MINI PROJECT

CZ1003: Introduction to Computational Thinking

Wang Wayne
Zachary Varella Lee

Real-time Canteen Information System
SESSION 2019/2020

SCHOOL OF COMPUTER SCIENCE and ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

Contents

1. Introduction

1.1 Purpose

This report documents the computational thinking process during the development of our Real-Time NTU North Spine Canteen Information System.

1.2 Contribution

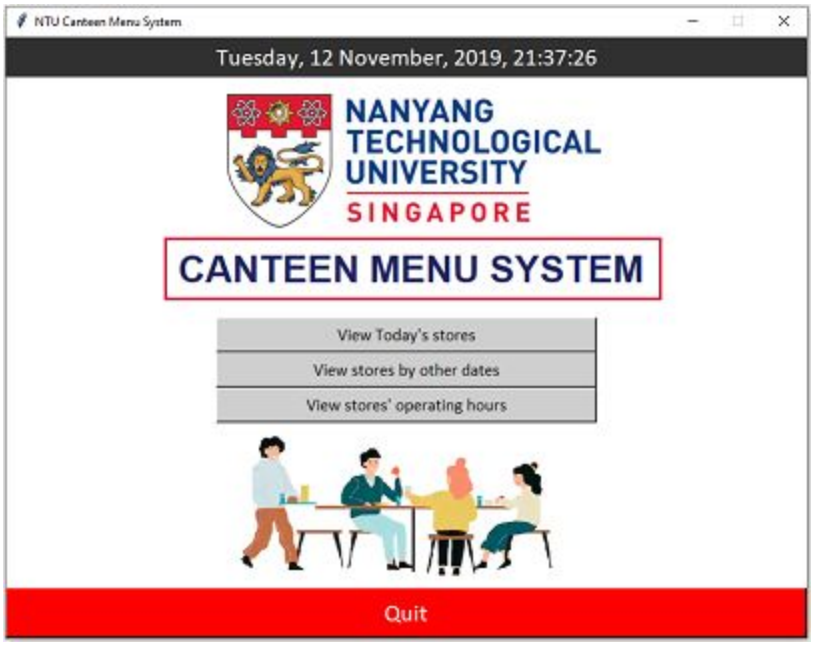
The work was split into front-end programming (Wang Wayne) and back-end programming (Zachary Varella Lee).

1.3 Scope

This report looks into the data processing, branching, looping, file handling, algorithm design, program testing and the difficulties encountered.

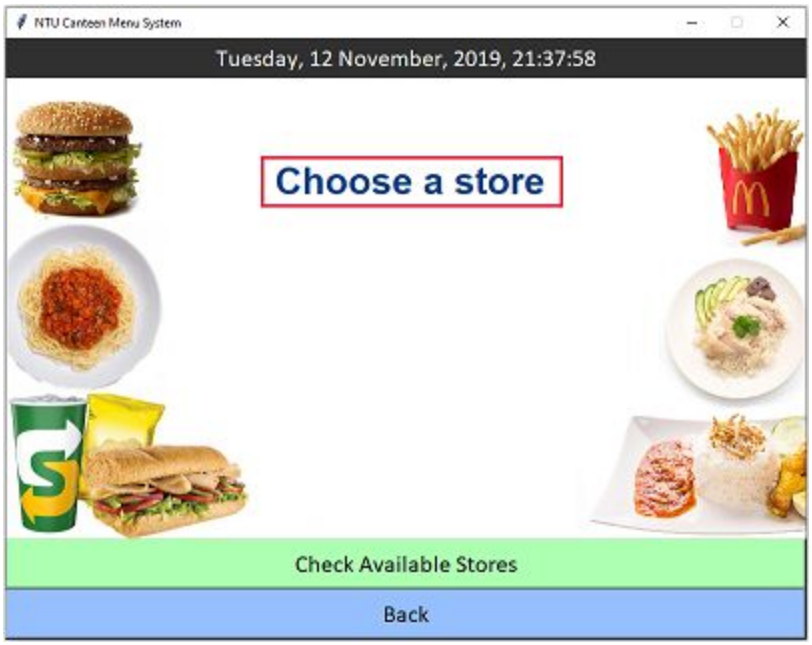
2. Overview of the Program

2.1 Start Page

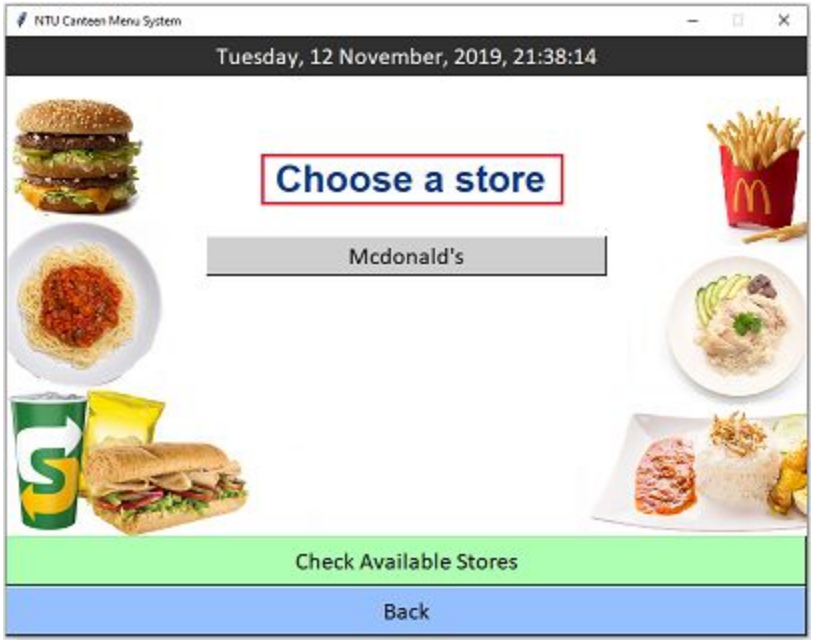


The clock at the top of the application updates in real-time. The user can interact with 4 buttons which each leads to a new frame.

2.2 Choose A Store (Feature A, C)

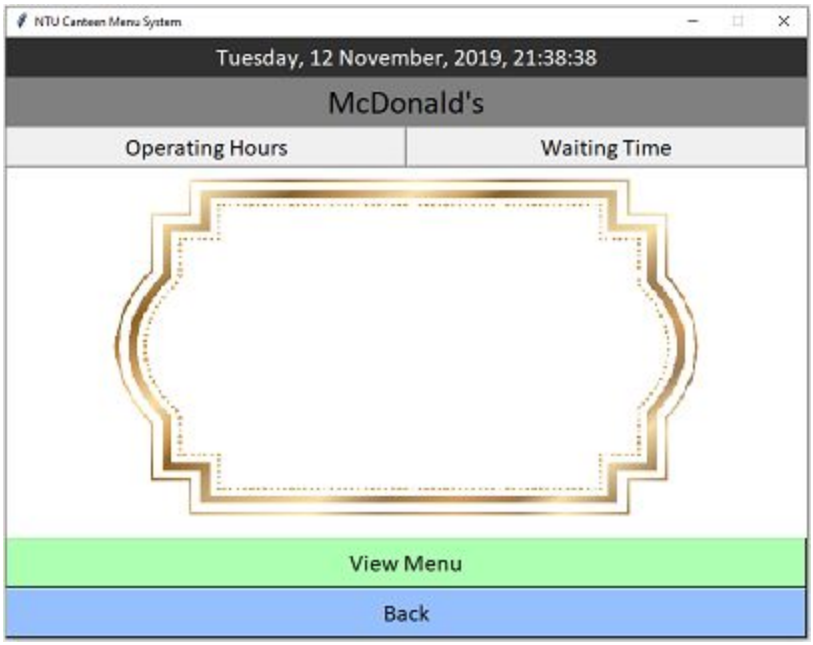


In this frame, the “Check Available Stores” button displays the available stores based on the current date and time, shown below.



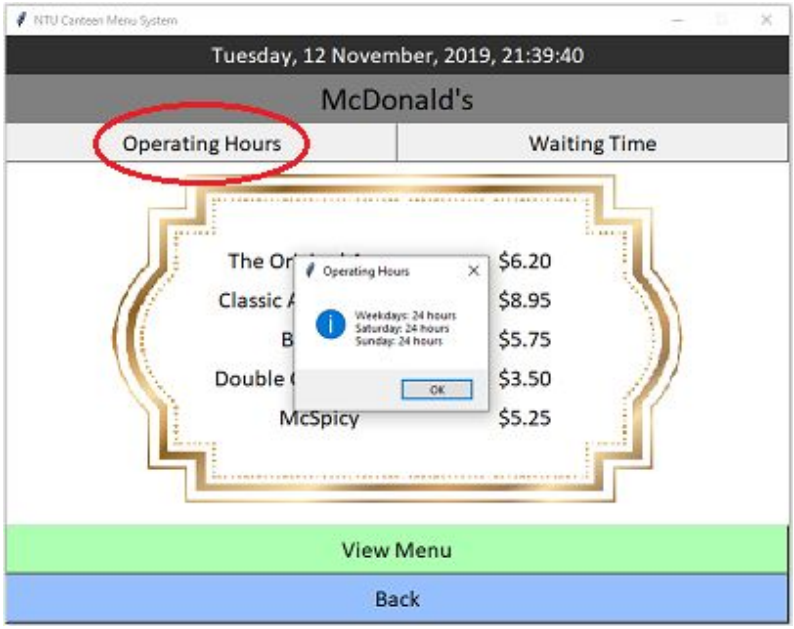
The user can choose any of the available stores to view the menu. In this case, only Mcdonald’s is available at the time.

2.3 Store Frame (Feature B)

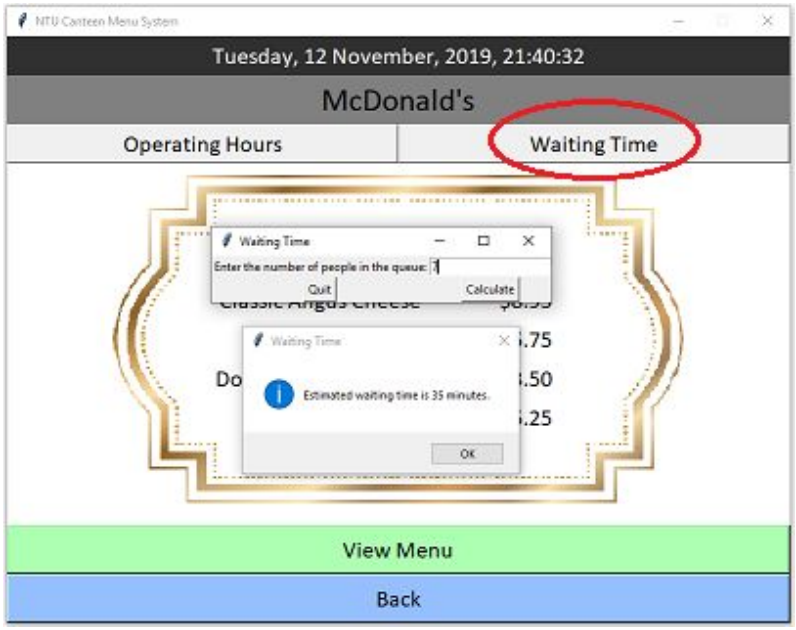


In this frame, there are a few functions. The main function is the “View Menu” Button which displays the menu for the selected store.

2.4 Operating Hours & Waiting Time (Feature E, F)

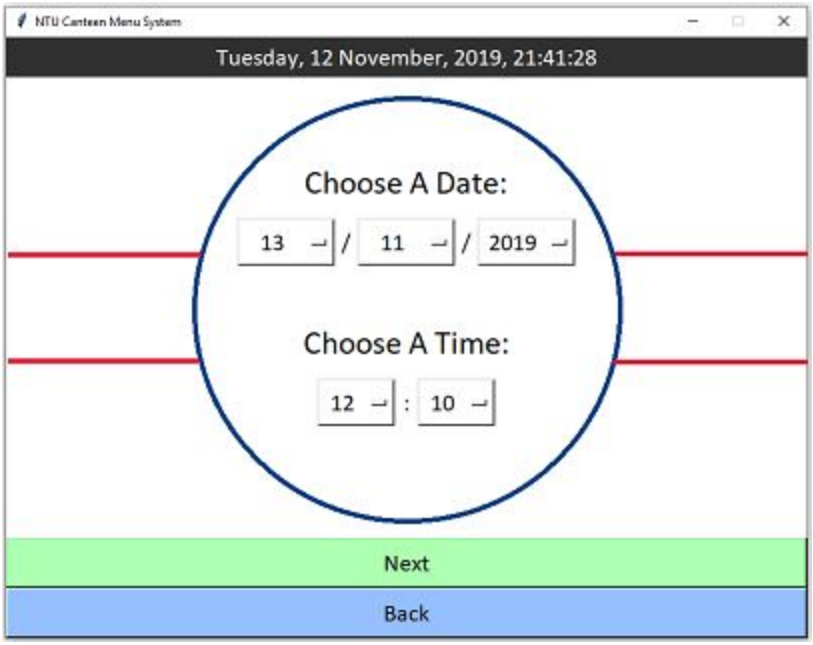


Within the same frame, the user can view the operating hours of the selected store via a popup window.

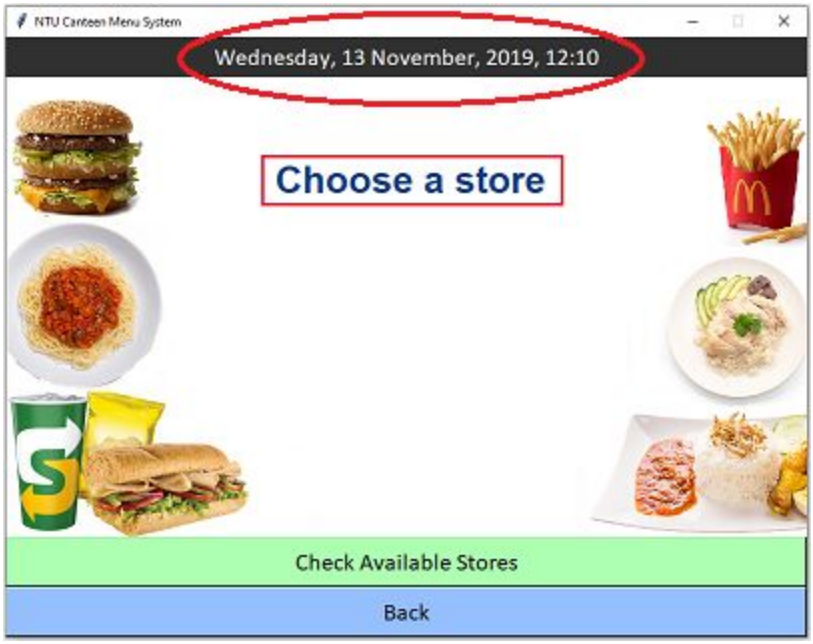


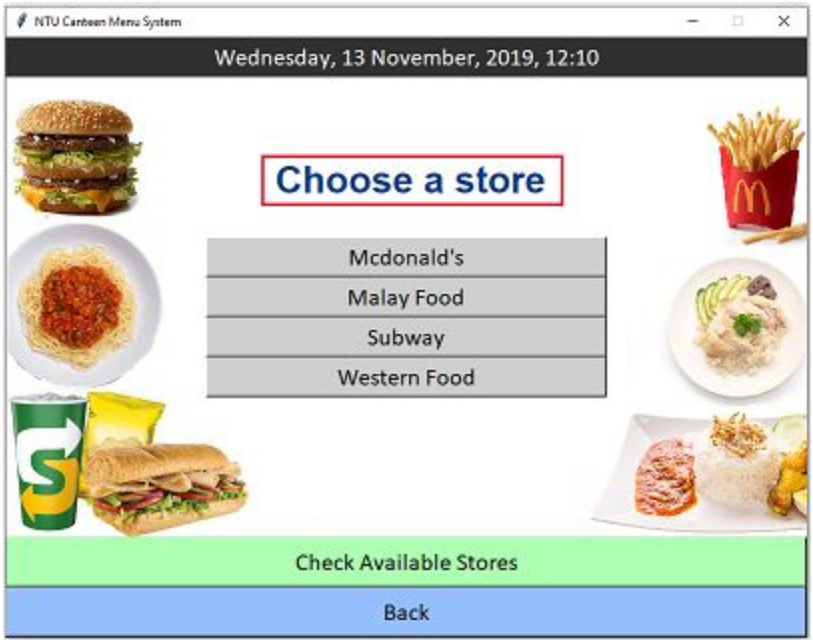
The “Waiting Time” button opens a popup window for the user to input the number of people in the queue to generate an estimated waiting time.

2.5 Choose A Date (Feature D)

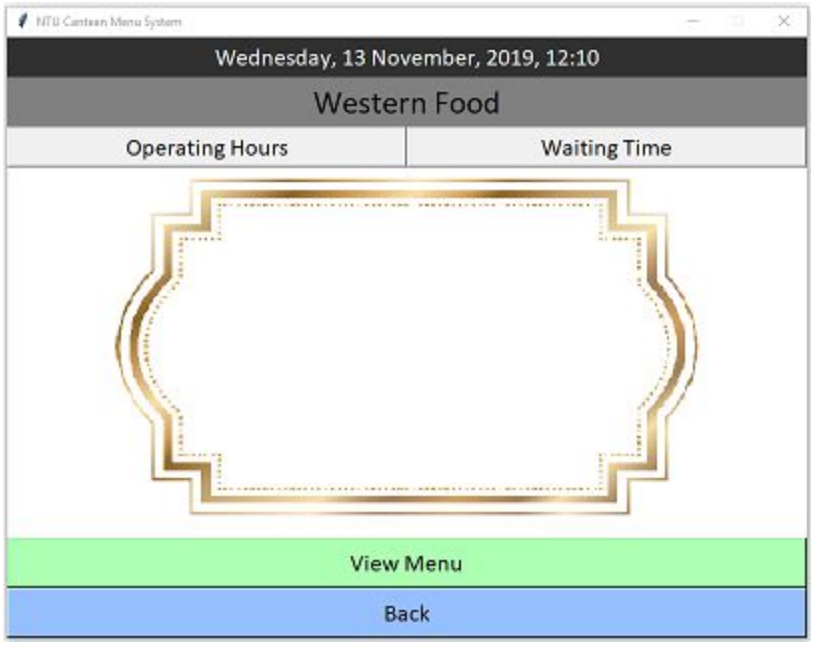


In this frame, the user can choose their desired date and time via drop-down lists. The “Next” button saves the user inputted date and time and proceeds to the next frame. The clock is set to the inputted date and time, shown below.



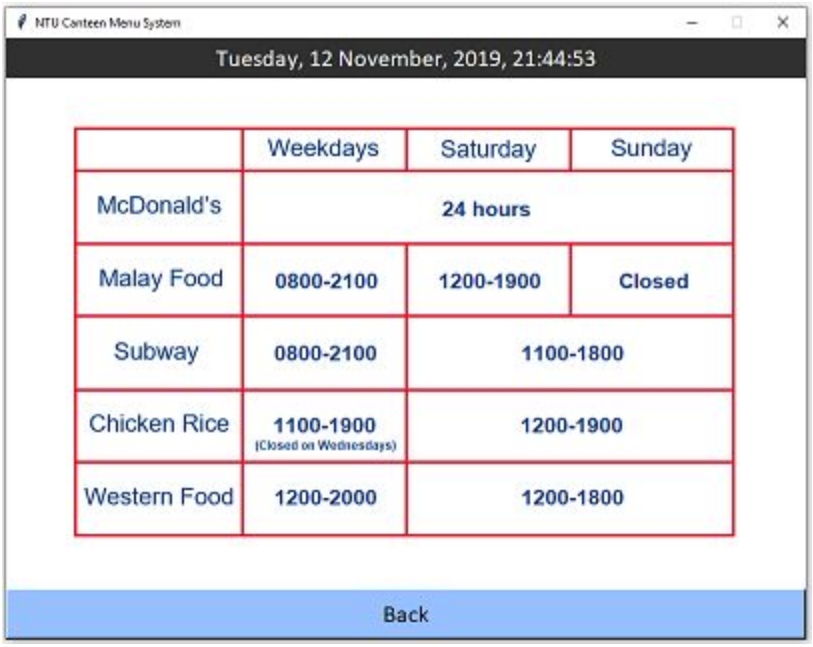


The functionality of this frame is similar to those above whereby the user can click on the “Check Available Stores” button, with stores listed according to the user input date and time instead.



The other functions like the “Operating Hours” button and “Waiting Time” button act similarly to those seen above.

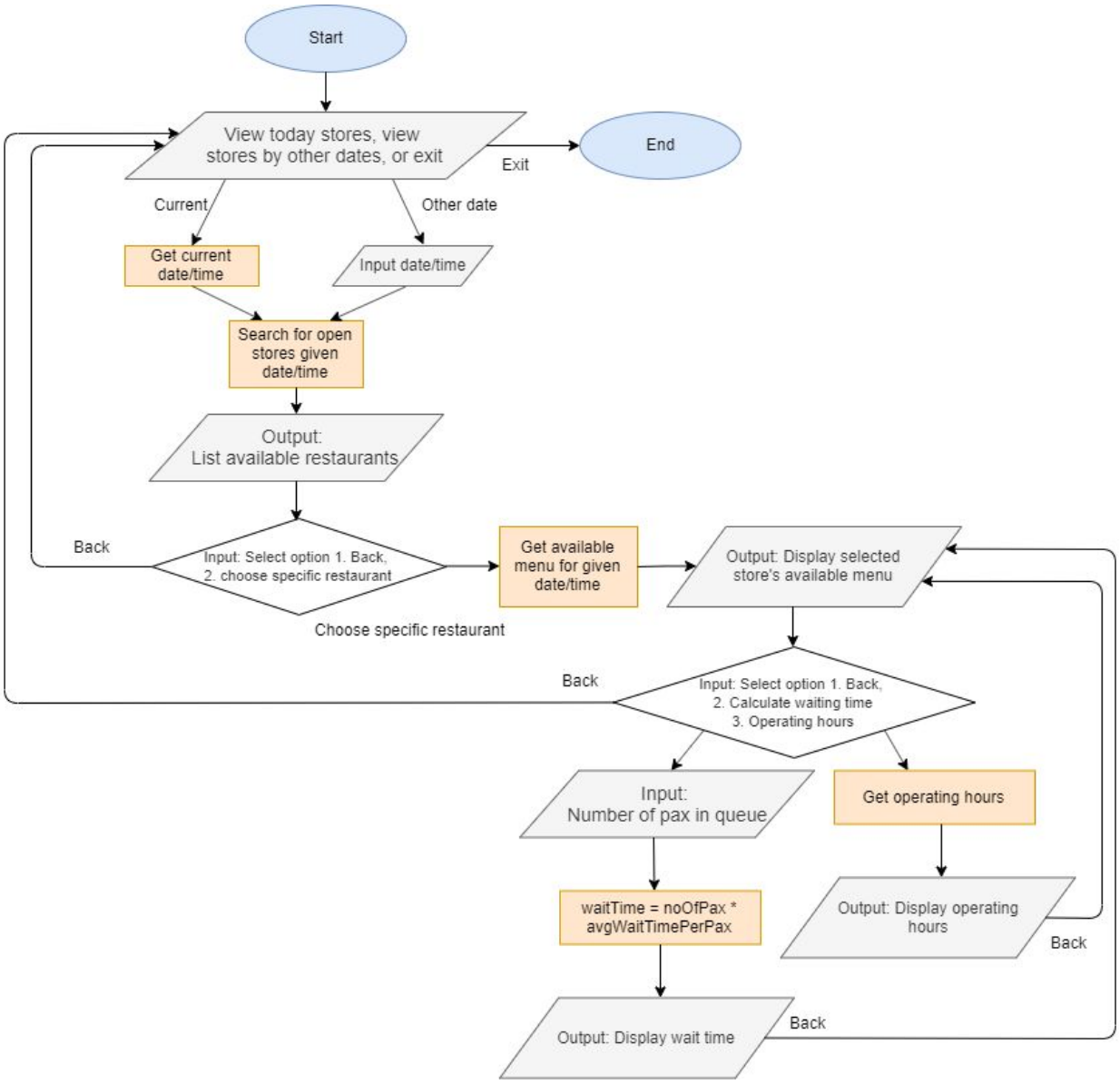
2.6 Operating Hours Frame (Feature F)



This is the “Operating Hours” frame, accessible from the start page. It displays the operating of all stores in one frame.

3. Algorithm Design

3.1 Top-level flow chart



This is the basic functionality of the application. There are two main choices: viewing stores today or at a given date. Then, the program will find any available stores. For each store, the user can view operating hours, calculate waiting time, or return.

3.2 Important user-defined functions

3.2.1 isRestaurantAvailable

```
def isRestaurantAvailable(restaurant,date = getTodaysDate(),time = getCurrentTime()):
```

...

```
operatingHours = open('Operating Hours/operatinghours.csv', 'r')
# open the operatinghours csv file
with operatingHours:
    csv_reader = csv.reader(operatingHours, delimiter=",")
    currentDay = dayTable.index(day) + 1
    #this will represent the column number we will be checking within the csv file.
    for row in csv_reader:
        if not restaurant in row:
        #skip until reach correct row
            continue
        if row[currentDay].lower() == "closed":
        #if closed during that day, return false
            return False
        if row[currentDay].lower() == "24 hours":
        #if the restaurant is always open for that day, return true
            return True
        time1 = row[currentDay][0:4]
        time2 = row[currentDay][5:9]
        if isWithinTime(int(time1), int(time2), int(time)):
        #if restaurant opening(time1) < currentTime < closing(time2), return true
            return True
    else:
        return False
```

This function performs a binary search for the inputted restaurant (first if statement). Then, it will return False if closed, or True if open.

3.2.2 getMenu

```
def getMenu(restaurant, date = getTodaysDate(), time = getCurrentTime()):
```

...

```
    #try to open menu csv file. if it does not exist in the directory, return false
    try:
        menu = open('Menus/'+restaurant+'.csv', 'r')
    except:
        print("Restaurant does not exist")
        return False
```

Check if ‘restaurant’ file is located inside Menus folder. Since there is a dedicated file for each restaurant, it is important that *FileNotFoundError* is caught if the file does not exist within the directory.

```
menuTimings = getMenuTiming(restaurant)
currentMenu = ""
time = time[0:2] + time[3:5]

for i in menuTimings:
    #check which menu to use according to time
    if i.lower() == "no":
    #skip if menu not available for lunch/breakfast/dinner
        continue
    time1 = i[0:4]
    time2 = i[5:9]
    if isWithinTime(int(time1), int(time2), int(time)):
        currentMenu = menuTable[menuTimings.index(i)]
    #Select from menuTable (Breakfast, lunch, dinner) the correct menu for the given time
        break
```

This portion determines which menu to use. *getMenuTiming()* returns list *menuTimings* of timings for breakfast, lunch and dinner. The for loop then chooses which menu is correct for the given time.

```
with menu:
    csv_reader = csv.reader(menu, delimiter=",")
    count = 0
    for row in csv_reader:
        if count < 2:
            #skip the header rows (row 0 and 1)
            count += 1
            continue
        itemName = row[0]
        itemPrice = row[1]
        if row[dayIndex].lower() == "yes" and currentMenu in row:
            #if column day == yes, that item is available for that day, and is the correct menu
            menuDict[itemName] = '${:,.2f}'.format(float(itemPrice))
            #append formatted price (convert to float, format float to dollar and cents).
return menuDict
```

Lastly, another search is performed, selecting the chosen menu’s items to add to dictionary *menuDict*.

3.2.3 calculateWaitingTime

```
def calculateWaitingTime(restaurant, numOfPax):
    #calculate the waiting time per pax (avg time * number of pax)
    operatingHours = open('Operating Hours/operatinghours.csv', 'r')
    # open the operatinghours csv file
    with operatingHours:
        csv_reader = csv.reader(operatingHours, delimiter=",")
        for row in csv_reader:
            if not restaurant in row:       #skip until reach correct row
                continue
            return int(row[11])*numOfPax    #return calculation
    return 0                                #if never found
```

Extracts the waiting time for the given restaurant from the *operatinghours* file. Then, return a calculation of waiting time * number of people.

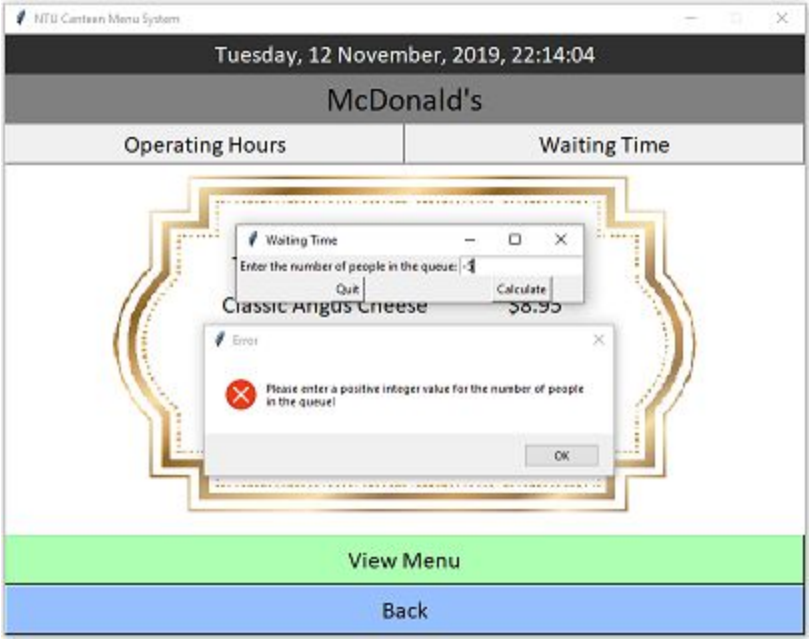
3.2.4 getOperatingHours

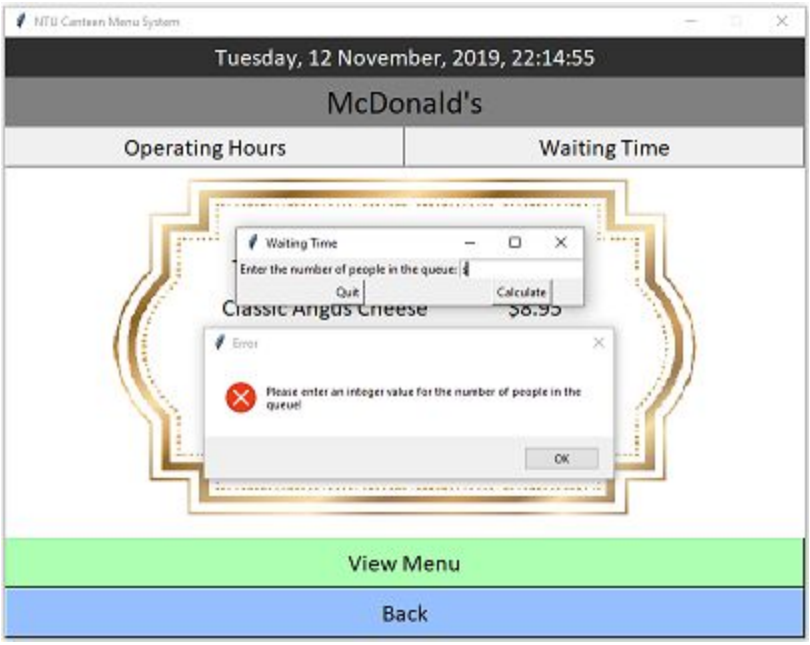
```
def getOperatingHours(restaurant):
    timingTable = []
    operatingHours = open('Operating Hours/operatinghours.csv', 'r')  # open the operatinghours csv file
    with operatingHours:
        csv_reader = csv.reader(operatingHours, delimiter=",")
        for row in csv_reader:
            if not restaurant in row:       #skip until reach correct row
                continue
            else:
                timingTable = row[1:8]          #clone the days columns for specified restaurant
                break                           #now we can exit early
    #parse data and extract date
    if timingTable[0:5].count(timingTable[0]) == 5:
        #if monday timing is repeated throughout the week, use weekdays;
        operatingHoursMessage = "Weekdays: {0} \n" \
                                "Saturday: {1} \n" \
                                "Sunday: {2}".format(timingTable[0], timingTable[5], timingTable[6])
    else:                                       #else, specify the timing for each day
        operatingHoursMessage = "Monday: {0} \nTuesday: {1} \nWednesday: {2} \n" \
                                "Thursday: {3} \nFriday {4} \nSaturday: {5} \n" \
                                "Sunday: {6}".format(timingTable[0],timingTable[1],\
                                                     timingTable[2],timingTable[3],\
                                                 timingTable[4],timingTable[5],timingTable[6])
```

Extracts a restaurant's operating hours and formats for display. If Mon-Fri have the same timing, display as weekdays. Otherwise, display each day separately.

4. Program Testing

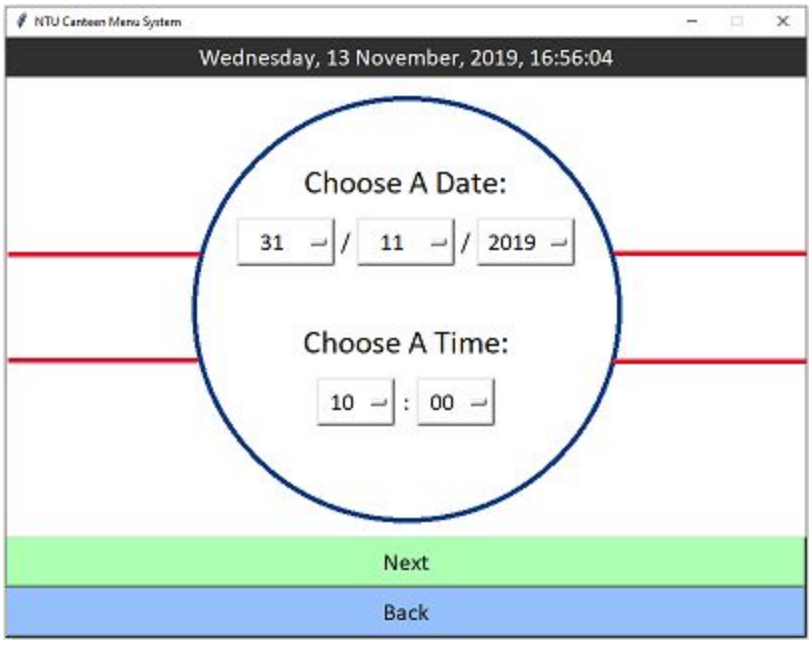
4.1 User input errors (Waiting Time)





The waiting time function checks for scenarios where the user inputs a negative number or a string and returns an error as shown above.
In both cases, the user can return and enter the correct value.

4.2 User input errors (Date and Time)



The application checks for invalid date inputs using try/except. For example, if “31/11/2019” was entered, the next frame would prompt the user to enter a valid date.

```
def isDateValid(date):
        dmy = date.split("/")
        try: #try to use datetime to return a date
                datetime.datetime(year=int(dmy[2]),month=int(dmy[1]),day=int(dmy[0]))
                return True
        except ValueError: #if date does not exist, throw a ValueError
                return False
```

Using the datetime module, we can check if the date exists. If *datetime()* throws an error, the input is an invalid date.

```
print(isDateValid('29/2/2019'))                    False
print(isDateValid('29/2/2020')) #leap year         True
print(isDateValid('31/11/2019'))                   False
```

4.3 Function input errors (Time)

```
def checkTimeValid(time):   #check if a time is 1. formatted as hh:mm and 2. is a valid time
    if len(time) != 5:  #hh:mm has length of 5
        return False
    time = time[0:2] + time[3:5] #remove colon
    if time.isdigit() == False: #should be an integer
        return False
    elif int(time) > 2359:    #check if time is within valid range
        return False
    else:
        return True
```

This function checks for syntax correctness in input. For example, a time of “23:59” would be accepted, but “24:00” would return False.

```
print(checkTimeValid("23:59"))
print(checkTimeValid("24:00"))
print(checkTimeValid("invalid"))
```

```
True
False
False
```

4.4 Function input errors (Date)

```
def getDayFromDate(date):   #format date as string "dd/mm/yyyy"
    dayTable = ("Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday")
    #Test to make sure input is valid for robustness
    if type(date) != str:   #make sure it is a string
        print("Please input date as a string")
        return False
    if len(date) != 10:
        print("Format according to dd/mm/yyyy")
        return False
    dayMonthYear = date.split('/') #split string by /
    if len(dayMonthYear) != 3:  #make sure there are three elements: day, month, and year
        print("Format according to dd/mm/yyyy")
        return False

    try:
        dayFromDate = dayTable[calendar.weekday(int(dayMonthYear[2]),int(dayMonthYear[1]),int(dayMonthYear[0]))]
    except ValueError:      # catch any invalid dates, if invalid return false
        print("Invalid date given!")
        return False
    return dayFromDate  # return day as string
```

This function returns *False* if the given date is invalid, checking for syntax and validity of date. Otherwise, return the day corresponding to the date.

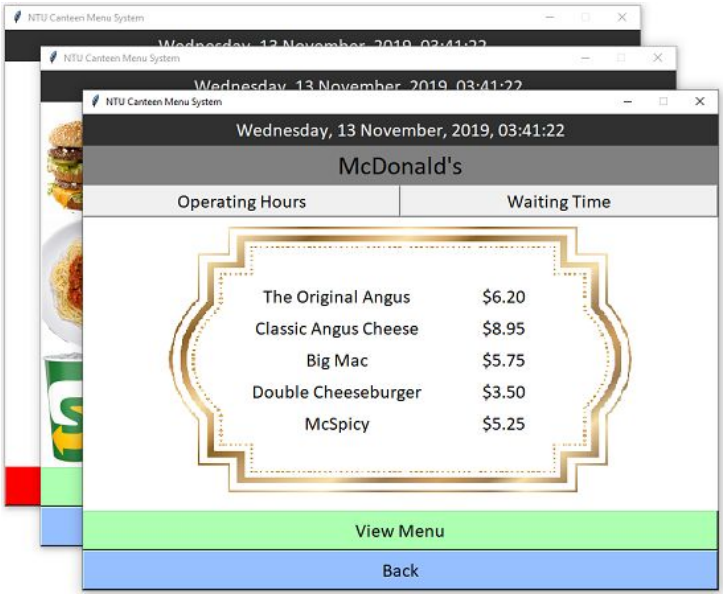
```
print(getDayFromDate("31/11/2019"))
print(getDayFromDate("29/10/2018"))
```

```
Invalid date given!
False
Monday
```

5. Reflection & Difficulties

5.1 Introduction to Object-Oriented Programming (OOP)

We couldn't put multiple frames into one single window using the Python knowledge gained from the course, as coding a new window into button presses would only create more windows, as shown below, which is not user-friendly.



With the use of classes and methods in OOP, each frame is its own class. Thus, we can initialize all the frames upon startup and then call the frames we want shown upon button presses using tkraise().

```
    # Dictionary of all frames in the application
    # Each frame is its own class
    self.frames = {}
    for F in (StartPage, ChooseAStore, StoreFrame, ViewOperatingHours, ChooseADate, ChooseAStore_UserDateAndTime,StoreFrame_UserDateAndTime):

        frame = F(container, self)
        self.frames[F] = frame
        frame.grid(row=0, column=0, sticky="nsew")

    self.show_frame(StartPage)

# Function to bind to button that raises frame to the top of the window
def show_frame(self, cont):

    frame = self.frames[cont]
    frame.tkraise()
```

5.2 Accessing variables between classes and functions

Knowing how to pass assigned variables between different classes and functions was crucial for bringing the user inputted date and time over different frames.

```
def pressed():
    InputDate = variable_date.get() + "/" + variable_month.get() + "/" + variable_year.get()
    InputTime = variable_hour.get() + ":" + variable_minute.get()
```

```
def CheckStores():
    n = 200
    if isRestaurantAvailable("McDonald's",InputDate,InputTime):
```

```
    if isRestaurantAvailable("McDonald's",InputDate,InputTime):
NameError: name 'InputDate' is not defined
```

In the above example, the local variables “InputDate” and “InputTime” were being assigned solely within the function.
Using the locally assigned variables “InputDate” and “InputTime” in a different frame throws a “NameError”.

```
7   InputDate = getTodaysDate()
8   InputTime = getCurrentTime()
```

```
def pressed():
    global InputDate
    global InputTime
    InputDate = variable_date.get() + "/" + variable_month.get() + "/" + variable_year.get()
    InputTime = variable_hour.get() + ":" + variable_minute.get()
```

The solution was to use the *global* keyword to reassign global variables “InputDate” and “InputTime” within functions.

We defined those variables at the top first. Then, we used *global* inside the “pressed()” function to ensure we are reassigning at the correct global scope. After amending this, the function was able to work correctly.

5.3 Excessive amounts of frames

```
 9 > class MiniProject(tk.Tk): ···
36
37 > class StartPage(tk.Frame): ···
53
54 > class ChooseAStore(tk.Frame): ···
75
76 > class ChooseADate(tk.Frame): ···
136
137 > class ChooseAStoreInput(tk.Frame): ···
162
163 > class Mcdonald(tk.Frame): ···
221
222 > class Subway(tk.Frame): ···
280
281 > class ChickenRice(tk.Frame): ···
339
340 > class MalayFood(tk.Frame): ···
398
399 > class WesternFood(tk.Frame): ···
457
458 > class McdonaldInput(tk.Frame): ···
516
517 > class SubwayInput(tk.Frame): ···
575
576 > class ChickenRiceInput(tk.Frame): ···
634
635 > class MalayFoodInput(tk.Frame): ···
693
694 > class WesternFoodInput(tk.Frame): ···
752
753   app = MiniProject()
754   app.mainloop()
```

```
16 > class MiniProject(tk.Tk): ···
47
48 > class StartPage(tk.Frame): ···
81
82 > class ChooseAStore(tk.Frame): ···
158
159 > class StoreFrame(tk.Frame): ···
252
253 > class ViewOperatingHours(tk.Frame): ···
277
278 > class ChooseADate(tk.Frame): ···
380
381 > class ChooseAStore_UserDateAndTime(tk.Frame): ···
462
463 > class StoreFrame_UserDateAndTime(tk.Frame): ···
555
556   app = MiniProject()
557   app.mainloop()
```

Using *global* keywords also allowed us to shorten the code and reduce the amount of “classes” needed for each individual store frame. By assigning a function that changes the global variable “Selection” to the store buttons, we can create a single store frame that retrieves the corresponding menus based on the “Selection” variable.

```
InputDate = getTodaysDate()
InputTime = getCurrentTime()
Selection = ""
```

```
def b1():
    global Selection
    Selection = "McDonald's"
def b2():
    global Selection
    Selection = "Malay Food"
def b3():
    global Selection
    Selection = "Subway"
def b4():
    global Selection
    Selection = "Chicken Rice"
def b5():
    global Selection
    Selection = "Western Food"
```


```
def ViewMenu():
    n = 210
    for item,price in getMenu(Selection, getTodaysDate(), getCurrentTime()).items():
```

5.4 File Formats

Initially a .txt file was used for storing the store info. However, there were two complications.

One, it became increasingly complex with more stores.

Two, it became hard to parse and edit manually through text editor.

To overcome these limitations we used .csv, a datasheet format.

```
Restaurant,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday,
McDonald's,24 hours,24 hours,24 hours,24 hours,24 hours,24 hours,24
```

The module csv comes with its own reader to be used with .csv files.

```
import csv

csv_reader = csv.reader(operatingHours, delimiter=",")
```

Importantly, the csv reader has a *delimiter* parameter specifying how each line is split, lending towards better readability and data extraction. Thus, we could solve both key problems.

5.5 Efficient Function Use

To ensure robustness, the program needed to fit all cases. However, one problem was that coding cases for each function detracted from the readability and simplicity of our design.

Through pattern recognition, we found that many elements were being repeated throughout. Our solution was to create functions for each scenario. Thus, we could call the function within the main function, improving readability.

```
def getDayFromDate(date):   #format date as string "dd/mm/yyyy"
    dayTable = ("Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday")
    #Test to make sure input is valid for robustness
    if type(date) != str:   #make sure it is a string
        print("Please input date as a string")
        return False
    if len(date) != 10:
        print("Format according to dd/mm/yyyy")
        return False
    dayMonthYear = date.split('/') #split string by /
    if len(dayMonthYear) != 3:  #make sure there are three elements: day, month, and year
        print("Format according to dd/mm/yyyy")
        return False

    try:
        dayFromDate = dayTable[calendar.weekday(int(dayMonthYear[2]),int(dayMonthYear[1]),int(dayMonthYear[0]))]
    except ValueError:      # catch any invalid dates, if invalid return false
        print("Invalid date given!")
        return False
    return dayFromDate  # return day as string
```

```
def getMenu(restaurant, date = getTodaysDate(), time = getCurrentTime()):
    day = getDayFromDate(date)
```

For example, we developed a function for extracting a day of the week from an inputted date. This function is then used in other functions, simplifying the code through encapsulation.

6. References

“datetime - Basic date and time types,” *datetime - Basic date and time types - Python 3.8.0 documentation*. [Online]. Available: https://docs.python.org/3/library/datetime.html.

“calendar - General calendar-related functions,” *calendar - General calendar-related functions - Python 3.8.0 documentation*. [Online]. Available: https://docs.python.org/3/library/calendar.html.

“Graphical User Interfaces with Tk,” *Graphical User Interfaces with Tk - Python 3.8.0 documentation*. [Online]. Available: https://docs.python.org/3/library/tk.html.

“csv - CSV File Reading and Writing,” *csv - CSV File Reading and Writing - Python 3.8.0 documentation*. [Online]. Available: https://docs.python.org/3/library/csv.html.

“time - Time access and conversions,” *time - Time access and conversions - Python 3.8.0 documentation*. [Online]. Available: https://docs.python.org/3/library/time.html.

Word count = 1200