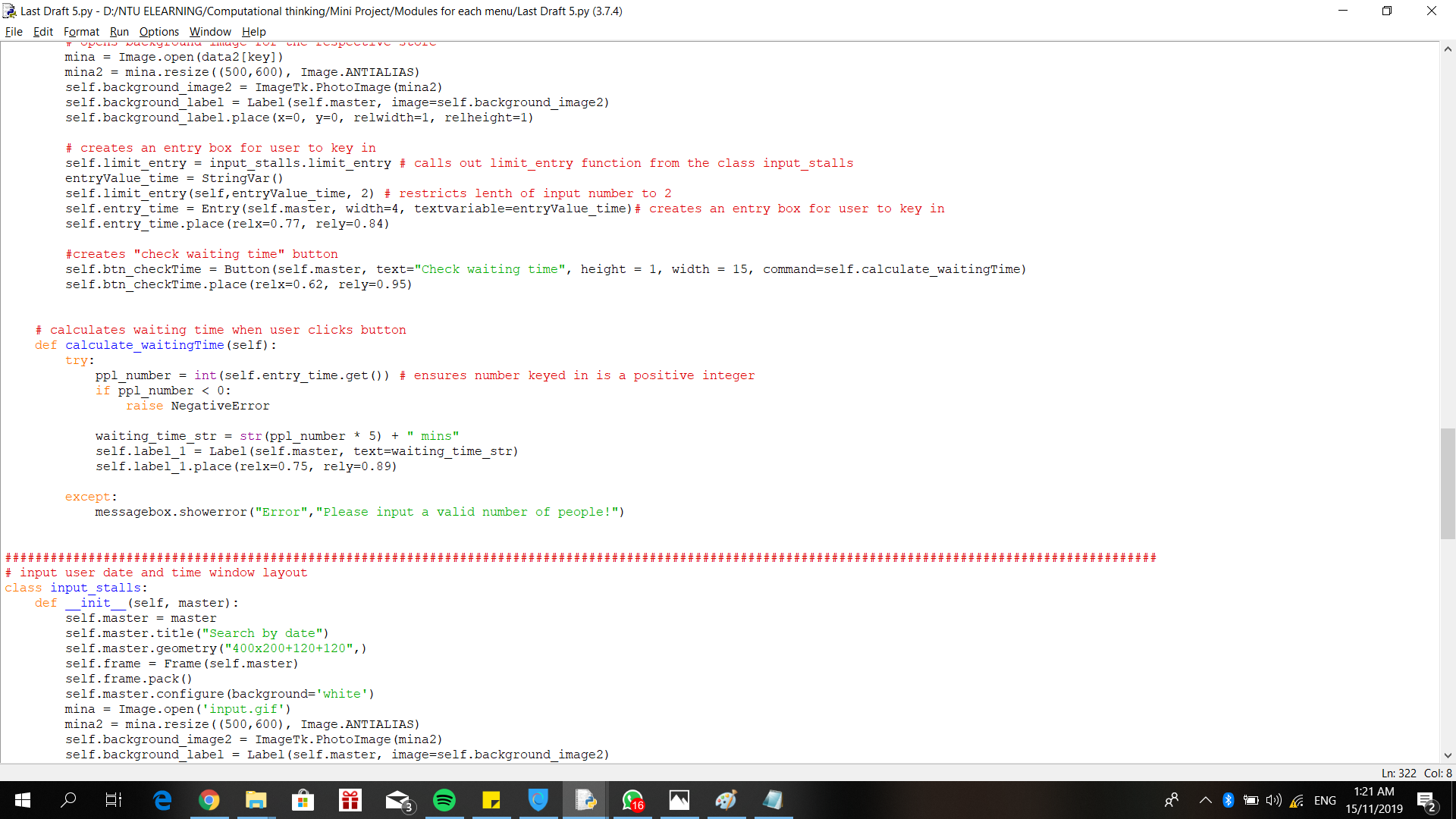
*Figure 13*

# 

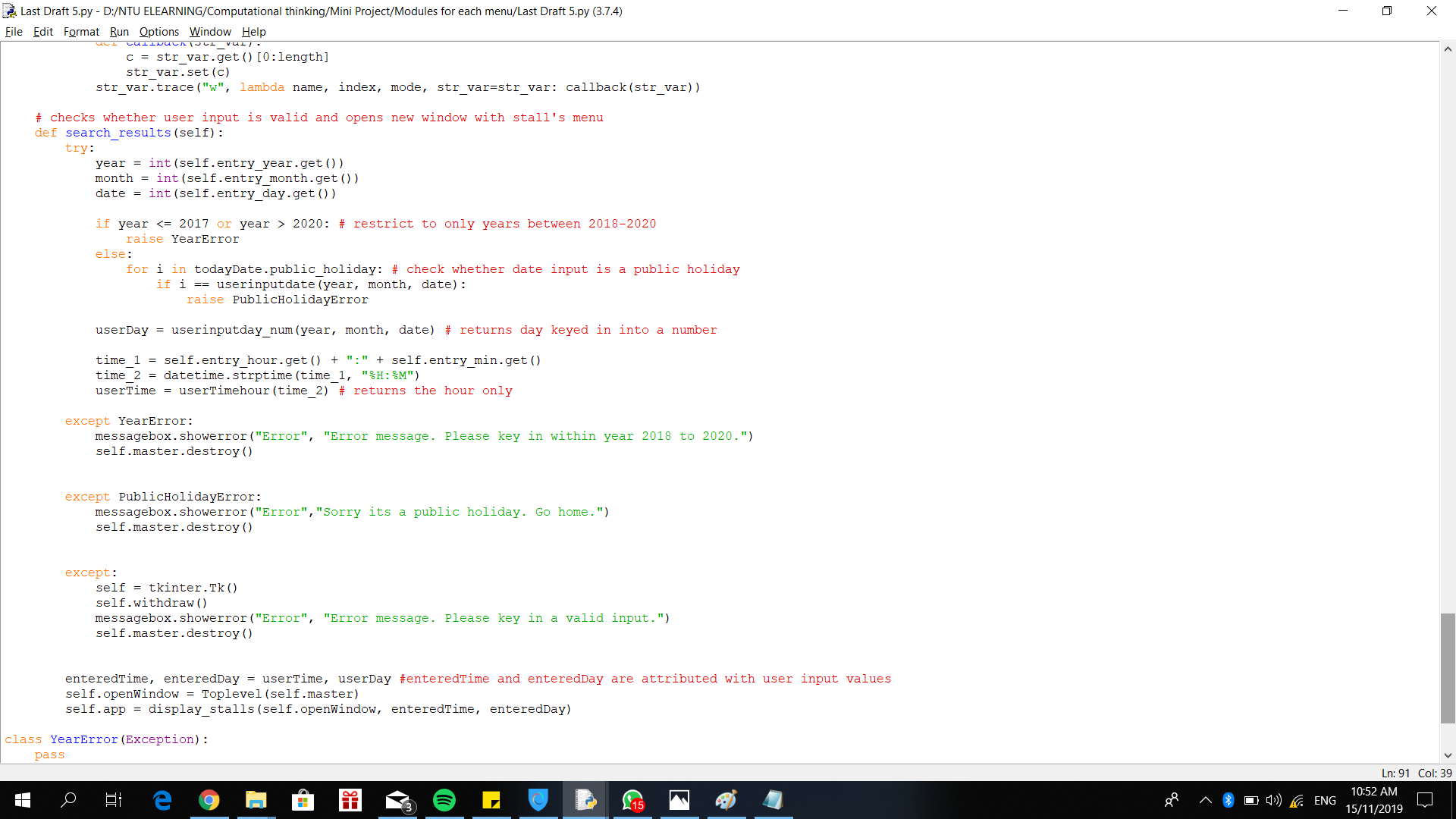
*Figure 14*

# 7. Try and Except

As can be seen from Figure 15 & 16, we also used try-except (by raising a custom Exception class for each of these restrictions) and if-else statements to check for valid user inputs; restricts user to only key in dates from year 2018 to 2020 and informs the user if canteen is closed due to a public holiday.



*Figure 15*



*Figure 16*

Reflections

# Difficulty 1: Duplicate code

In the beginning, we created functions that were specific to each stall. For example, we created one function to check if chicken rice stall was open and another function to check if Mini Wok was open etc. We also had one function to view stalls based on system’s current time and another function to view stalls based on user’s selected time. This made the code lengthy and inefficient.

After reviewing the codes and their functions, we realised our mistakes and refactored the functions such that it was stall-independent and takes the stall’s name as an input instead. Through this, we learnt to avoid code duplication by making functions more generic, having optional arguments to make it more flexible and calling them when we need them.

# Difficulty 2: High dependency on another team member’s work

As the codes for the front-end interface and back-end data retrieval was dependent on each other, our codes needed to be highly aligned. This meant a time delay when we needed to create functions due to We had to wait and coordinate schedules a lot as multiple people would work on the same functionalities and their progress would be limited by another person’s progress, resulting in less efficient work

We then learnt to start with a high level understanding of the program first before dividing up the work according to front-end, back-end etc. After which, we spotted dependencies and agree on each function’s inputs and outputs before implementing it.

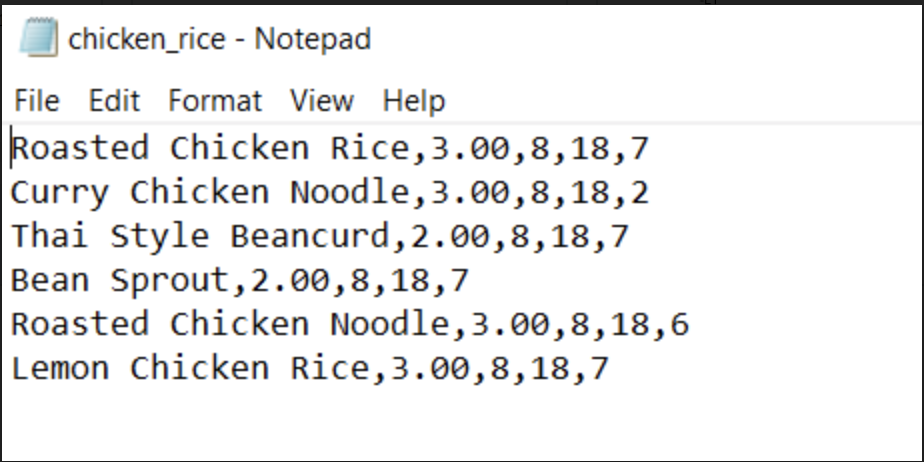
# Difficulty 3: Code readability

In the initial stages, our codes were disorganized and difficult to understand because we created new functions whenever the need arose without making comments. This meant that we had to understand the functions again every time we used it. Team members also had to understand the algorithm from scratch, taking up a lot of time.

After reviewing the source codes, we had more detailed documentation and abstracted implementation details so that those relying on our functions need only know its inputs and outputs.

# Difficulty 4: File Handling

We found file handling to be challenging initially because we stored each stall’s information in individual csv files as seen in Figure 17 and could not structure the code to make data retrieval efficient.



*Figure 17*

After reviewing the files, we learnt to store all information into a nested dictionary in a single json file (Figure 18) which made retrieving information from the database more efficient.

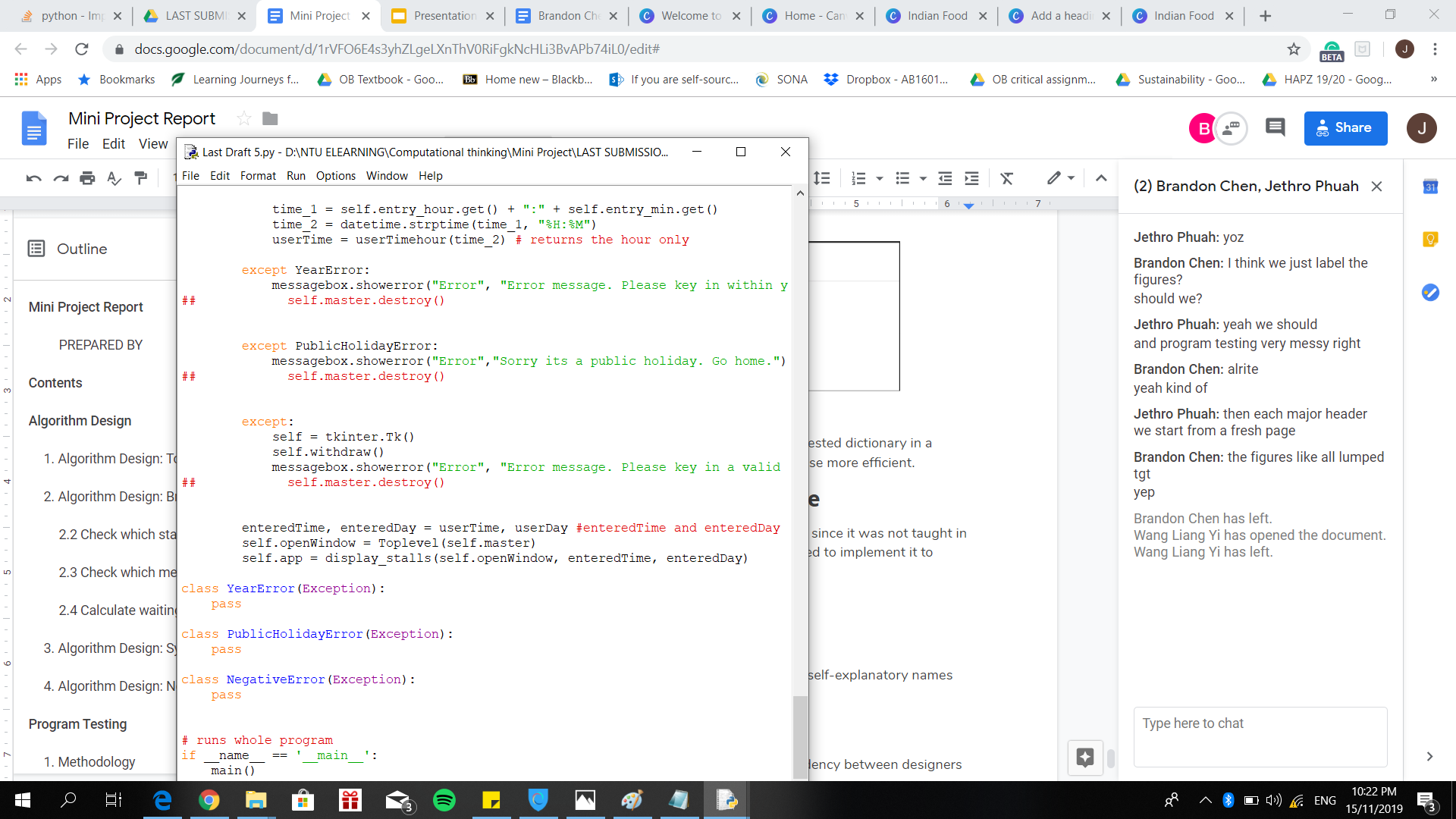
# 

*Figure 18*

# 

# Difficulty 5: Exploring unfamiliar code

We faced the challenge of using a class to simplify our algorithms since it was not taught in our syllabus yet. However, after much online research, we managed to implement it to simplify our program. Figure 18 shows a simple example of our usage of class.



*Figure 19*

# Other good practices

To improve code readability and understanding, we implemented self-explanatory names for functions and variables.

# Further improvements

1. Create a separate file for design patterns to reduce dependency between designers and front-end engineers.
2. Create a mobile app for users’ increased convenience.
3. Learn GitHub for version control and better coordination.