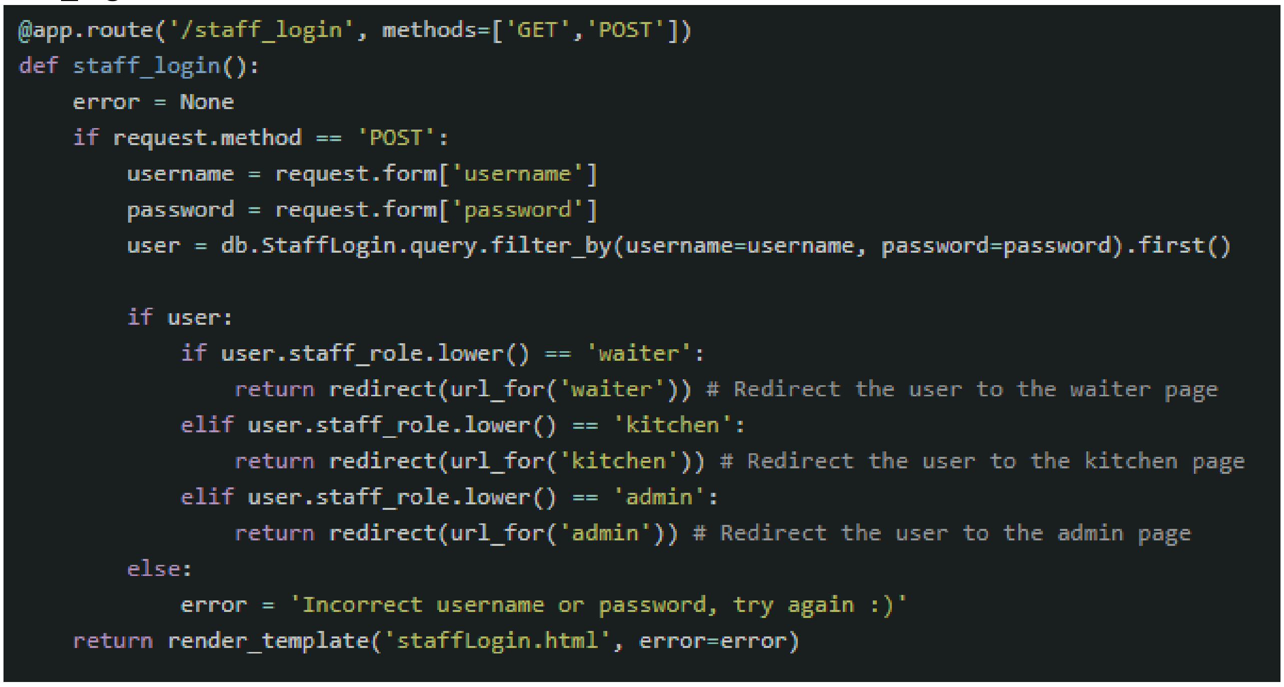
<u>Individual Project Report CS2810</u>

Name: Aryaman Rawat

Section 1:Code Snippets Description

staff_login method:

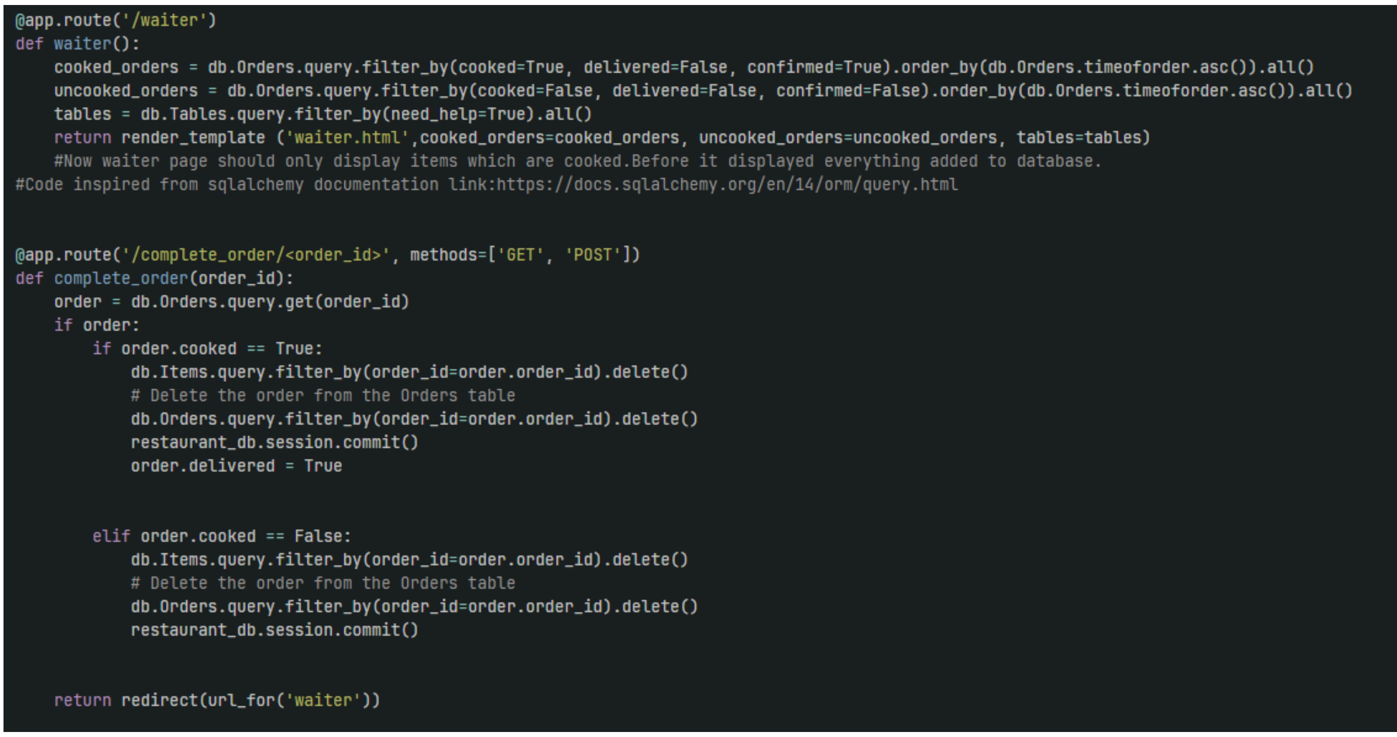


```python
@app.route('/staff_login', methods=['GET','POST'])
def staff_login():
    error = None
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        user = db.StaffLogin.query.filter_by(username=username, password=password).first()

        if user:
            if user.staff_role.lower() == 'waiter':
                return redirect(url_for('waiter')) # Redirect the user to the waiter page
            elif user.staff_role.lower() == 'kitchen':
                return redirect(url_for('kitchen')) # Redirect the user to the kitchen page
            elif user.staff_role.lower() == 'admin':
                return redirect(url_for('admin')) # Redirect the user to the admin page
        else:
            error = 'Incorrect username or password, try again :)'
    return render_template('staffLogin.html', error=error)
```

The above code defines a route for a staff login page and handles both GET and POST request.I found this quite ingenious as in the POST request, the code fetches the username and password from the form data, queries the database for a matching user with the given credentials, and redirects the user to the appropriate page based on the user's role.If credentials don't match error is displayed.For a GET request, the code renders the staffLogin.html template with an optional error message as context.As shown in the code the ‘error’ variable is initially set to ‘none‘ but during POST request error message is stored in ‘error’ variable.It is then passes as a context variable when rendering the page.

waiter/complete_order/confirm_order methods:

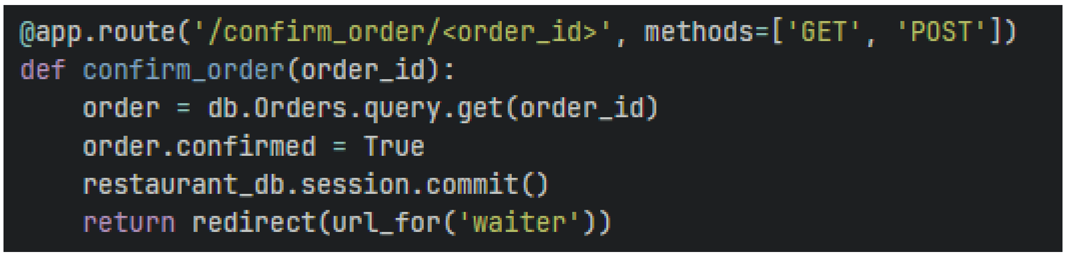


```
@app.route('/waiter')
def waiter():
    cooked_orders = db.Orders.query.filter_by(cooked=True, delivered=False, confirmed=True).order_by(db.Orders.timeoforder.asc()).all()
    uncooked_orders = db.Orders.query.filter_by(cooked=False, delivered=False, confirmed=False).order_by(db.Orders.timeoforder.asc()).all()
    tables = db.Tables.query.filter_by(need_help=True).all()
    return render_template ('waiter.html',cooked_orders=cooked_orders, uncooked_orders=uncooked_orders, tables=tables)
    #Now waiter page should only display items which are cooked.Before it displayed everything added to database.
#Code inspired from sqlalchemy documentation link:https://docs.sqlalchemy.org/en/14/orm/query.html


@app.route('/complete_order/<order_id>', methods=['GET', 'POST'])
def complete_order(order_id):
    order = db.Orders.query.get(order_id)
    if order:
        if order.cooked == True:
            db.Items.query.filter_by(order_id=order.order_id).delete()
            # Delete the order from the Orders table
            db.Orders.query.filter_by(order_id=order.order_id).delete()
            restaurant_db.session.commit()
            order.delivered = True


        elif order.cooked == False:
            db.Items.query.filter_by(order_id=order.order_id).delete()
            # Delete the order from the Orders table
            db.Orders.query.filter_by(order_id=order.order_id).delete()
            restaurant_db.session.commit()


    return redirect(url_for('waiter'))
```



```
@app.route('/confirm_order/<order_id>', methods=['GET', 'POST'])
def confirm_order(order_id):
    order = db.Orders.query.get(order_id)
    order.confirmed = True
    restaurant_db.session.commit()
    return redirect(url_for('waiter'))
```

The whole waiter page functionality was quite a challenge for me as I was assigned to make it work in such a way so both cooked and uncooked orders were displayed on the waiter's page. The 'waiter()' function retrieved cooked and uncooked orders from the database and displayed them on the waiter page so staff can see cooked and uncooked orders. The complete_order method uses the 'order_id' parameter to update the status of orders and delete items and orders from the database if the order is cooked and marks the order as delivered. If an order is uncooked only the corresponding items are deleted from the orders table changes are committed to the database and the user is redirected to the 'waiter()' function to display updated orders. The above methods work with the buttons in the HTML page to make it function properly.

Section 2:

EI: User inputs that add, modify or delete data within the restaurant application system

User Requirement:	**Priority: H=High,A=Average,L=Low**
Register client	H
Add order	H
Modify order	A
Delete order	L

Function Point for EI:
L = 3 FP
A = 4 FP
H = 6 FP

Total EIs = 4 **Total EIs FP** = (3 x 1) + (1 x 4) + (1 x 6) + (1 x 6) = 19

EO: External Outputs: Data generated by the system that is sent to users

User Requirement:	**Priority: H=High,A=Average,L=Low**
Total spent by all clients in each age range	A
Order information	H

Function Point for EO:
L = 4 FP
A = 5 FP
H = 7 FP

Total EOs = 2 **Total EOs FP** = (5 x 1) + (1 x 7) = 12

EQ: External Inquiries: Requests for data by users

User Requirement:	**Priority: H=High,A=Average,L=Low**
Get order status	L
Get client details	L

Function Point for EQ:
L = 3 FP
A = 4 FP
H = 6 FP

Total EQs = 2 **Total EQs FP** = (3 x 1) + (1 x 3) = 6

ILF: User data stored and processed within the system

User Requirement:	**Priority: H=High,A=Average,L=Low**
Client information	H
Order information	H

Function Point for ILF:
L = 7 FP
A = 10 FP
H = 15 FP

Total ILFs = 2   **Total ILFs FP** = (15 x 1) + (1 x 15) = 30

EIF: Data shared between the system and external entities

Total EIFs = None
Total EIFs FP = 0

Total Unadjusted Function Points (UFC) = 19 + 12 + 6 + 30 + 0 = 67

Therefore, the estimated Unadjusted Function Points (UFC) is 67

<u>Reference Sheet used to determine function points.</u>

Section 3:Learning Outcome

Utilizing the scrum development process during a group project provided me with invaluable lessons. Through the use of sprints over short periods, I learned the significance of remaining organized in such an environment, which includes scheduled meetups to discuss issues, effective communication, efficient teamwork, and adaptability. Since we had a large group and members with varying levels of experience, it was crucial for me to be adaptable to ensure every group member contributed equally. We prioritized specific tasks differently, such as initializing the database which was our top priority in the first two sprints, without which none of us could've begun with the assigned tasks. The utilization of scrum allowed me to prepare a plan ahead of important team meetings to identify and address any issues. Collaborating with my teammates, and knowing we all shared the same goal, enabled me to make progress and overcome any challenges I faced with my tasks. I also learned to take accountability as each sprint we were given tasks based on difficulty and agreed to assign tasks based on the individual's capability. This created a stronger bond between us as we trusted each other to complete the assigned tasks before the deadline. Even though we encountered challenges and had to pivot quickly at some point where everything went down, we made sure to be flexible and adjust tasks as necessary to get everything back on track.

Section 4: Refactoring in team project

We unknowingly refactored the code as we continued to rethink for a better and optimal solution at the beginning of each sprint after new features were implemented. It was only after sprint 3 that we realized the importance of refactoring. Refactoring made the code easier to read, modify, and extend for further sprints as we added new functionalities. We followed a specific workflow of saving the original code under our own branch, ensuring the code still worked as expected, simplifying code by refactoring, retesting to see if the code worked as expected, and merging it to the main if it did. If it did not, we fixed the code and continued the cycle.

We discussed ideas for constant improvement to enhance the look of our website during each sprint. We thoroughly refactored the code after most of our user stories were implemented to find bugs and any other issues the product had. This saved us time as it helped reduce debugging time, so we could work on any remaining tasks.

The code was readable and well-maintained, and we could reuse code that was previously used elsewhere. In the last sprint, I searched for code smells, which is a huge part of refactoring. I removed any unwanted codes present in the main application, including dead codes, duplicated code, and modified long methods, so the code was easily readable and organized.