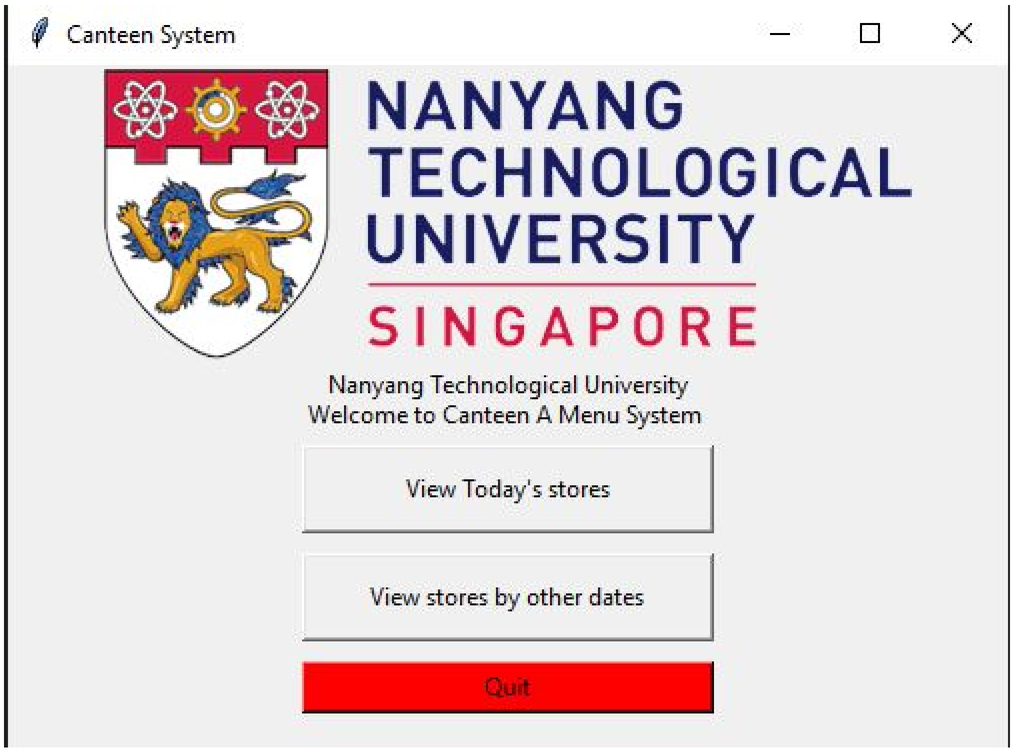
NTU NORTH SPINE CANTEEN SYSTEM


Canteen System
NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE
Nanyang Technological University
Welcome to Canteen A Menu System
View Today's stores
View stores by other dates
Quit

Acharya Atul

Aravind S/O Sivakumaran

Joshua Chan

11.11.2019

NTU SCSE ICT MINI PROJECT

CONTRIBUTIONS

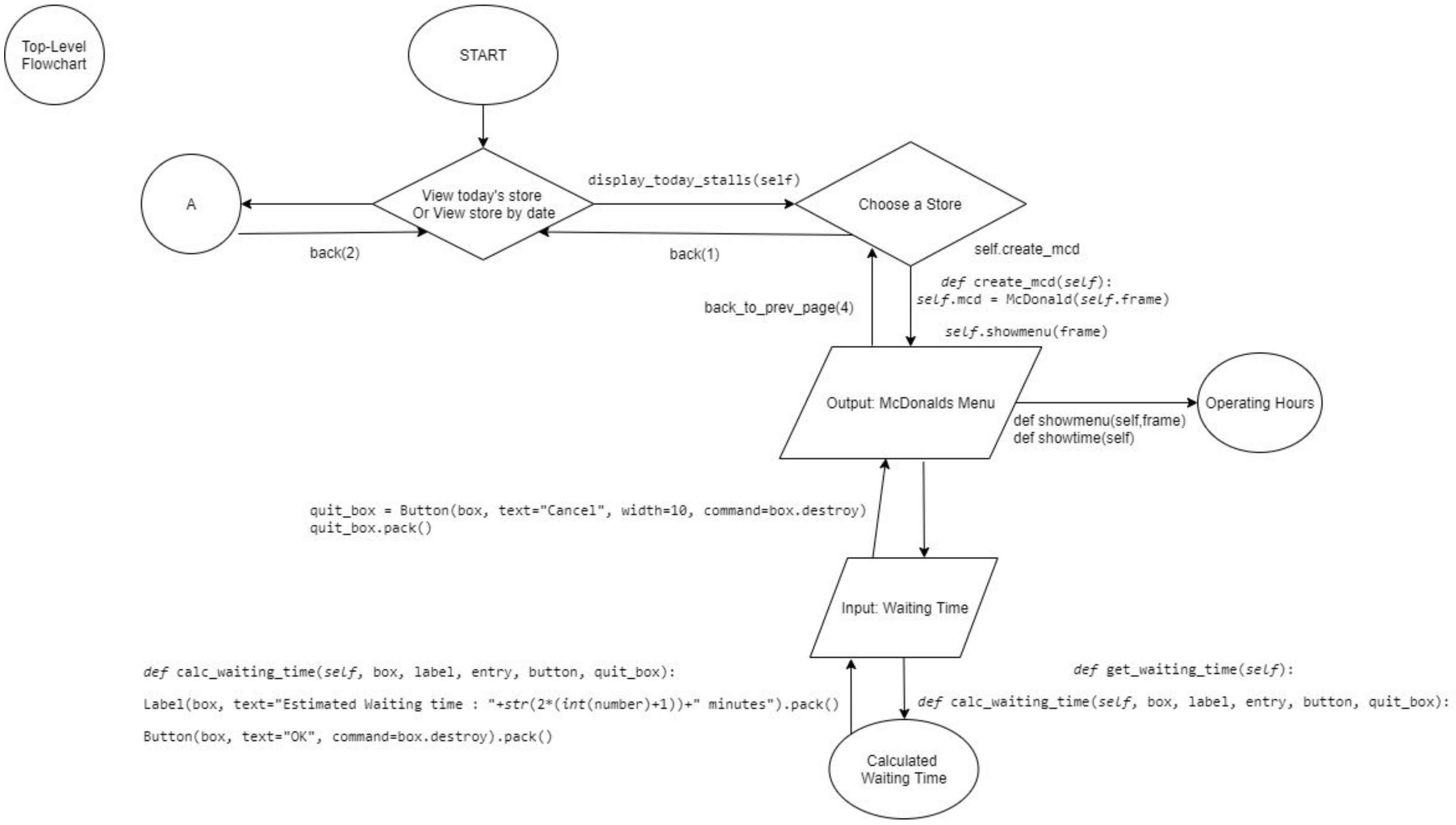
Acharya Atul: Main Frame structure, back buttons and Overall Algorithmic Structure

Aravind S/O Sivakumaran: Stall Classes along with the other groupmates

Joshua Chan: External menu and time files, displaying menus based on datetime input and user interface design

ALGORITHM DESIGN

The program is designed to display menus and stall opening times for food stalls in the North Spine Canteen. The following is the top-level flowchart for the program:



```
def create_mcd(self):
self.mcd = McDonald(self.frame)
self.showmenu(frame)
```

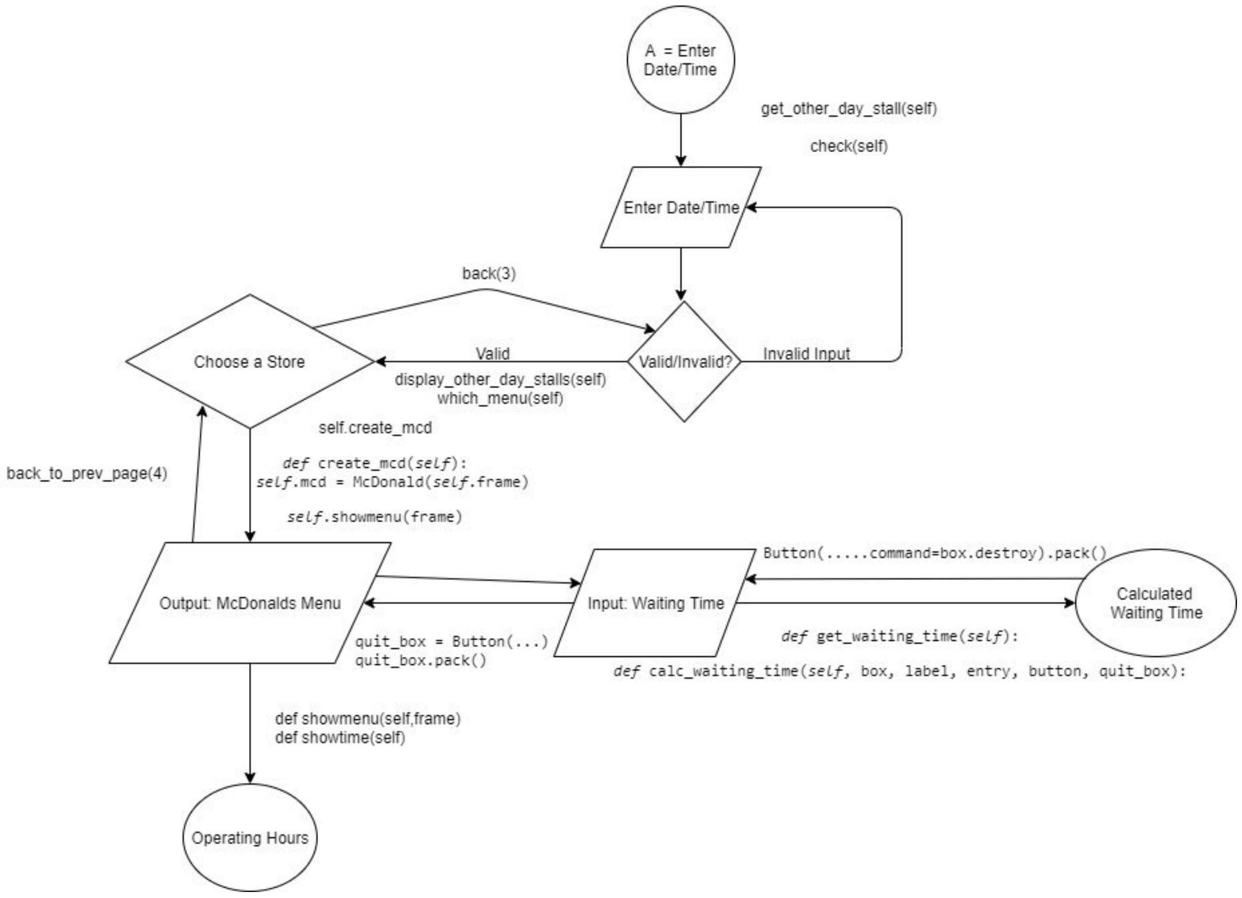
```
def showmenu(self,frame)
def showtime(self)
```

```
quit_box = Button(box, text="Cancel", width=10, command=box.destroy)
quit_box.pack()
```

```
def calc_waiting_time(self, box, label, entry, button, quit_box):
Label(box, text="Estimated Waiting time : "+str(2*(int(number)+1))+" minutes").pack()
Button(box, text="OK", command=box.destroy).pack()
```

```
def get_waiting_time(self):
def calc_waiting_time(self, box, label, entry, button, quit_box):
```

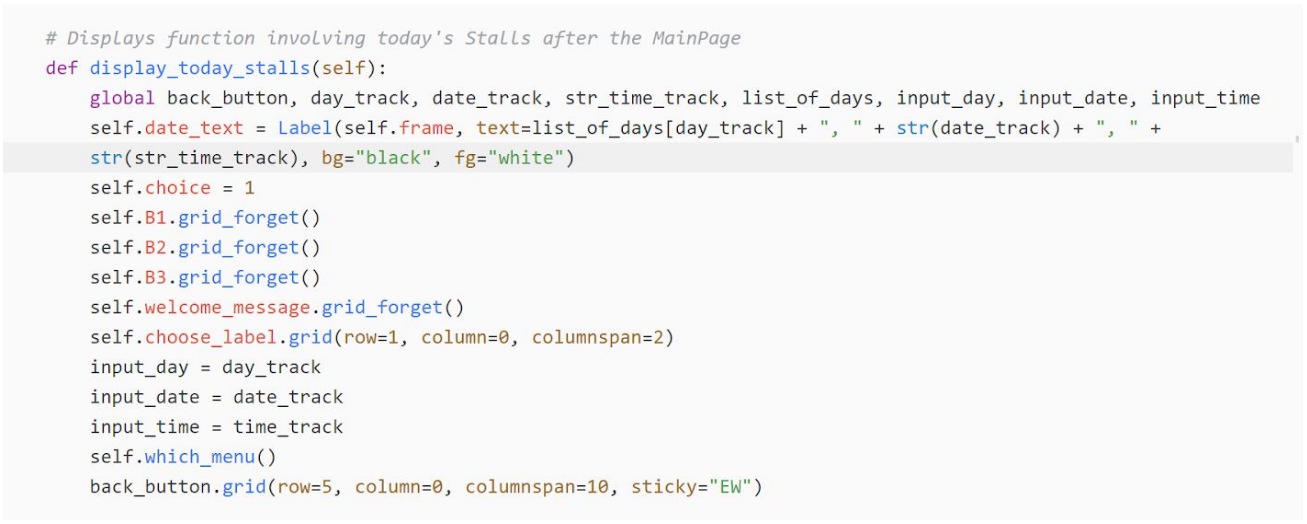
Top-level flowchart (right branch)



Top-level flowchart (Left branch)

The program begins with displaying the main frame, which is done using the *display_mainpage* function. This loads 3 buttons, with the option for the user to (1) view what is open currently, (2) see what is open based on a user-defined date and/or time, and (3) to quit the program. Option (1) leads down the right branch of the flowchart while option (2) goes down the left branch.

When we go down the right branch by clicking to view the current stalls, the function *display_today_stalls* populates the frame with buttons for each of the stalls that are open at the current system date and time. This is done by assigning today's day, date and time to the global variables *input_day*, *input_date* and *input_time* respectively:



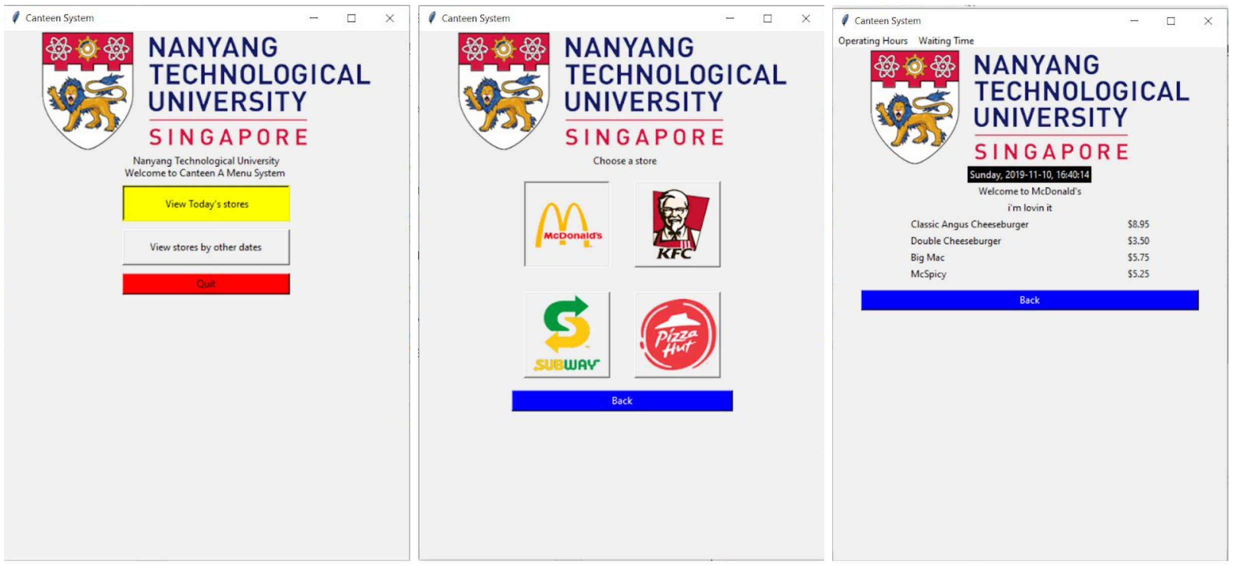
```
# Displays function involving today's Stalls after the MainPage
def display_today_stalls(self):
    global back_button, day_track, date_track, str_time_track, list_of_days, input_day, input_date, input_time
    self.date_text = Label(self.frame, text=list_of_days[day_track] + ", " + str(date_track) + ", " +
    str(str_time_track), bg="black", fg="white")
    self.choice = 1
    self.B1.grid_forget()
    self.B2.grid_forget()
    self.B3.grid_forget()
    self.welcome_message.grid_forget()
    self.choose_label.grid(row=1, column=0, columnspan=2)
    input_day = day_track
    input_date = date_track
    input_time = time_track
    self.which_menu()
    back_button.grid(row=5, column=0, columnspan=10, sticky="EW")
```

The above function is reliant on the *which_menu* function, which combines date and time information using the datetime library to determine what stalls are open at that time. In this snippet, the function checks if it is a weekday, and if it is, checks if it is open at that particular time before populating the grid with the appropriate stall buttons:

The user can then click on the stall that he wishes to see more information for. Upon clicking, the program switches to the class frame for the stall that the user has selected. The stall's dishes are then populated from the 'menus' file, and places them on the grid:

```
self.menu = {}
if input_time <= datetime.time(11):
    for key, value in menus['mc_donalds'][0].items():
        updater = {Label(frame, text=key):
                       Label(frame, text=value)}
        self.menu.update(updater)
    i = 4
    for key, value in self.menu.items():
        key.grid(row=i, column=0, sticky="W", padx=60)
        value.grid(row=i, column=1, sticky="E", padx=60)
        i += 1
else:
    for key, value in menus['mc_donalds'][1].items():
        updater = {Label(frame, text=key):
                       Label(frame, text=value)}
        self.menu.update(updater)
    i = 4
    for key, value in self.menu.items():
        key.grid(row=i, column=0, sticky="W", padx=60)
        value.grid(row=i, column=1, sticky="E", padx=60)
        i += 1
```

Here is the aforementioned process (from left to right) as seen by the user when he makes the corresponding button clicks in the program:



Sunday, 2019-11-10, 16:40:14
Welcome to McDonald's
i'm lovin it

Classic Angus Cheeseburger	$8.95
Double Cheeseburger	$3.50
Big Mac	$5.75
McSpicy	$5.25

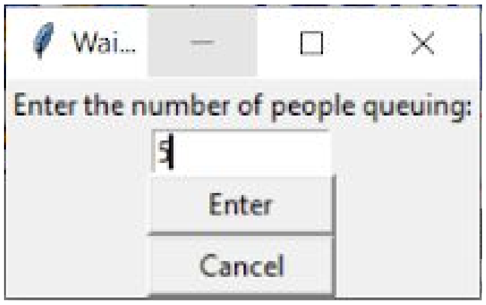
The left branch is similar to the right branch of the flowchart, except that the user can provide a different date and time. The input fields are combo boxes which allow the user to select the date and time from a series of dropdown boxes. The user-defined date and time are then assigned to the *input_date* and *input_time* variables:

```
        input_date = datetime.date(int(self.year_entry.get()), int(self.month_entry.get()),
                                   int(self.day_entry.get()))
        input_time = datetime.time(int(self.hour_entry.get()), int(self.minute_entry.get()))
        integer_year = int(self.year_entry.get())
```

If the input is valid, the program proceeds to return the list of stalls open at the date and time specified by the user. If not, an error is raised and the user is prompted to re-enter a valid date/time. Beyond this point, the left branch of the flowchart behaves similarly to the right branch.

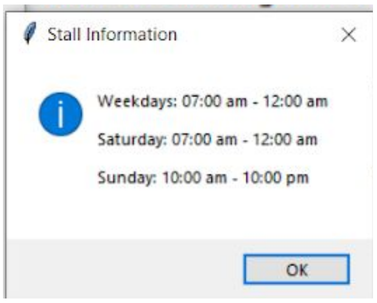
Users can also check the estimated waiting time at each stall. By clicking on the “Waiting Time” button on any of the menu pages, a dialogue box pops up, prompting the user to enter the number of people in the queue. The input is then applied to the formula defined in *calc_waiting_time*, as seen below:

```
def calc_waiting_time(self, box, label, entry, button, quit_box):
    number = entry.get()
    if number.isnumeric() and 0 <= int(number):
        label.destroy()
        entry.destroy()
        button.destroy()
        quit_box.destroy()
        Label(box, text="Estimated Waiting time : " + str(2 * (int(number) + 1)) + " minutes").pack()
        Button(box, text="OK", command=box.destroy).pack()
    else:
        messagebox.showerror("INVALID", "Enter a Valid Number!")
```



On the same menu frame, by clicking on the “Operating Hours” button, the user can see the operating times for that stall. It extracts the opening and closing times from the ‘times’ file and uses the built-in *strftime* function to format the time into a readable string for the user:

```python
def showtime(self):
    t_weekday = "Weekdays: " + times['kfc_'][0][0].strftime("%I:%M %p").lower() + " - " + times['kfc_'][0][
        1].strftime("%I:%M %p").lower() + "\n\n"
    if times['kfc_'][1] is not None:
        t_saturday = "Saturday: " + times['kfc_'][1][0].strftime("%I:%M %p").lower() + " - " +
        times['kfc_'][1][
            1].strftime("%I:%M %p").lower() + "\n\n"
    else:
        t_saturday = "Saturday: Closed"
    if times['kfc_'][2] is not None:
        t_sunday = "Sunday: " + times['kfc_'][2][0].strftime("%I:%M %p").lower() + " - " +
        times['kfc_'][2][1].strftime("%I:%M %p").lower() + "\n\n"
    else:
        t_sunday = "Sunday: Closed"
    string = t_weekday + t_saturday + t_sunday
```



```
Weekdays: 07:00 am - 12:00 am

Saturday: 07:00 am - 12:00 am

Sunday: 10:00 am - 10:00 pm
```

As mentioned previously, ‘menus’ and ‘times’ are files that have to be unpickled and loaded into the main program for use.

```python
in_menus = open('menus', 'r+b')
menus = pickle.load(in_menus)

in_times = open('times', 'r+b')
times = pickle.load(in_times)
```

‘menus’ is used to store the dishes of every stall. It is a dictionary of dictionaries for most stalls, with dish name being the key and price being the value of the inner dictionary. McDonald’s is an exception as it has a breakfast menu, and thus, it contains a list, with the breakfast menu dictionary and lunch/dinner menu dictionary as elements of the list:

```
menus = {'mc_donalds':[{'Hotcakes':'$4.95', 'Sausage McMuffin with Egg':'$4.20', 'Big 
Breakfast':'$5.90'},
{'Classic Angus Cheeseburger':'$8.95', 'Double Cheeseburger':'$3.50', 'Big Mac':'$5.75', 
'McSpicy':'$5.25'}],
'kfc_':{'2 pcs Chicken Meal':'$10.15', '3 pcs Chicken Meal':'$12.15', 'Zinger':'$5.30', 'Curry 
Rice Bowl':'$5.30'},
'subway':{'6-inch':'$5.70', 'Footlong':'$9.50', 'Wrap':'$6.10'},
'pizza_hut':{'Hawaiian':'$10.80', 'Pepperoni':'$10.80', 'Seafood Deluxe':'$11.20'},
'sandwich_guys':{'BBQ Pulled Pork':'$6.00', 'Philly Cheesesteak':'$7.00', 'Mexicana Grilled 
Chicken':'$6.00', 'Cubano':'$7.00'}}
```

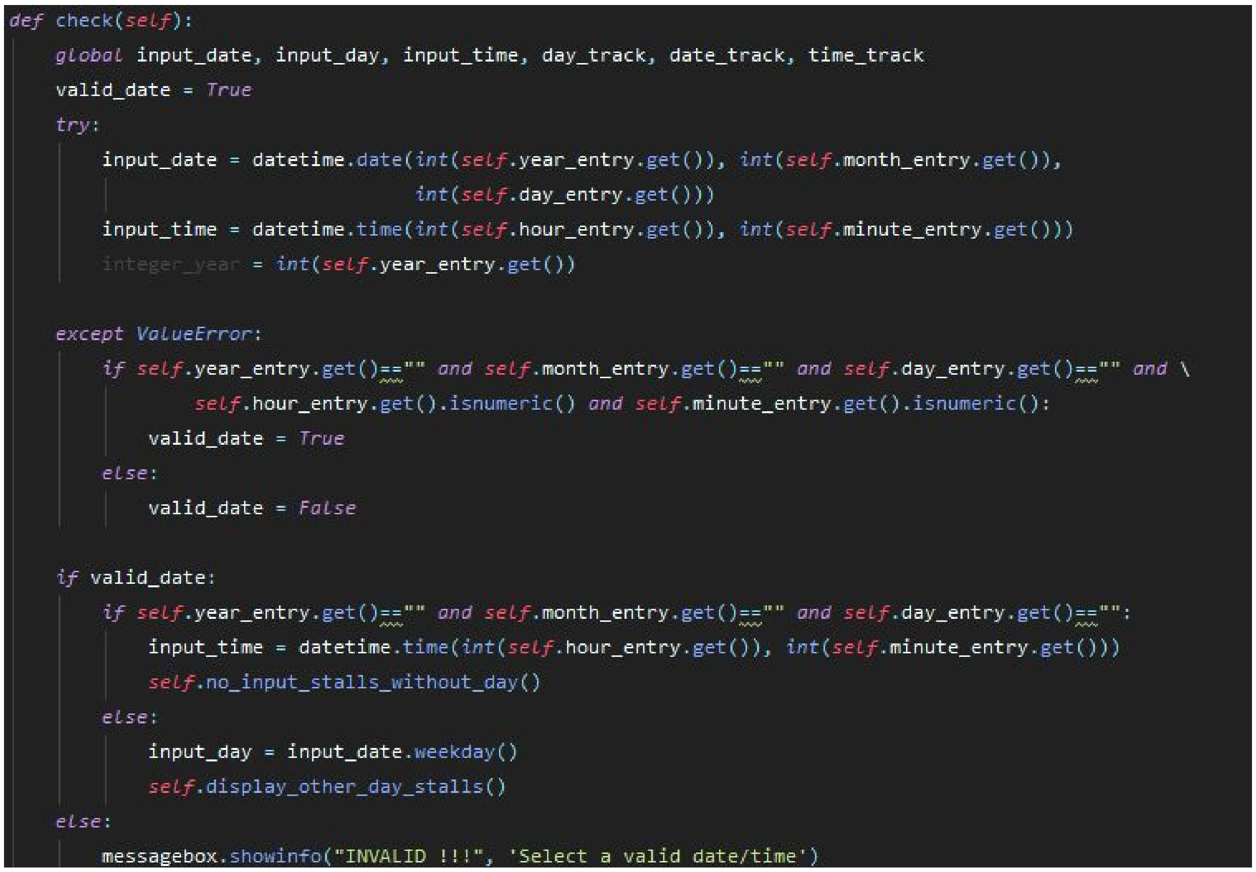
‘times’ contains the opening and closing times of each stall. It is a dictionary of lists, with each entry in the list containing the opening and closing time for each stall. The timings are divided into weekdays, Saturday and Sunday from indexes 0 to 2 respectively:

```
times = {'mc_donalds':[[time(7,0),time(0,0)],[time(7,0),time(0,0)],[time(10,0),time(22,0)]],
'kfc_':[[time(7,30),time(22,0)],[time(11,0),time(20,0)],[time(11,0),time(20,0)]],
'subway':[[time(8,0),time(21,0)],[time(11,0),time(18,0)],[time(11,0),time(18,0)]],
'pizza_hut':[[time(11,0),time(22,0)],[time(11,0),time(21,0)],[time(11,0),time(20,0)]],
'sandwich_guys':[[time(10,0),time(20,0)],[time(10,0),time(15,0)],None]}
```

PROGRAM TESTING

When requesting date and time input from the user, we have to make sure that a valid input is received. By using a Combobox and setting its state to ‘readonly’, it ensures that the user can only select a range of integers that is provided in the dropdown box. This will eliminate the input of the wrong type, such as strings or floats.

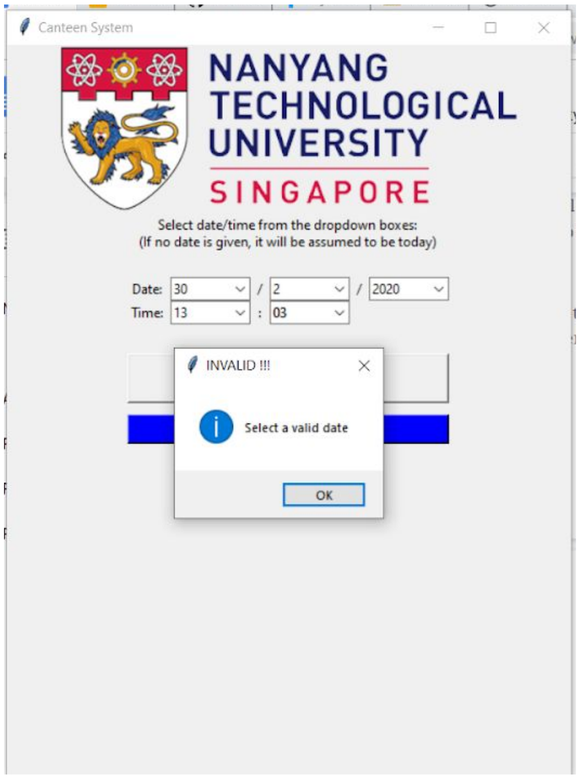
However, there is still a possibility of an invalid input as the user could still select a date that does not exist (e.g. 30 Feb). Thus, we designed a *check* function to catch any errors that cannot be mitigated by limiting the user’s input:



```
def check(self):
    global input_date, input_day, input_time, day_track, date_track, time_track
    valid_date = True
    try:
        input_date = datetime.date(int(self.year_entry.get()), int(self.month_entry.get()),
                                   int(self.day_entry.get()))
        input_time = datetime.time(int(self.hour_entry.get()), int(self.minute_entry.get()))
        integer_year = int(self.year_entry.get())

    except ValueError:
        if self.year_entry.get()=="" and self.month_entry.get()=="" and self.day_entry.get()=="" and \
                self.hour_entry.get().isnumeric() and self.minute_entry.get().isnumeric():
            valid_date = True
        else:
            valid_date = False

    if valid_date:
        if self.year_entry.get()=="" and self.month_entry.get()=="" and self.day_entry.get()=="":
            input_time = datetime.time(int(self.hour_entry.get()), int(self.minute_entry.get()))
            self.no_input_stalls_without_day()
        else:
            input_day = input_date.weekday()
            self.display_other_day_stalls()
    else:
        messagebox.showinfo("INVALID !!!", 'Select a valid date/time')
```


Canteen System

Select date/time from the dropdown boxes:
(If no date is given, it will be assumed to be today)

Date: 30 / 2 / 2020
Time: 13 : 03

INVALID !!!

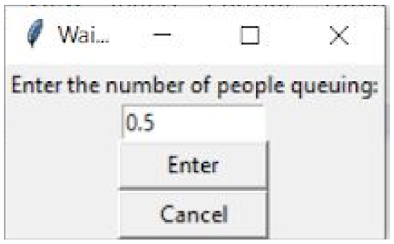
Select a valid date

OK

An additional feature of the *check* function is that it allows users to get a list of stalls even if they key in only the time with no date. This accounts for the case where the user wants to know what will open at a different time, such as later in the day. In this scenario, the date is assumed to be the current date.

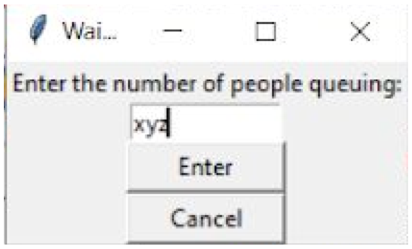
```
if valid_date:
    if self.year_entry.get()=="" and self.month_entry.get()=="" and self.day_entry.get()=="":
        input_time = datetime.time(int(self.hour_entry.get()), int(self.minute_entry.get()))
        self.no_input_stalls_without_day()
    else:
        input_day = input_date.weekday()
        self.display_other_day_stalls()
else:
    try:
        if integer_year < 1950:
            messagebox.showinfo("INVALID !!!", 'Select a valid year')
    except:
        messagebox.showinfo("INVALID !!!", "Select a valid date")
```

We also designed the *calc_waiting_time* function to check that the user enters an integer. Failure to input a positive integer will result in the program always returning an error.



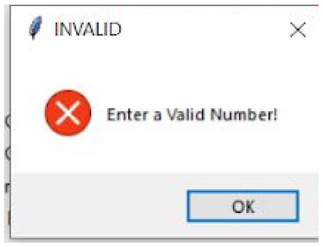
Enter the number of people queuing:

0.5

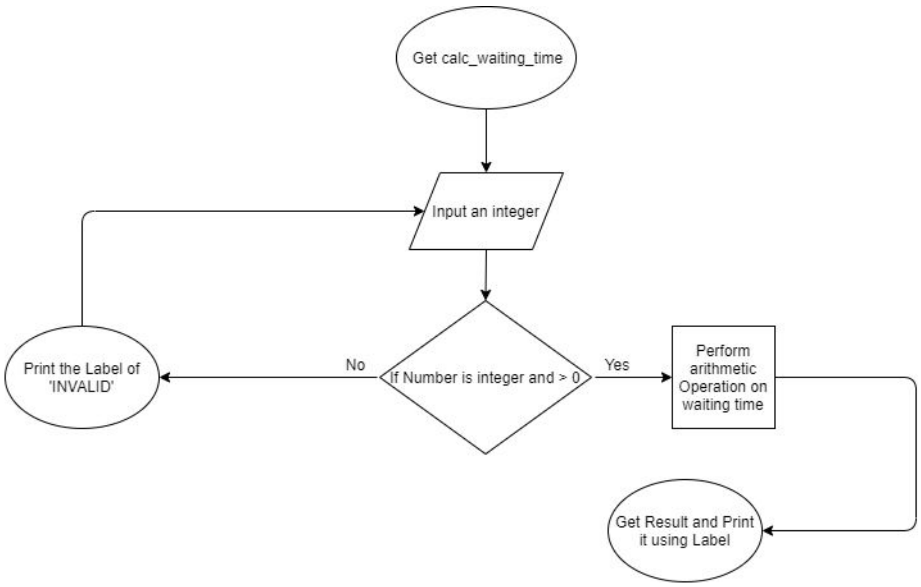


Enter the number of people queuing:

xyz



Enter a Valid Number!


Get calc_waiting_time
Input an integer
If Number is integer and > 0
No
Yes
Print the Label of 'INVALID'
Perform arithmetic Operation on waiting time
Get Result and Print it using Label

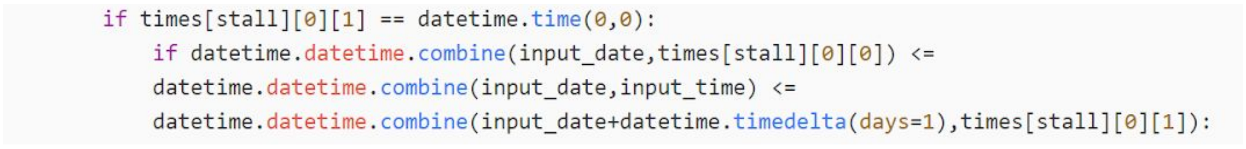
```python
def calc_waiting_time(self, box, label, entry, button, quit_box):
    number = entry.get()
    if number.isnumeric() and 0 <= int(number):
        label.destroy()
        entry.destroy()
        button.destroy()
        quit_box.destroy()
        Label(box, text="Estimated Waiting time : " + str(2 * (int(number) + 1)) + " minutes").pack()
        Button(box, text="OK", command=box.destroy).pack()
    else:
        messagebox.showerror("INVALID", "Enter a Valid Number!")
```

REFLECTIONS

One major difficulty that we faced was the creation of a universal back button. As the back button was designed to work with the navigation frames and stall frames, we had to account for the differences between each of the different frames. As such, we needed to be able to keep track of the transitions from frame to frame. Designing the top-level flowchart allowed us to stay organised and ensured that we were aware of how to move between frames.

Using tkinter as our GUI framework meant that we would have to learn how to deal with frames. The most common way to implement the GUI is through object-oriented programming. We often made mistakes because it was confusing using global and local variables within the same scope. This led to multiple errors when creating our frames and their methods. However, it allowed us to make our code more readable and concise.

The project allowed us to be creative. As McDonald's closes at midnight, we had to find a way to ensure that the program would not mistake the closing time to be 0000hrs of the same day that it opened. By researching the datetime library documentation, we were able to apply a *timedelta* of 1 day to ensure that the program would rollover to the next day, thus allowing the program to run as expected.



```
        if times[stall][0][1] == datetime.time(0,0):
            if datetime.datetime.combine(input_date,times[stall][0][0]) <=
            datetime.datetime.combine(input_date,input_time) <=
            datetime.datetime.combine(input_date+datetime.timedelta(days=1),times[stall][0][1]):
```

This showed us that while libraries exist to make it easier for programmers as built-in functions are present, we still had to learn how to use the built-in functions and apply them to the context of our problem to generate a solution.

We faced some difficulties when trying to put the program together. Each individual was responsible for coding up different features of the program. However, due to the varying styles and habits that each of us has when coding, the resultant code fragments were very different. As such, it took a long time for us to put the code together and ensure that all features could run as expected without breaking another part of the code. If we agreed on coding styles and conventions for naming variables and functions beforehand, it would have been easier and made the code more cohesive.

EXTERNAL LIBRARIES

The external libraries that we used are the following: 1) tkinter, 2) pickle, 3)datetime, 4) PIL, 5) os.

REFERENCES

Code snippets are from main.py, menupickler.py, timepickler.py respectively