DSA3101 Modelling Occupancy at NUS Central Library: Technical Handover Documentation

Overview of the Problem

The target audience of our model is the operations staff at NUS Central Library (CLB).

From our meetings with the staff, we identified that the staff aim to maximise utilisation and minimise resource consumption of the library. Ideally, utilisation should scale linearly with occupancy.

To address these needs, our model aims to help staff to

1. simulate level occupancy.
2. understand user behaviour between different periods such as exam period and non-exam period.

and how these factors affect the occupancy and thus utilisation in the library.

Description of source codes on Github repository

main branch:

- EDA – holds the Jupyter notebooks used to perform exploratory data analysis (EDA) on different datasets. More information about these datasets will be provided in a later section.
    - *EDA-dsa_data.ipynb*: EDA on dataset containing gantry data across different weeks (dsa_data.csv)
    - *EDA-firstdataset.ipynb*: EDA on datasets containing gantry data for one day (20230413_clb_taps.csv) and description of the gantries in CLB (cl_gates.xlsx)
    - *EDA-survey_data.ipynb:* Overall EDA on dataset containing responses from survey conducted (library_user_behaviour_survey_responses.csv)
    - *EDA-survey_data_by_levels.ipynb***:** Specific EDA for user level preference on survey dataset
    - *occupancy_across_whole_library.ipynb:* Specific EDA on gantry dataset to find out average occupancy and entry times
- dash – contains .py files for front-end.
    - *assets:* folder containing .png files for front-end.
    - *compare_Dockerfile:* Dockerfile for compare_page.py
    - *compare_page.py:* front-end code for comparison page
    - *floor_plan.py:* front-end code for floor plan page
    - *floorplan_Dockerfile:* Dockerfile for floor_plan.py
    - *main_Dockerfile:* Dockerfile for main_page.py
    - *main_page.py:* front-end code for landing page
    - *simulation_page.py:* front-end code for simulation model page, which links to the back-end simulation model.
    - *summary_page.py*: front-end code for summary page
- library_model – contains .py files for simulation model for back-end.
    - *clb_floor_plan*: folder containing .png and .geojson files for floorplans for the library
    - *results:* folder containing .csv files for results collected from running the simulation model.
    - *agents.py:* back-end code containing agents' classes for model
    - *flask_Dockerfile:* Dockerfile for flask_endpoints.py
    - *flask_endpoints.py*: back-end code to link model and return output to comparison page on the front-end.
    - *model.py:* code containing model class and functions for model datacollector
    - *model_Dockerfile:* Dockerfile for model.py
    - *multiple_runs.py*: code to run the model multiple times with same parameters to get the average and confidence intervals.
    - *run.py:* code to launch model onto server.
    - *server.py:* code to visualise the model on the server and to take in user inputs.

    - *simulation_results_analysis.ipynb:* Jupyter notebook used to perform analysis on model simulation results
    - *space.py:* code containing floorplan class to control space of the model.
- old_model – contains .py files for version 1 of simulation model.
- docker-compose.yml – for dockerisation of the front-end and back-end

Data Cleaning and Analysis

We cleaned and performed analysis on the following datasets:

1. *20230413_clb_taps.csv* - datasets containing gantry data for one day
2. *cl_gates.xlsx* - description of the gantries in CLB
3. *dsa_data.csv* - dataset containing gantry data of multiple dates across different weeks
4. *library_user_behaviour_survey_responses.csv* - dataset containing responses from survey conducted to understand more about user behaviour

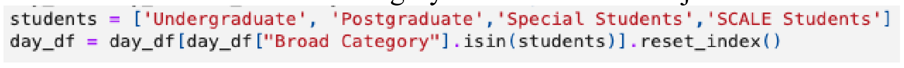
Cleaning and analysis for the first two datasets were preliminary and trivial as the datasets were very small. The short analysis done can be found in *EDA-firstdataset.ipynb.*

I will mainly focus on the later datasets: *dsa_data.csv* and *library_user_behaviour_survey_responses.csv*, as the analysis from these two datasets were incorporated into the simulation model later.

<u>*Dataset: dsa_data.csv*</u>

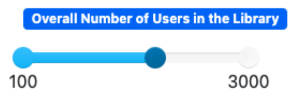
This dataset was provided by the library staff and had the following fields: *(Datetime, User Number, Broad Category, Library, Method, Direction)*.

Data Cleaning:
- Conversion to SG time
- Split Datetime column by Date, Hour, Time to create three more columns.
- Split datasets into respective time periods (week 3, recess week, reading week, exam week 1) based on the Date column.
- Narrowed down the "Broad Category" column to include just students.



```
students = ['Undergraduate', 'Postgraduate','Special Students','SCALE Students']
day_df = day_df[day_df["Broad Category"].isin(students)].reset_index()
```

Data Analysis:
- Number of unique visitors per day in different time periods. (*EDA-dsa_data.ipynb)*
    - Purpose: To determine allowable range for user input for number of users in library in simulation model.



- Number of entries and exits hour by hour in different time periods. (*EDA-dsa_data.ipynb & occupancy_across_whole_library.ipynb)*
    - Purpose: To determine distribution of arrival time of library users based on timestamp of a unique user (determined by "User Number" column).
    - Analysed that the library was open 24 hours during reading week and exam week 1.
- Average time spent in library per student in different time periods. (*EDA-dsa_data.ipynb)*
    - Purpose: To determine distribution of length of time based on the difference between the entries and exits of a unique user.

For our analysis, we combined week 3 and recess week to form the non-exam period and reading week and exam week 1 to form the exam-period, to streamline determining the user behaviour later in the model.

<u>*Dataset: library_user_behaviour_survey_responses.csv*</u>

This dataset consists of the survey responses of 260+ library users collected over the span of 1 week. We conducted this survey as we realised *dsa_data.csv* was unable to provide information on user behaviour of our model.

The survey had questions to enquire about user behaviours such as level preference, seat preference and seat hogging behaviour. More details about the specific questions asked can be found in *EDA-survey_data.ipynb,* where the column names of the dataset is the questions.

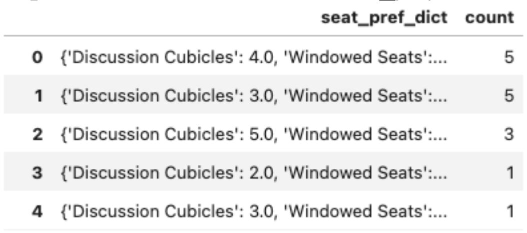
The analysis for this dataset is split across two ipynb files as we split work across group members.

Data Cleaning and Data Analysis:

- Analysis on level preference (*EDA-survey_data_by_levels.ipynb)*
  - Cleaning of the column names related to level preference and standardising into numerical format.
  - Purpose: To determine distribution of user's level preference, which is then returned as a dictionary.

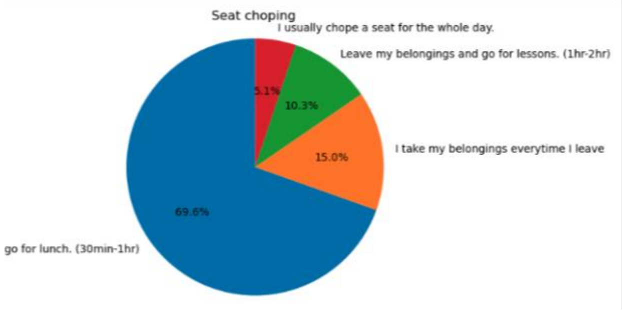
```
{'Level 3 Preference': 70, 'Level 4 Preference': 122, 'Level 5 Preference': 103, 'Level 6 CLB Pr
eference': 117, 'Level 6 Chinese Lib Preference': 69}
```

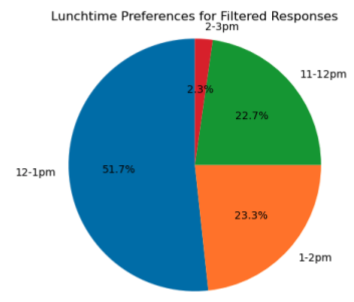
- Analysis on seat preference *(EDA-survey_data.ipynb)*
  - Cleaning of columns related to seat preference.
  - Analysing average seat preference for different seat types across the levels.
  - Extract seat preference scores and frequencies into a dictionary, which is then exported out into a csv file: *seat_pref.csv*



	seat_pref_dict	count
0	{'Discussion Cubicles': 4.0, 'Windowed Seats':...	5
1	{'Discussion Cubicles': 3.0, 'Windowed Seats':...	5
2	{'Discussion Cubicles': 5.0, 'Windowed Seats':...	3
3	{'Discussion Cubicles': 2.0, 'Windowed Seats':...	1
4	{'Discussion Cubicles': 3.0, 'Windowed Seats':...	1

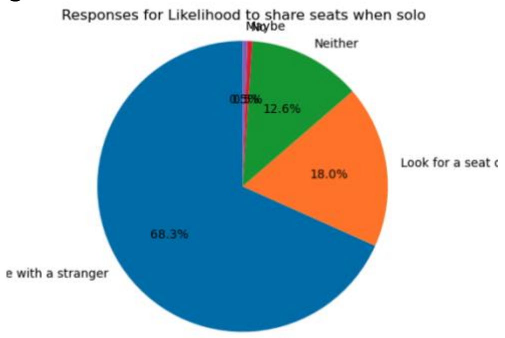
  - Purpose: To determine distribution of user's seat preference.
- Analysis on seat hogging *(EDA-survey_data.ipynb)*
  - Perform value_counts on columns related to seat hogging.
  - Purpose: To determine length of time users seat hog for and the time of day that they tend to seat hog (such as lunch time), which is exported out into two separate dictionaries with their frequencies.





- Analysis on willingness to share seats *(EDA-survey_data.ipynb)*
  - Purpose: Max capacity of the areas in the library will be adjusted in the model according to the willingness of users to share seats. This means an area has a lower

effective max capacity than the number of seats as some users may be unwilling to share tables with strangers.



- The effective max capacity is calculating using the formula below to get a value of 0.8415 based on the distribution above.
  (E.g. if the area has 100 seats, the effective max capacity is 84 seats)

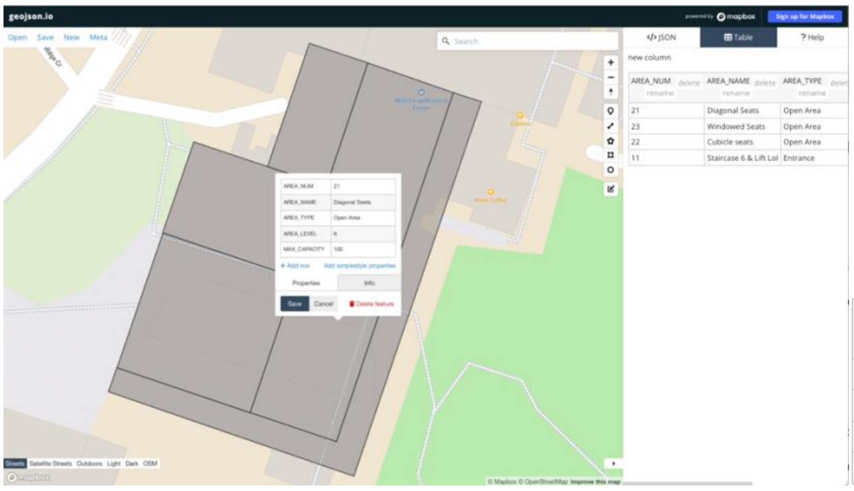
```
percentage_willing_to_share = 0.683
percentage_not_willing_to_share = 0.317

# Calculate capacity
capacity = (percentage_willing_to_share * 1.0) + (percentage_not_willing_to_share * 0.5)
```

Library Simulation Model

The library simulation model is an agent-based model that utilised the mesa_geo library in python. The mesa_geo package is the GIS extension for Mesa agent-based modelling, which allows us to take in the floor plans of CLB as geojson files.

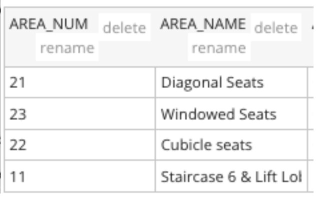
The structure of code for the library simulation model is adapted from the geo_schelling_points mesa_geo model: https://github.com/projectmesa/mesa-examples/tree/main/gis/geo_schelling_points

Before constructing the python scripts for the model, we first converted the images of the CLB floor plans into geojson files. We did this using https://geojson.io/. We replicated the areas in the floorplans by manually drawing it out on the map. We then added features to each area: AREA_NUM, AREA_NAME, AREA_TYPE, AREA_LEVEL and MAX_CAPACITY. These features will be used later in our model. For AREA_NUM, entry points, such as staircase and lift area, start with 1 and open study areas start with 2. AREA_NAME for open study areas correspond to their seat types. This is important for distinguishing the area types in our model later.
We then exported it out as a geojson file and saved it in the *clb_floor_plan* folder.



AREA_NUM	21
AREA_NAME	Diagonal Seats
AREA_TYPE	Open Area
AREA_LEVEL	6
MAX_CAPACITY	100

AREA_NUM	AREA_NAME	AREA_TYPE
21	Diagonal Seats	Open Area
23	Windowed Seats	Open Area
22	Cubicle seats	Open Area
11	Staircase 6 & Lift Lol	Entrance



AREA_NUM	AREA_NAME
21	Diagonal Seats
23	Windowed Seats
22	Cubicle seats
11	Staircase 6 & Lift Lol

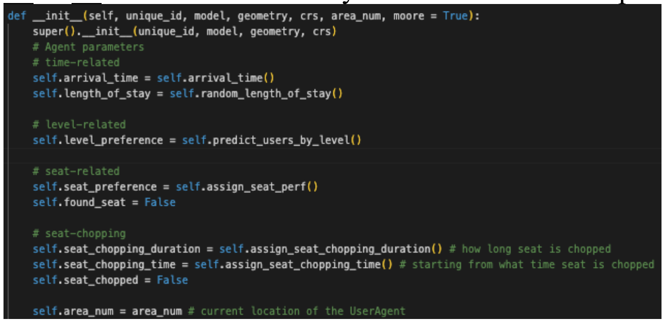
The model itself consists of several python scripts:

agents.py

agents.py contains the mesa GeoAgent classes UserAgent and AreaAgent.

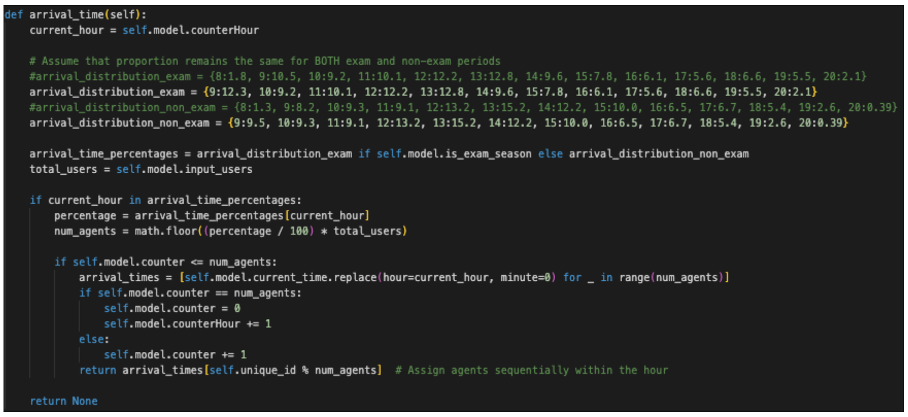
Class UserAgent is used to model the library user. One instance of the class represents one library user in the model. The UserAgent class has the following functions:

- *__init__*: to initialise a library user with attributes representing user behaviours.



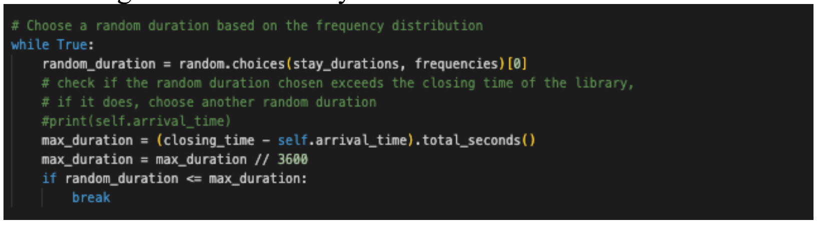
```python
def __init__(self, unique_id, model, geometry, crs, area_num, moore = True):
    super().__init__(unique_id, model, geometry, crs)
    # Agent parameters
    # time-related
    self.arrival_time = self.arrival_time()
    self.length_of_stay = self.random_length_of_stay()

    # level-related
    self.level_preference = self.predict_users_by_level()

    # seat-related
    self.seat_preference = self.assign_seat_perf()
    self.found_seat = False

    # seat-chopping
    self.seat_chopping_duration = self.assign_seat_chopping_duration() # how long seat is chopped
    self.seat_chopping_time = self.assign_seat_chopping_time() # starting from what time seat is chopped
    self.seat_chopped = False

    self.area_num = area_num # current location of the UserAgent
```

- *arrival_time:* to assign an arrival time to the agent that is following the distribution found in the data analysis earlier. The distribution is converted into a dictionary that is inserted into the function. We have two dictionaries, exam period and non-exam period, as the distribution of arrival times differ.



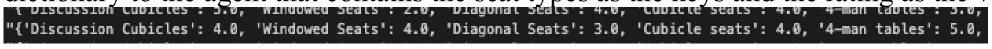
```python
def arrival_time(self):
    current_hour = self.model.counterHour

    # Assume that proportion remains the same for BOTH exam and non-exam periods
    #arrival_distribution_exam = {8:1.8, 9:10.5, 10:9.2, 11:10.1, 12:12.2, 13:12.8, 14:9.6, 15:7.8, 16:6.1, 17:5.6, 18:6.6, 19:5.5, 20:2.1}
    arrival_distribution_exam = {9:12.3, 10:9.2, 11:10.1, 12:12.2, 13:12.8, 14:9.6, 15:7.8, 16:6.1, 17:5.6, 18:6.6, 19:5.5, 20:2.1}
    #arrival_distribution_non_exam = {8:1.3, 9:8.2, 10:9.3, 11:9.1, 12:13.2, 13:15.2, 14:12.2, 15:10.0, 16:6.5, 17:6.7, 18:5.4, 19:2.6, 20:0.39}
    arrival_distribution_non_exam = {9:9.5, 10:9.3, 11:9.1, 12:13.2, 13:15.2, 14:12.2, 15:10.0, 16:6.5, 17:6.7, 18:5.4, 19:2.6, 20:0.39}

    arrival_time_percentages = arrival_distribution_exam if self.model.is_exam_season else arrival_distribution_non_exam
    total_users = self.model.input_users

    if current_hour in arrival_time_percentages:
        percentage = arrival_time_percentages[current_hour]
        num_agents = math.floor((percentage / 100) * total_users)

        if self.model.counter <= num_agents:
            arrival_times = [self.model.current_time.replace(hour=current_hour, minute=0) for _ in range(num_agents)]
            if self.model.counter == num_agents:
                self.model.counter = 0
                self.model.counterHour += 1
            else:
                self.model.counter += 1
            return arrival_times[self.unique_id % num_agents]  # Assign agents sequentially within the hour

    return None
```

- *random_length_of_stay:* to assign length of stay to the agent. The function is similar to arrival_time. However, there is an additional check to ensure that the user does not stay past the closing time of the library.



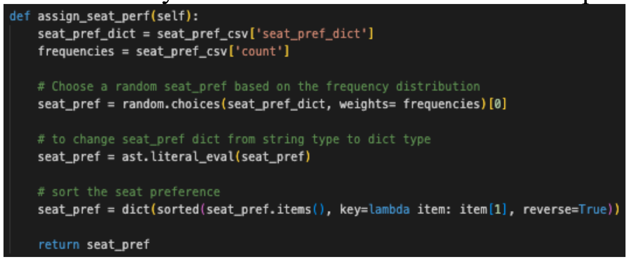
```python
# Choose a random duration based on the frequency distribution
while True:
    random_duration = random.choices(stay_durations, frequencies)[0]
    # check if the random duration chosen exceeds the closing time of the library,
    # if it does, choose another random duration
    #print(self.arrival_time)
    max_duration = (closing_time - self.arrival_time).total_seconds()
    max_duration = max_duration // 3600
    if random_duration <= max_duration:
        break
```

- *predict_users_by_level:* to assign level preference to the agent. Function is similar to arrival_time.
- *assign_seat_pref*: to assign seat preference to the agent. This function utilises the dataset *seat_pref.csv*, which is from the data analysis of survey data. The function assigns a dictionary to the agent that contains the seat types as the keys and the rating as the value.



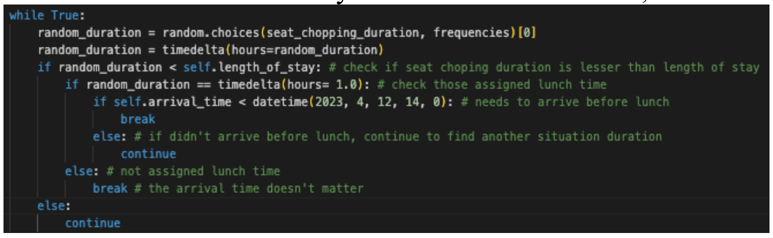
```
"{'Discussion Cubicles': 5.0, 'Windowed Seats': 2.0, 'Diagonal Seats': 4.0, 'Cubicle seats': 4.0, '4-man tables': 5.0,
"{'Discussion Cubicles': 4.0, 'Windowed Seats': 4.0, 'Diagonal Seats': 3.0, 'Cubicle seats': 4.0, '4-man tables': 5.0,
```

The dictionary is then sorted so that the first seat preference is the first key.



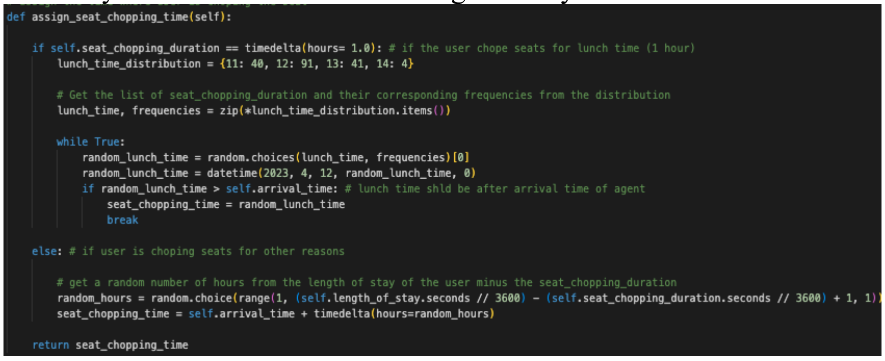
```python
def assign_seat_perf(self):
    seat_pref_dict = seat_pref_csv['seat_pref_dict']
    frequencies = seat_pref_csv['count']

    # Choose a random seat_pref based on the frequency distribution
    seat_pref = random.choices(seat_pref_dict, weights= frequencies)[0]

    # to change seat_pref dict from string type to dict type
    seat_pref = ast.literal_eval(seat_pref)

    # sort the seat preference
    seat_pref = dict(sorted(seat_pref.items(), key=lambda item: item[1], reverse=True))

    return seat_pref
```

- *assign_seat_chopping_duration*: to assign seat hogging duration to agent. Function is similar to arrival_time. There is an additional check to ensure the duration is lesser than agent's length of stay. Agents assigned duration of 1 hour are taken to be seat hogging for lunch, thus there is a check to ensure they arrive before lunch time, which ranges from 11am to 2pm.



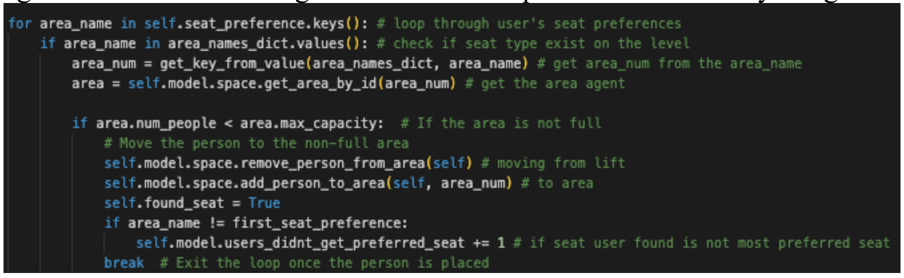
```python
while True:
    random_duration = random.choices(seat_chopping_duration, frequencies)[0]
    random_duration = timedelta(hours=random_duration)
    if random_duration < self.length_of_stay: # check if seat choping duration is lesser than length of stay
        if random_duration == timedelta(hours= 1.0): # check those assigned lunch time
            if self.arrival_time < datetime(2023, 4, 12, 14, 0): # needs to arrive before lunch
                break
            else: # if didn't arrive before lunch, continue to find another situation duration
                continue
        else: # not assigned lunch time
            break # the arrival time doesn't matter
    else:
        continue
```

- *assign_seat_chopping_time*: to assign seat hogging timing to agent. Function is similar to arrival_time. If agent's seat hogging duration is 1 hour, the seat hogging timing is taken from the lunch time distribution obtained from data analysis of the survey data above. Else, it is randomly chosen within the user's length of stay.



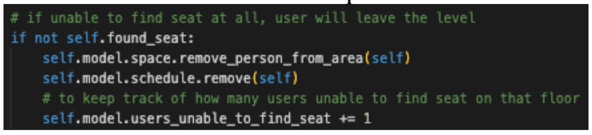
```python
def assign_seat_chopping_time(self):

    if self.seat_chopping_duration == timedelta(hours= 1.0): # if the user chope seats for lunch time (1 hour)
        lunch_time_distribution = {11: 40, 12: 91, 13: 41, 14: 4}

        # Get the list of seat_chopping_duration and their corresponding frequencies from the distribution
        lunch_time, frequencies = zip(*lunch_time_distribution.items())

        while True:
            random_lunch_time = random.choices(lunch_time, frequencies)[0]
            random_lunch_time = datetime(2023, 4, 12, random_lunch_time, 0)
            if random_lunch_time > self.arrival_time: # lunch time shld be after arrival time of agent
                seat_chopping_time = random_lunch_time
                break

    else: # if user is choping seats for other reasons

        # get a random number of hours from the length of stay of the user minus the seat_chopping_duration
        random_hours = random.choice(range(1, (self.length_of_stay.seconds // 3600) - (self.seat_chopping_duration.seconds // 3600) + 1, 1))
        seat_chopping_time = self.arrival_time + timedelta(hours=random_hours)

    return seat_chopping_time
```

- *find_seat:* to simulate the behaviour of users finding a seat in the library after arriving. The agent finds seat according to the sorted seat preference dictionary assigned to them.



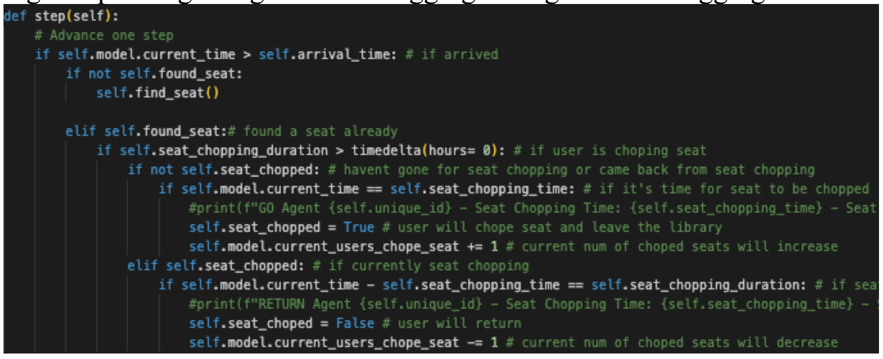
```python
for area_name in self.seat_preference.keys(): # loop through user's seat preferences
    if area_name in area_names_dict.values(): # check if seat type exist on the level
        area_num = get_key_from_value(area_names_dict, area_name) # get area_num from the area_name
        area = self.model.space.get_area_by_id(area_num) # get the area agent

        if area.num_people < area.max_capacity:  # If the area is not full
            # Move the person to the non-full area
            self.model.space.remove_person_from_area(self) # moving from lift
            self.model.space.add_person_to_area(self, area_num) # to area
            self.found_seat = True
            if area_name != first_seat_preference:
                self.model.users_didnt_get_preferred_seat += 1 # if seat user found is not most preferred seat
            break  # Exit the loop once the person is placed
```

If they are unable to find a seat, by the model's logic, they will leave the level and therefore will be removed from the map.



```python
# if unable to find seat at all, user will leave the level
if not self.found_seat:
    self.model.space.remove_person_from_area(self)
    self.model.schedule.remove(self)
    # to keep track of how many users unable to find seat on that floor
    self.model.users_unable_to_find_seat += 1
```

- *step:* to determine what the agent does at each step. If the agent hasn't found a seat, the find_seat function will be called. If they have found a seat already, implement seat hogging logic depending on agent's seat hogging timing and seat hogging duration.



```
def step(self):
    # Advance one step
    if self.model.current_time > self.arrival_time: # if arrived
        if not self.found_seat:
            self.find_seat()

        elif self.found_seat:# found a seat already
            if self.seat_chopping_duration > timedelta(hours= 0): # if user is choping seat
                if not self.seat_chopped: # havent gone for seat chopping or came back from seat chopping
                    if self.model.current_time == self.seat_chopping_time: # if it's time for seat to be chopped
                        #print(f"GO Agent {self.unique_id} - Seat Chopping Time: {self.seat_chopping_time} - Seat
                        self.seat_chopped = True # user will chope seat and leave the library
                        self.model.current_users_chope_seat += 1 # current num of choped seats will increase
                elif self.seat_chopped: # if currently seat chopping
                    if self.model.current_time - self.seat_chopping_time == self.seat_chopping_duration: # if sea
                        #print(f"RETURN Agent {self.unique_id} - Seat Chopping Time: {self.seat_chopping_time} -
                        self.seat_choped = False # user will return
                        self.model.current_users_chope_seat -= 1 # current num of choped seats will decrease
```

Class AreaAgent is used to model the areas in the library that are determined by the geojson file created earlier. One instance of the class represents one area in the model. The AreaAgent class has the following functions:

- *__init__:* to initialise an area with the following attributes
  - num_people: number of people in the area
  - max_capacity: number of seats in the area
- *change_max_capacity:* to update max capacity of area based on user input and willingness to share seats
- *random_point*: to get a random point which is subsequently used in add_person_to_area function in space.py
- *add_person:* to update num_people by +1 when a UserAgent is added to the area
- *remove_person*: to update num_people by -1 when UserAgent is removed from the area

space.py

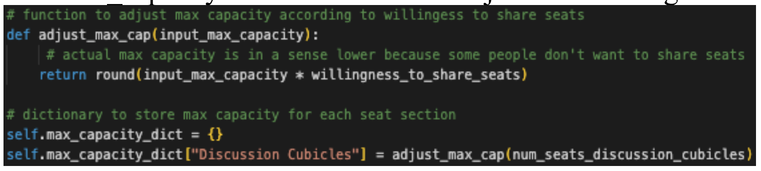
space.py contains the GeoSpace class FloorPlan which interacts with the AreaAgent class from agents.py. The FloorPlan class has the following key functions:

- *__init__:* to initialise the floorplan with attributes:
  - _id_area_map: dictionary that contains the AREA_NUM as key and AreaAgent instances corresponding to the AREA_NUM as values.
- *add_areas:* add AreaAgents to _id_area_map
- *add_person_to_area:* add a UserAgent to an area (AreaAgent). It utilises the *random_point* function under AreaAgent class in agents.py to find a random point for the UserAgent to move to in the area.
- *remove_person_from_area:* remove a UserAgent from an area (AreaAgent)
- other functions in this class are mainly helper functions for the other scripts in the model

model.py

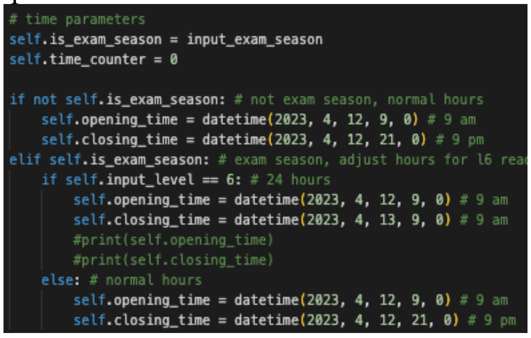
model.py contains the Model class LibraryModel which interacts with the classes in agents.py and space.py. The Model class has the following key functions:

- *__init__:* to initialise the model with user inputs.
  The max_capacity of each area is also adjusted according to willingness to share seats.



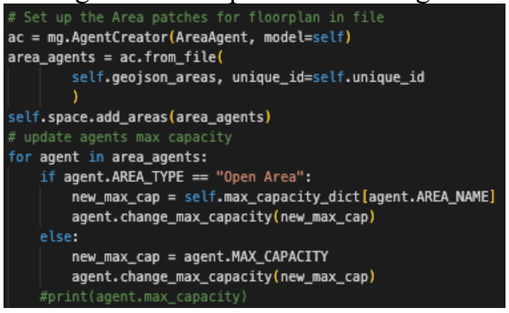
```
# function to adjust max capacity according to willingess to share seats
def adjust_max_cap(input_max_capacity):
    # actual max capacity is in a sense lower because some people don't want to share seats
    return round(input_max_capacity * willingness_to_share_seats)

# dictionary to store max capacity for each seat section
self.max_capacity_dict = {}
self.max_capacity_dict["Discussion Cubicles"] = adjust_max_cap(num_seats_discussion_cubicles)
```

The time parameters of the model are adjusted according to the opening time and closing time of the library based on exam period and non-exam period. During exam period, level 6 is open for 24 hours.



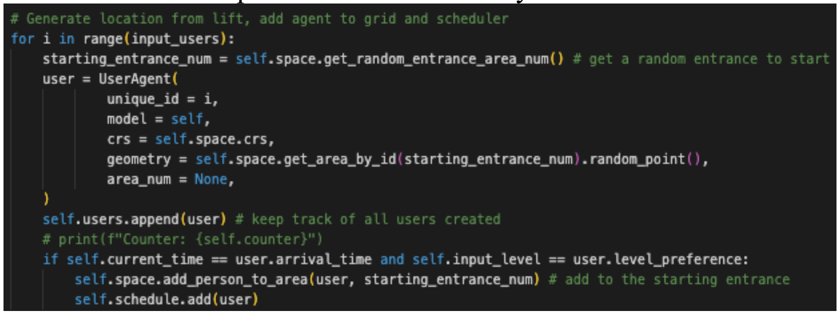
```
# time parameters
self.is_exam_season = input_exam_season
self.time_counter = 0

if not self.is_exam_season: # not exam season, normal hours
    self.opening_time = datetime(2023, 4, 12, 9, 0) # 9 am
    self.closing_time = datetime(2023, 4, 12, 21, 0) # 9 pm
elif self.is_exam_season: # exam season, adjust hours for l6 rea
    if self.input_level == 6: # 24 hours
        self.opening_time = datetime(2023, 4, 12, 9, 0) # 9 am
        self.closing_time = datetime(2023, 4, 13, 9, 0) # 9 am
        #print(self.opening_time)
        #print(self.closing_time)
    else: # normal hours
        self.opening_time = datetime(2023, 4, 12, 9, 0) # 9 am
        self.closing_time = datetime(2023, 4, 12, 21, 0) # 9 pm
```

The AreaAgents are also initialised here from the geojson files. Max capacity of each AreaAgent is also updated according to user input.



```
# Set up the Area patches for floorplan in file
ac = mg.AgentCreator(AreaAgent, model=self)
area_agents = ac.from_file(
        self.geojson_areas, unique_id=self.unique_id
        )
self.space.add_areas(area_agents)
# update agents max capacity
for agent in area_agents:
    if agent.AREA_TYPE == "Open Area":
        new_max_cap = self.max_capacity_dict[agent.AREA_NAME]
        agent.change_max_capacity(new_max_cap)
    else:
        new_max_cap = agent.MAX_CAPACITY
        agent.change_max_capacity(new_max_cap)
    #print(agent.max_capacity)
```

The UserAgents are then initialised, with their initial location in the FloorPlan to be an entry point, such as the staircase and lift area. UserAgents are then added to the map based on their assigned arrival_time and level_preference. The number of UserAgents to be generated is based on the user input for number of library users.



```
# Generate location from lift, add agent to grid and scheduler
for i in range(input_users):
    starting_entrance_num = self.space.get_random_entrance_area_num() # get a random entrance to start
    user = UserAgent(
            unique_id = i,
            model = self,
            crs = self.space.crs,
            geometry = self.space.get_area_by_id(starting_entrance_num).random_point(),
            area_num = None,
    )
    self.users.append(user) # keep track of all users created
    # print(f"Counter: {self.counter}")
    if self.current_time == user.arrival_time and self.input_level == user.level_preference:
        self.space.add_person_to_area(user, starting_entrance_num) # add to the starting entrance
        self.schedule.add(user)
```

- *get_current_time*: to update current_time of the model to increase by 30 minutes for each step of the model until the current_time reaches the closing time.
- *step:* to run one step of the model. At each step, UserAgents are added and removed from the map based on their assigned arrival_time and length_of_stay in reference to the current_time of the model.
  The step function also stops the model once the time has reached the closing_time of the library.

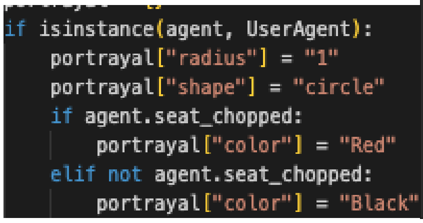
__server.py & run.py__

server.py, along with run.py, contains the code to visualise the model and launch it on a server.

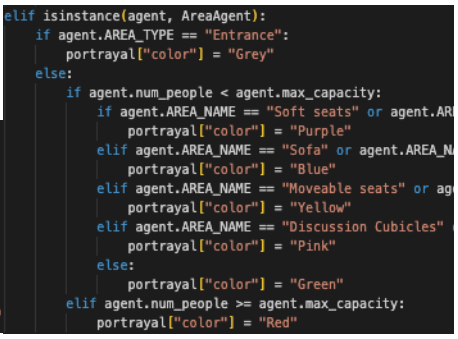
server.py contains the following visualisation elements:

- *ClockElement*: TextElement class to display current_time of model at each step.
- *SeatColourLegend*: TextElement class to display the legend showing the seat types and their corresponding colours.
- *agent_portrayal:* function to display the agents based on their attributes.
  If an UserAgent is seat hogging, they are turned red.

The colours of the AreaAgents differs depending on their AREA_TYPE, AREA_NAME and max_capacity. Entry points are grey. A seating area that hit max capacity turns red.

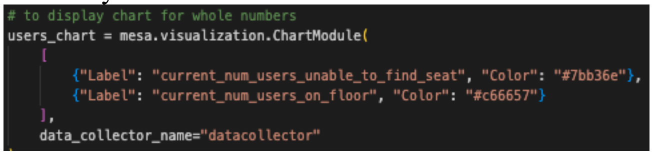


```python
if isinstance(agent, UserAgent):
    portrayal["radius"] = "1"
    portrayal["shape"] = "circle"
    if agent.seat_chopped:
        portrayal["color"] = "Red"
    elif not agent.seat_chopped:
        portrayal["color"] = "Black"
```



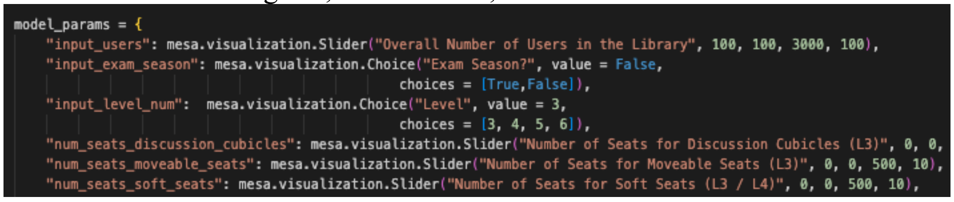
```python
elif isinstance(agent, AreaAgent):
    if agent.AREA_TYPE == "Entrance":
        portrayal["color"] = "Grey"
    else:
        if agent.num_people < agent.max_capacity:
            if agent.AREA_NAME == "Soft seats" or agent.AR
                portrayal["color"] = "Purple"
            elif agent.AREA_NAME == "Sofa" or agent.AREA_N
                portrayal["color"] = "Blue"
            elif agent.AREA_NAME == "Moveable seats" or ag
                portrayal["color"] = "Yellow"
            elif agent.AREA_NAME == "Discussion Cubicles"
                portrayal["color"] = "Pink"
            else:
                portrayal["color"] = "Green"
        elif agent.num_people >= agent.max_capacity:
            portrayal["color"] = "Red"
```

- *users_chart:* to visualise metrics consisting of whole numbers collected while running the model. The metrics to be visualised can be amended by changing the value for Label in the dictionary. This allows the staff to choose what metrics they would like to see in the chart.



```python
# to display chart for whole numbers
users_chart = mesa.visualization.ChartModule(
    [
        {"Label": "current_num_users_unable_to_find_seat", "Color": "#7bb36e"},
        {"Label": "current_num_users_on_floor", "Color": "#c66657"}
    ],
    data_collector_name="datacollector"
```

- *satisfaction_chart:* to visualise results consisting of percentages collected while running the model. We split this up from users_chart due to different scales.
- *map_element:* to visualise the map along with the agent visualisation from agent_portrayal.
- *model_params:* to allow the model to take in user input from the server. User inputs are related to number of agents, exam season, floor level and number of seats.



```python
model_params = {
    "input_users": mesa.visualization.Slider("Overall Number of Users in the Library", 100, 100, 3000, 100),
    "input_exam_season": mesa.visualization.Choice("Exam Season?", value = False,
                                                    choices = [True,False]),
    "input_level_num":  mesa.visualization.Choice("Level", value = 3,
                                                    choices = [3, 4, 5, 6]),
    "num_seats_discussion_cubicles": mesa.visualization.Slider("Number of Seats for Discussion Cubicles (L3)", 0, 0,
    "num_seats_moveable_seats": mesa.visualization.Slider("Number of Seats for Moveable Seats (L3)", 0, 0, 500, 10),
    "num_seats_soft_seats": mesa.visualization.Slider("Number of Seats for Soft Seats (L3 / L4)", 0, 0, 500, 10),
```

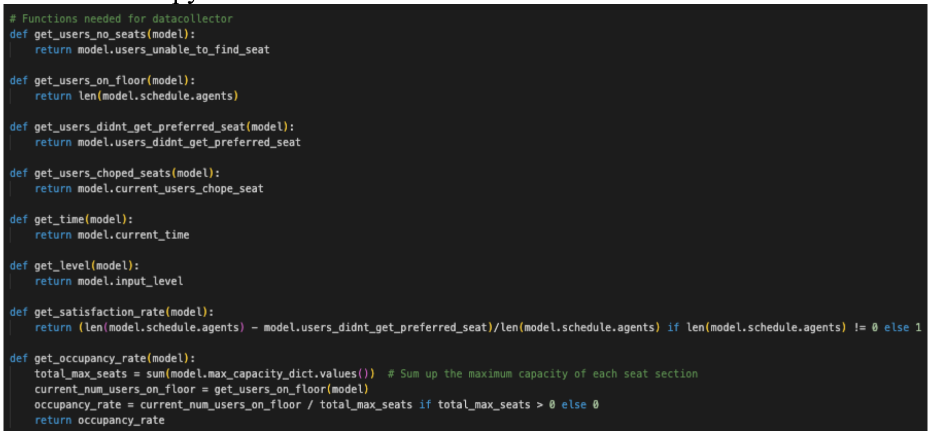
- *server*: to build the server to visualise the model with their various visualisation elements such as ClockElement, SeatColourLegend, users_chart and satisfaction chart.

run.py is then called to launch the server and make the simulation model accessible on the server.

Library Simulation Model – DataCollector and Metrics Collected

mesa.DataCollector allows us to collect metrics while running the model.

LibraryModel also has a special attribute: datacollector, which collects the metrics from running the model. model.py contains the functions for the datacollector as well.



```python
# Functions needed for datacollector
def get_users_no_seats(model):
    return model.users_unable_to_find_seat

def get_users_on_floor(model):
    return len(model.schedule.agents)

def get_users_didnt_get_preferred_seat(model):
    return model.users_didnt_get_preferred_seat

def get_users_choped_seats(model):
    return model.current_users_chope_seat

def get_time(model):
    return model.current_time

def get_level(model):
    return model.input_level

def get_satisfaction_rate(model):
    return (len(model.schedule.agents) - model.users_didnt_get_preferred_seat)/len(model.schedule.agents) if len(model.schedule.agents) != 0 else 1

def get_occupancy_rate(model):
    total_max_seats = sum(model.max_capacity_dict.values())  # Sum up the maximum capacity of each seat section
    current_num_users_on_floor = get_users_on_floor(model)
    occupancy_rate = current_num_users_on_floor / total_max_seats if total_max_seats > 0 else 0
    return occupancy_rate
```

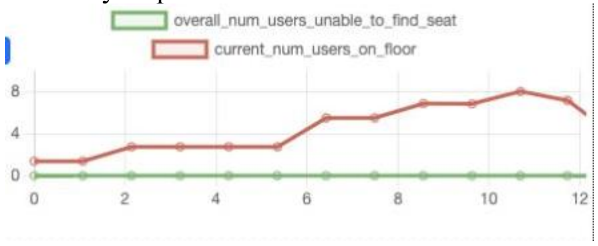
The metrics that are currently collected are shown below. The keys of the dictionary are the names of the metrics. The value is the function used to collect the metric. The function can be found above.

```
self.datacollector = mesa.DataCollector(
    model_reporters = {
        "time": get_time,
        "level": get_level,
        "current_num_users_on_floor": get_users_on_floor,
        "current_num_users_unable_to_find_seat": get_users_no_seats,
        "current_num_users_didnt_get_preferred_seat": get_users_didnt_get_preferred_seat,
        "current_num_seats_choped": get_users_choped_seats,
        "satisfaction_rate": get_satisfaction_rate,
        "occupancy_rate(level)": get_occupancy_rate,
    }
)
```

At each step of the model, the metrics are collected and inserted into a dataframe row by row.

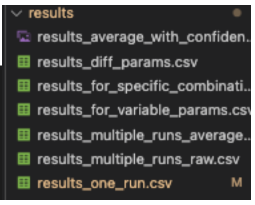
```
self.datacollector.collect(self)
```

The metrics are also visualised on the server using users_chart and satisfaction_chart in server.py. These charts will update with every step as the model runs.



Once the model reaches closing time and stops running, the dataframe is exported out into *results_one_run.csv* and saved in the results folder.

```
if self.current_time == self.closing_time:
    self.running = False
    results_df = self.datacollector.get_model_vars_dataframe()
    results_df.to_csv("results/results_one_run.csv", index = False)
```



```
results
    results_average_with_confiden...
    results_diff_params.csv
    results_for_specific_combinati...
    results_for_variable_params.csv
    results_multiple_runs_average...
    results_multiple_runs_raw.csv
    results_one_run.csv          M
```

This allows the library staff to access the csv file and perform their own analysis if they wish to do so.

<u>*multiple_runs.py*</u>

Due to the inherent error from the randomness of the model, multiple_runs.py runs the model multiple times with the same model parameters to get the average and confidence intervals of the metrics collected.

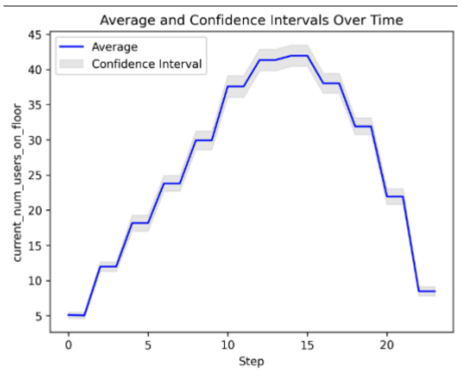
Note: multiple_runs.py runs independently from the other scripts and is considered as an additional script to run if the staff wishes to find out the average and confidence intervals. This script is also not incorporated in the front-end. This can be a consideration for future developments.

multiple_runs.py has the following user inputs:

```
# USER INPUTS
# constant parameters
params = {"input_level_num": 3,
          "input_users": 500,
            "input_max_capacity": 100,
            "input_exam_season": True}

# number of times to run the model
input_iterations = 50
# confidence level for finding average and confidence intervals
input_confidence_level = 0.90  # e.g. 0.9 is 90%
# metric that they want to find (e.g. current number of users on the floor, number of users unable to find seat)
input_metric = 'current_num_users_on_floor'
```

The inputs can be freely changed. Currently, the *params* input in the script is for an older version of the model and may not have all the recent updated model parameters.

The output of the script is exported out into *results_multiple_runs_averaged.csv,* which is in the results folder. The results are also plotted and the image is saved as *results_average_with_confidence_intervals.png.* Here is an example of the graph.

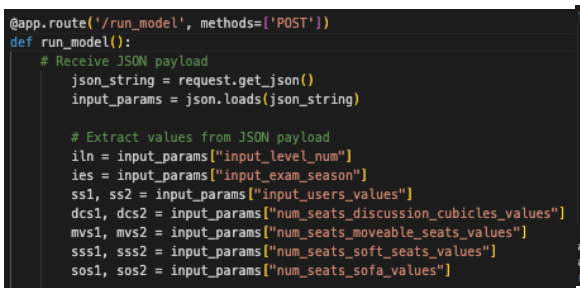


Library Simulation Model – Comparing between different simulations

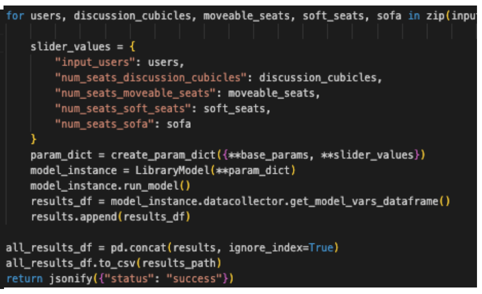
flask_endpoints.py

flask_endpoints.py links the model to the compare_page.py under the dash folder for the front-end. The script has the following key functions:

- *run_model:* it takes in the POST request from the compare_page.py, which contains the model parameters to compare. The model parameters are then processed and inserted into the model to run. The results collected are then saved to *results_for_specific_combinations.csv*. The csv file is then used to provide data visualisation of the comparison results on the front end.



```python
@app.route('/run_model', methods=['POST'])
def run_model():
    # Receive JSON payload
        json_string = request.get_json()
        input_params = json.loads(json_string)

        # Extract values from JSON payload
        iln = input_params["input_level_num"]
        ies = input_params["input_exam_season"]
        ss1, ss2 = input_params["input_users_values"]
        dcs1, dcs2 = input_params["num_seats_discussion_cubicles_values"]
        mvs1, mvs2 = input_params["num_seats_moveable_seats_values"]
        sss1, sss2 = input_params["num_seats_soft_seats_values"]
        sos1, sos2 = input_params["num_seats_sofa_values"]
```



```python
for users, discussion_cubicles, moveable_seats, soft_seats, sofa in zip(inpu

    slider_values = {
        "input_users": users,
        "num_seats_discussion_cubicles": discussion_cubicles,
        "num_seats_moveable_seats": moveable_seats,
        "num_seats_soft_seats": soft_seats,
        "num_seats_sofa": sofa
    }
    param_dict = create_param_dict({**base_params, **slider_values})
    model_instance = LibraryModel(**param_dict)
    model_instance.run_model()
    results_df = model_instance.datacollector.get_model_vars_dataframe()
    results.append(results_df)

all_results_df = pd.concat(results, ignore_index=True)
all_results_df.to_csv(results_path)
return jsonify({"status": "success"})
```

The run_model function is split into 4: one for each level to allow the comparison to run seamlessly.

Library Simulation Model – Other Attempted Models

The *old_model* folder contains the scripts for the original version of the library simulation model. It utilises the original mesa package.

It has the following scripts:

LibraryModel.py

This script contained the Agent class LibraryUser and Model class LibraryModel. The code implemented can change the number of library users across different time periods such as morning, afternoon and evening.

main.py

This script is similar to server.py and run.py. It contains the code to visualise the model.

However, the model had difficulty in visualising the floorplan of the library. Thus, we switched to using the GIS version of mesa, mesa_geo, and implemented the model in *library_model* folder

Overview of the Solution

The datacollector in the library simulation model collects and exports out useful metrics for the library staff to determine the optimal resource allocation in the library. The library staff can understand how the utilisation of the library will change when they decide to change resource allocation such as changing the number of seats of a certain seat type.

One useful metric is satisfaction rate which is defined as the number of users who get their preferred seat type divided by the total number of users on the floor. For example, during exam period, there may be lower satisfaction level as more users are unable to get their preferred seat type. The library staff can consider improving this by increasing the number of seats according to the preferred seat type of users. Before implementing the actual idea, they can first simulate the changes in number of seats using the simulation model to determine whether they will get the ideal increase in satisfaction rate.

Further developments of the model can also be implemented according to the changing needs of the library staff and the changing user behaviour of the library users.