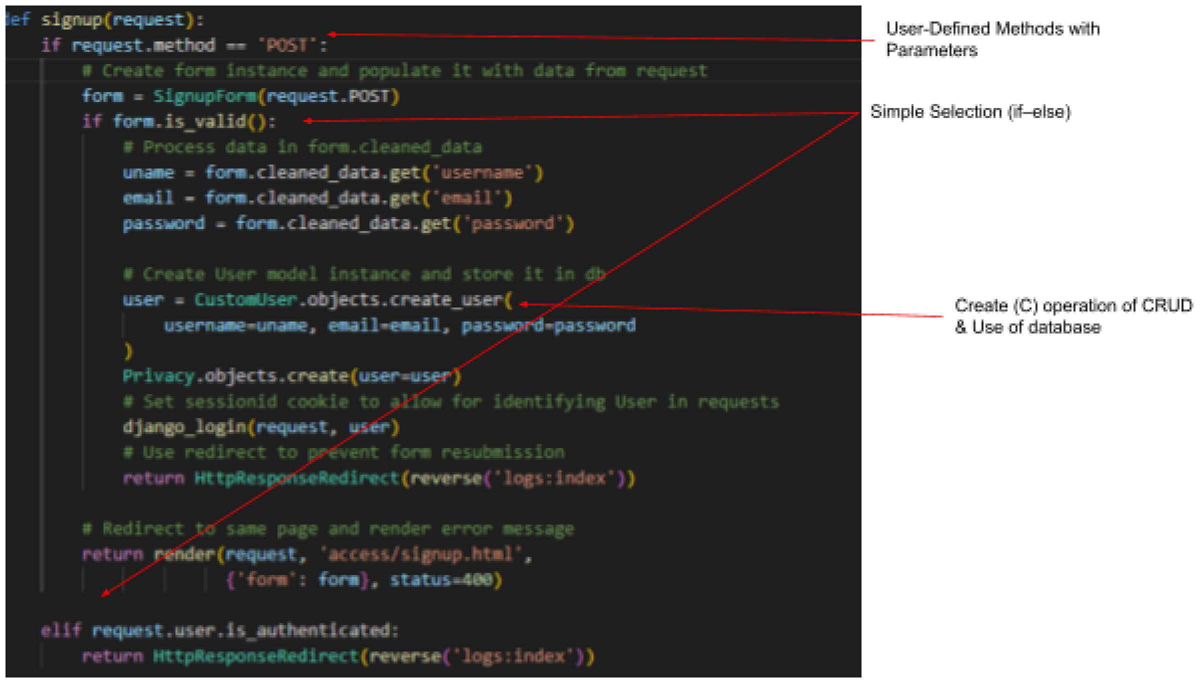
Success criteria for product:

1. Authentication: Working Sign up and Login System
2. Questionnaire: recommended daily calorie and protein amounts
3. Home Page. Displays meal logs, calorie and protein amounts, and navigation bar
4. The user is capable of adding, editing, and deleting meals
5. Calorie/Protein Total and Remaining

Complexities/Techniques Used

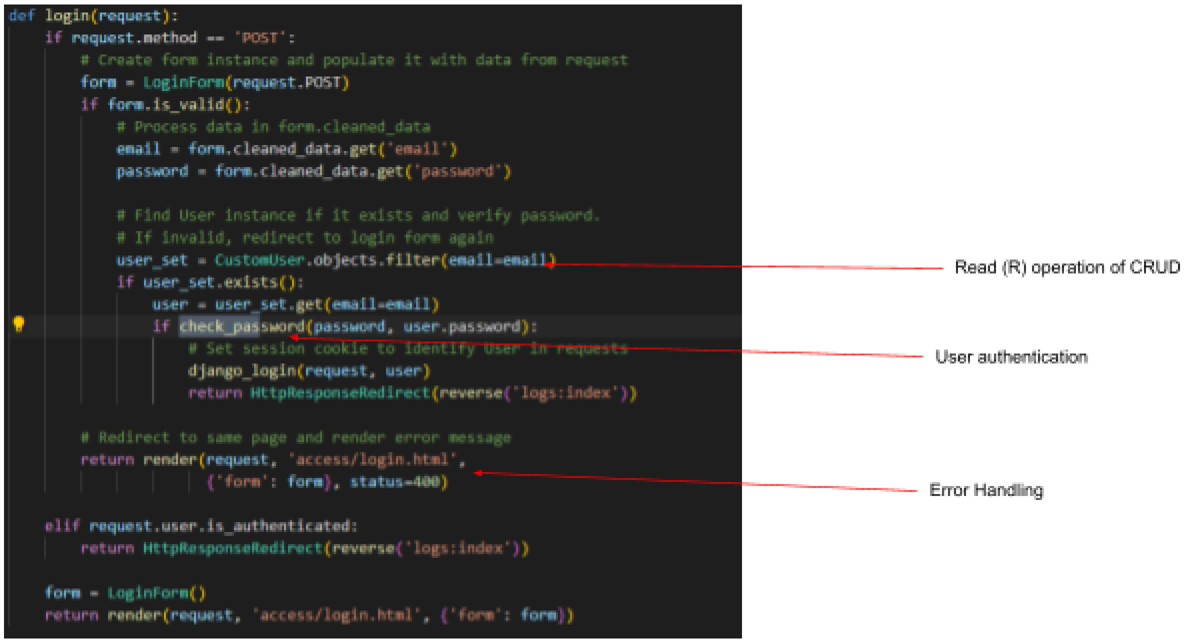
1. Object Oriented Programming
2. User Authentication
3. Dictionary Data Structures - Associative Array
4. CRUD operations
5. Error handling
6. Data Handling
7. Use of additional libraries
8. Use of databases
9. Encapsulation
10. User-Defined Objects
11. User-defined methods with parameters (the parameters have to be useful and used within the method body)
12. Templates and Context Management
13. User-defined methods with appropriate return values (primitives or objects)
14. Querying and Date/Time Handling
15. Simple selection (if–else)
16. Complex selection (nested if, if with multiple conditions or switch)

1. Authentication: Working Sign up and Login System


User-Defined Methods with Parameters
Simple Selection (if–else)
Create (C) operation of CRUD & Use of database

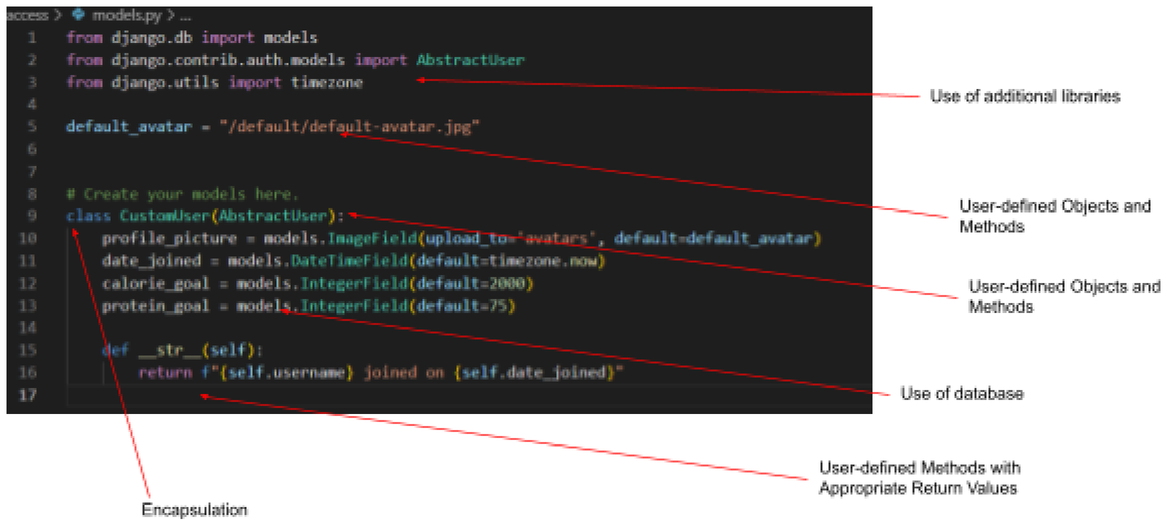
```
def signup(request):
    if request.method == 'POST':
        # Create form instance and populate it with data from request
        form = SignupForm(request.POST)
        if form.is_valid():
            # Process data in form.cleaned_data
            uname = form.cleaned_data.get('username')
            email = form.cleaned_data.get('email')
            password = form.cleaned_data.get('password')

            # Create User model instance and store it in db
            user = CustomUser.objects.create_user(
                username=uname, email=email, password=password
            )
            Privacy.objects.create(user=user)
            # Set sessionid cookie to allow for identifying User in requests
            django_login(request, user)
            # Use redirect to prevent form resubmission
            return HttpResponseRedirect(reverse('logs:index'))

        # Redirect to same page and render error message
        return render(request, 'access/signup.html',
                      {'form': form}, status=400)

    elif request.user.is_authenticated:
        return HttpResponseRedirect(reverse('logs:index'))
```

(Appendix 2)



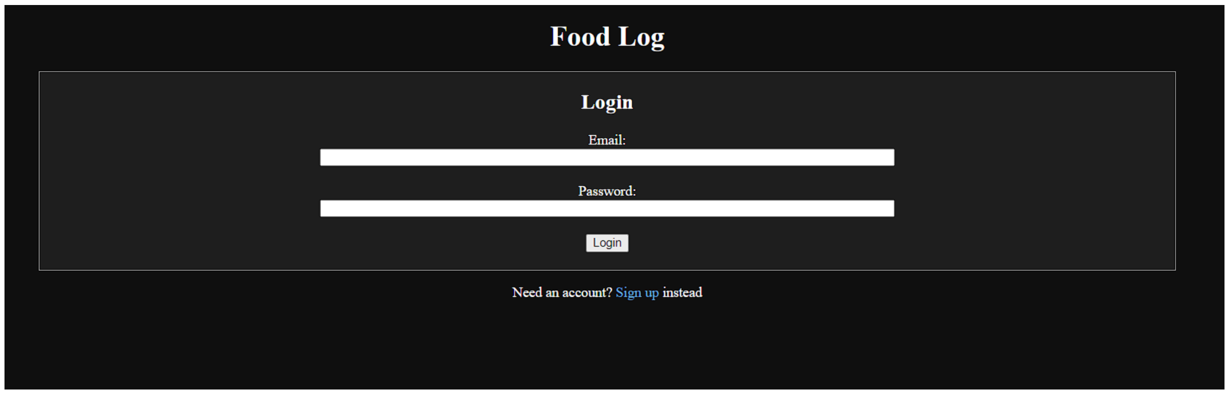
```
def login(request):
    if request.method == 'POST':
        # Create form instance and populate it with data from request
        form = LoginForm(request.POST)
        if form.is_valid():
            # Process data in form.cleaned_data
            email = form.cleaned_data.get('email')
            password = form.cleaned_data.get('password')

            # Find User instance if it exists and verify password.
            # If invalid, redirect to login form again
            user_set = CustomUser.objects.filter(email=email)
            if user_set.exists():
                user = user_set.get(email=email)
                if check_password(password, user.password):
                    # Set session cookie to identify User in requests
                    django_login(request, user)
                    return HttpResponseRedirect(reverse('logs:index'))

        # Redirect to same page and render error message
        return render(request, 'access/login.html',
                      {'form': form}, status=400)

    elif request.user.is_authenticated:
        return HttpResponseRedirect(reverse('logs:index'))

    form = LoginForm()
    return render(request, 'access/login.html', {'form': form})
```

(Appendix 3)


access > models.py > ...
Use of additional libraries
User-defined Objects and Methods
User-defined Objects and Methods
Use of database
User-defined Methods with Appropriate Return Values
Encapsulation

```python
from django.db import models
from django.contrib.auth.models import AbstractUser
from django.utils import timezone

default_avatar = "/default/default-avatar.jpg"


# Create your models here.
class CustomUser(AbstractUser):
    profile_picture = models.ImageField(upload_to='avatars', default=default_avatar)
    date_joined = models.DateTimeField(default=timezone.now)
    calorie_goal = models.IntegerField(default=2000)
    protein_goal = models.IntegerField(default=75)

    def __str__(self):
        return f"{self.username} joined on {self.date_joined}"

```

(Appendix 4)



Food Log

Login

Email:

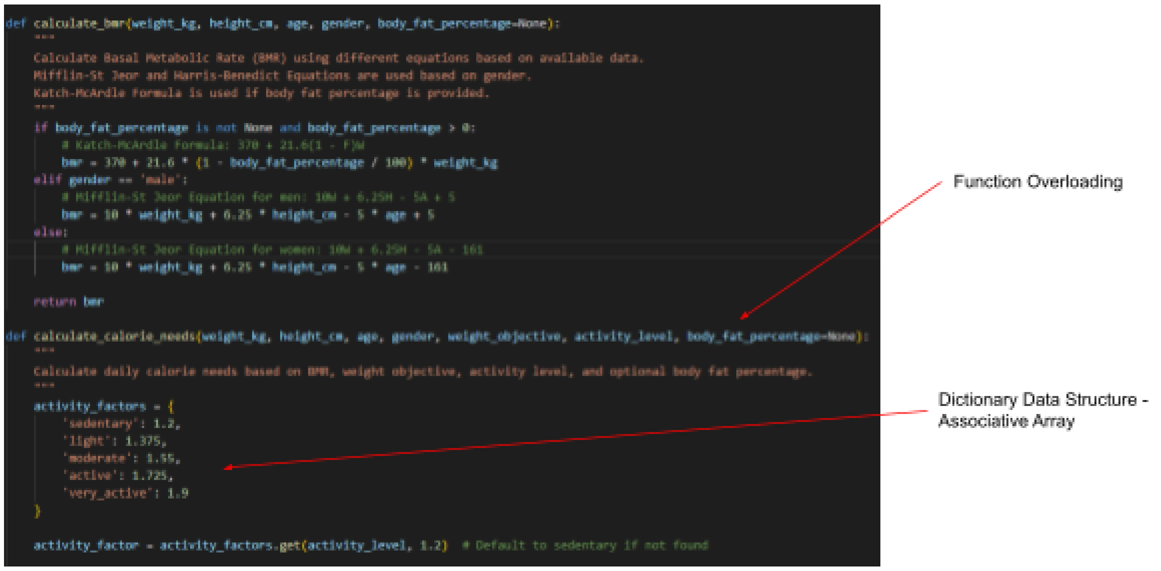
Password:

Need an account? Sign up instead

(Appendix 11)

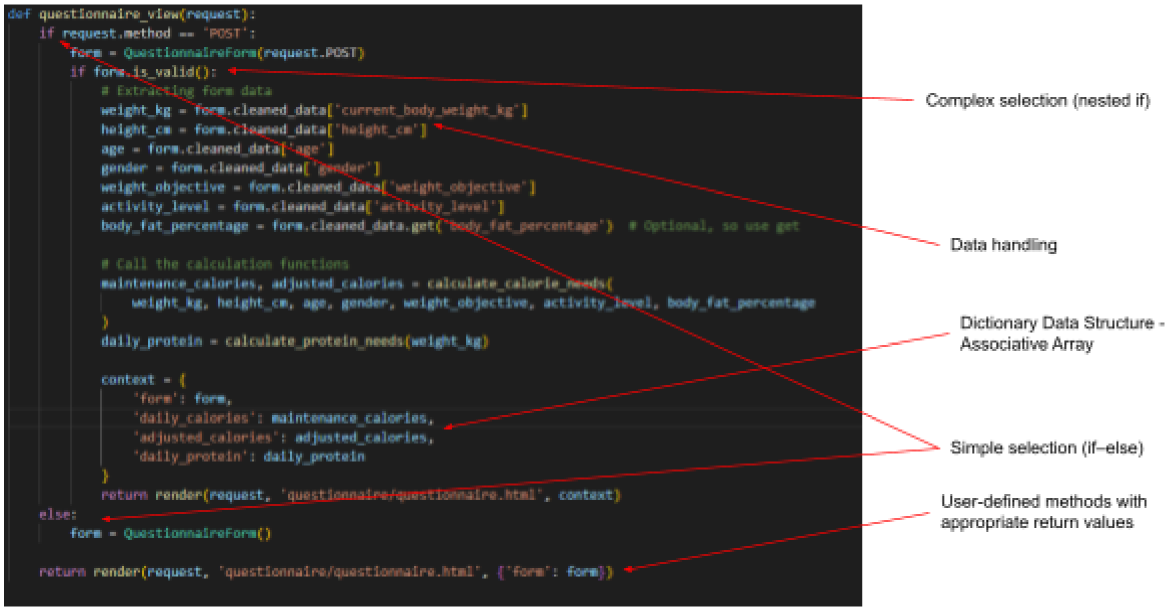
I used an SQLite database to store user information for a login and sign-up system. When a new user registers, their details are added to the database. Each user entry includes a username, password (Encapsulation), and default values for calorie and protein goals. The sign-up page displays a form for users to enter their desired information. Upon submission, the form data is validated to ensure uniqueness (Error Handling). The registration process involves creating instances of the SignupForm and CustomUser models (Encapsulation). The form's methods check if the provided username and email already exist in the database (CRUD Operations). If not unique, a validation error is raised. Once validated, a new CustomUser object is created using Django's ORM (Database ORM). The ORM abstracts database interaction, allowing Python object interaction. The sign-up system also incorporates user authentication. Upon login, the system validates credentials by checking email existence and password matching (Error Handling). If valid, the user is redirected.

2. Questionnaire: calculations for recommended daily calorie and protein amounts



```
def calculate_bmr(weight_kg, height_cm, age, gender, body_fat_percentage=None):
    """
    Calculate Basal Metabolic Rate (BMR) using different equations based on available data.
    Mifflin-St Jeor and Harris-Benedict Equations are used based on gender.
    Katch-McArdle Formula is used if body fat percentage is provided.
    """
    if body_fat_percentage is not None and body_fat_percentage > 0:
        # Katch-McArdle Formula: 370 + 21.6(1 - F)W
        bmr = 370 + 21.6 * (1 - body_fat_percentage / 100) * weight_kg
    elif gender == 'male':
        # Mifflin-St Jeor Equation for men: 10W + 6.25H - 5A + 5
        bmr = 10 * weight_kg + 6.25 * height_cm - 5 * age + 5
    else:
        # Mifflin-St Jeor Equation for women: 10W + 6.25H - 5A - 161
        bmr = 10 * weight_kg + 6.25 * height_cm - 5 * age - 161

    return bmr

def calculate_calorie_needs(weight_kg, height_cm, age, gender, weight_objective, activity_level, body_fat_percentage=None):
    """
    Calculate daily calorie needs based on BMR, weight objective, activity level, and optional body fat percentage.
    """
    activity_factors = {
        'sedentary': 1.2,
        'light': 1.375,
        'moderate': 1.55,
        'active': 1.725,
        'very_active': 1.9
    }

    activity_factor = activity_factors.get(activity_level, 1.2)  # Default to sedentary if not found
```

(Appendix 5)

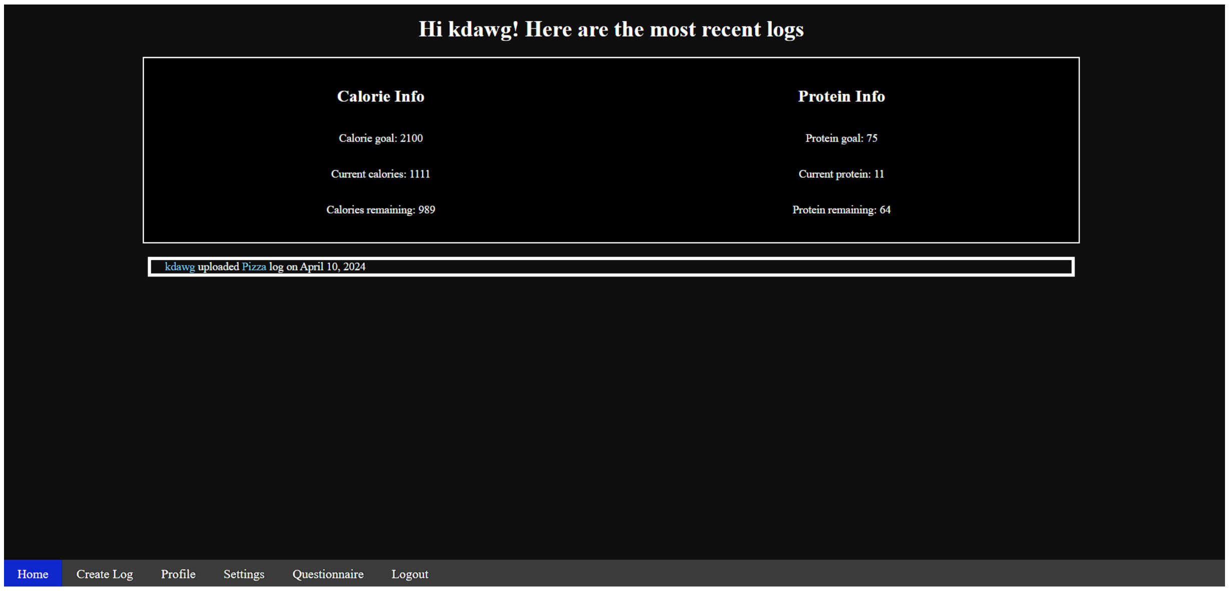

Complex selection (nested if)
Data handling
Dictionary Data Structure - Associative Array
Simple selection (if–else)
User-defined methods with appropriate return values

```
def questionnaire_view(request):
    if request.method == 'POST':
        form = QuestionnaireForm(request.POST)
        if form.is_valid():
            # Extracting form data
            weight_kg = form.cleaned_data['current_body_weight_kg']
            height_cm = form.cleaned_data['height_cm']
            age = form.cleaned_data['age']
            gender = form.cleaned_data['gender']
            weight_objective = form.cleaned_data['weight_objective']
            activity_level = form.cleaned_data['activity_level']
            body_fat_percentage = form.cleaned_data.get('body_fat_percentage')  # Optional, so use get

            # Call the calculation functions
            maintenance_calories, adjusted_calories = calculate_calorie_needs(
                weight_kg, height_cm, age, gender, weight_objective, activity_level, body_fat_percentage
            )
            daily_protein = calculate_protein_needs(weight_kg)

            context = {
                'form': form,
                'daily_calories': maintenance_calories,
                'adjusted_calories': adjusted_calories,
                'daily_protein': daily_protein
            }
            return render(request, 'questionnaire/questionnaire.html', context)
    else:
        form = QuestionnaireForm()

    return render(request, 'questionnaire/questionnaire.html', {'form': form})
```

(Appendix 6)

The questionnaire system collects user data through a web form (data handling) to calculate Basal Metabolic Rate (BMR), and therefore daily calorie and protein needs through that. The code uses Mifflin-St Jeor and Harris-Benedict Equations as well as the Katch-McArdle Formula for these calculations. Using simple selection, it checks if the form is submitted via POST, then processes the validated data. The system employs complex selection in the calculate_bmr function to choose the appropriate calculation method based on user inputs like gender and body fat percentage. Calorie needs are determined by calculate_calorie_needs, which uses an associative array (dictionary).

3. Home Page. Displays meal logs, calorie and protein amounts, and navigation bar



Hi kdawg! Here are the most recent logs

Calorie Info

Calorie goal: 2100

Current calories: 1111

Calories remaining: 989

Protein Info

Protein goal: 75

Current protein: 11

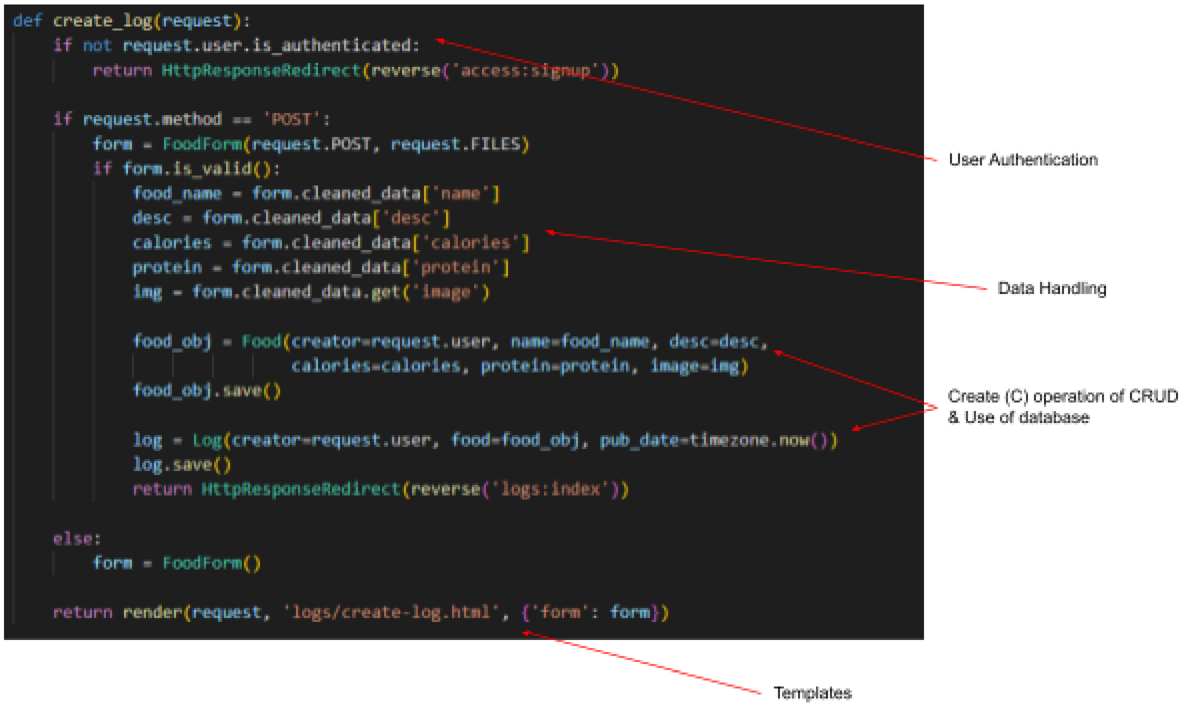
Protein remaining: 64

kdawg uploaded Pizza log on April 10, 2024

(Appendix 11)

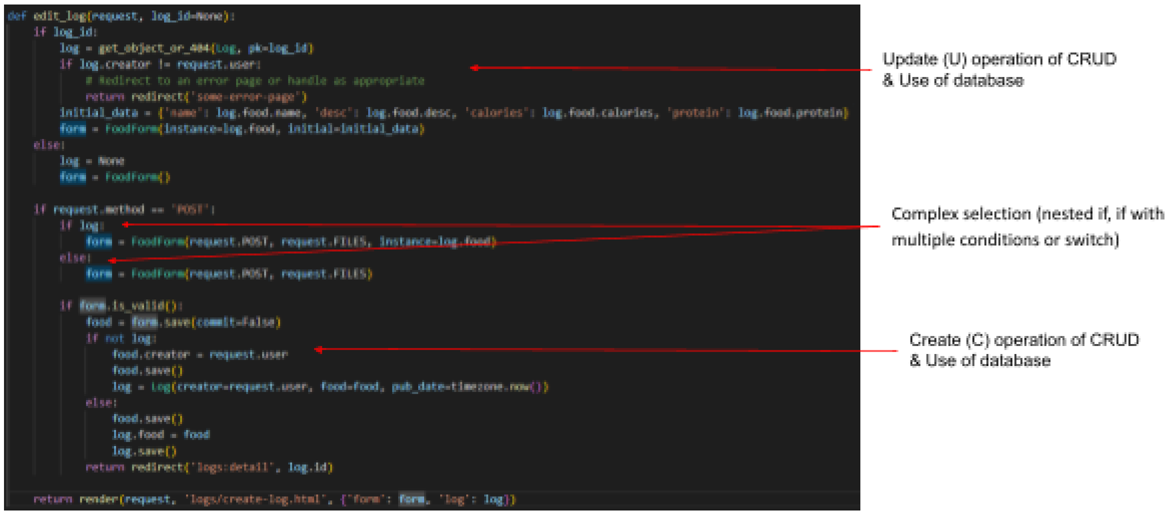
The Home Page, implemented using HTML and CSS, serves as the primary interface for the application. This page employs Object-Oriented Programming principles, organizing various components such as meal logs into User-Defined Objects that encapsulate their attributes and behaviors. The center of the home page features all meal logs you create in a list format, a navigation bar, and calorie/protein info. User authentication state influences the content displayed on the home page, and the website in general. Simple selection (if–else) is used to differentiate between guest and logged-in user views.

4. The user is capable of adding, editing, and deleting meals



```
def create_log(request):
    if not request.user.is_authenticated:
        return HttpResponseRedirect(reverse('access:signup'))

    if request.method == 'POST':
        form = FoodForm(request.POST, request.FILES)
        if form.is_valid():
            food_name = form.cleaned_data['name']
            desc = form.cleaned_data['desc']
            calories = form.cleaned_data['calories']
            protein = form.cleaned_data['protein']
            img = form.cleaned_data.get('image')

            food_obj = Food(creator=request.user, name=food_name, desc=desc,
                            calories=calories, protein=protein, image=img)
            food_obj.save()

            log = Log(creator=request.user, food=food_obj, pub_date=timezone.now())
            log.save()
            return HttpResponseRedirect(reverse('logs:index'))

    else:
        form = FoodForm()

    return render(request, 'logs/create-log.html', {'form': form})
```

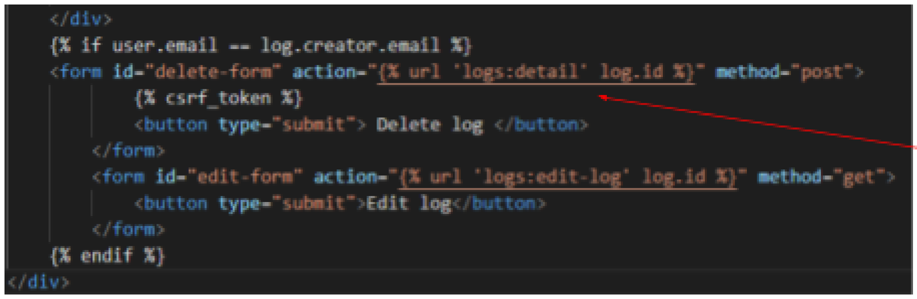
(Appendix 7)



```
def edit_log(request, log_id=None):
    if log_id:
        log = get_object_or_404(Log, pk=log_id)
        if log.creator != request.user:
            # Redirect to an error page or handle as appropriate
            return redirect('some-error-page')
        initial_data = {'name': log.food.name, 'desc': log.food.desc, 'calories': log.food.calories, 'protein': log.food.protein}
        form = FoodForm(instance=log.food, initial=initial_data)
    else:
        log = None
        form = FoodForm()

    if request.method == 'POST':
        if log:
            form = FoodForm(request.POST, request.FILES, instance=log.food)
        else:
            form = FoodForm(request.POST, request.FILES)

        if form.is_valid():
            food = form.save(commit=False)
            if not log:
                food.creator = request.user
                food.save()
                log = Log(creator=request.user, food=food, pub_date=timezone.now())
            else:
                food.save()
                log.food = food
                log.save()
            return redirect('logs:detail', log.id)

    return render(request, 'logs/create-log.html', {'form': form, 'log': log})
```

Update (U) operation of CRUD & Use of database
Complex selection (nested if, if with multiple conditions or switch)
Create (C) operation of CRUD & Use of database

(Appendix 8)



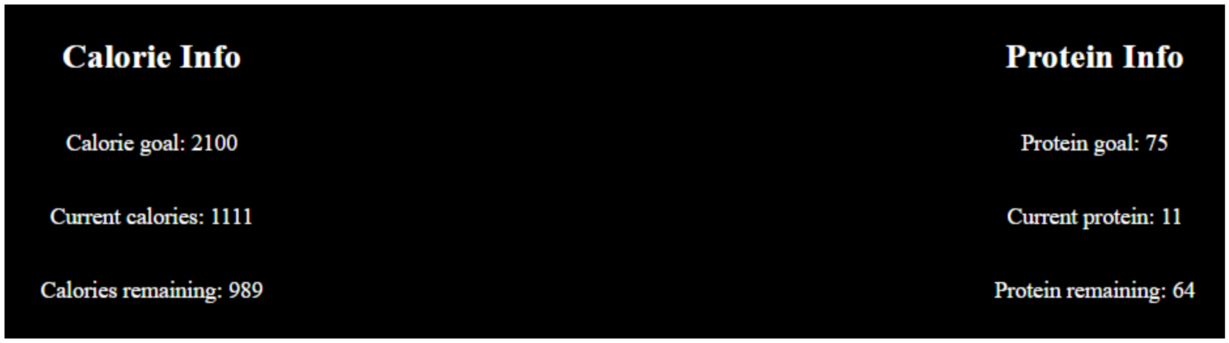
```
    </div>
    {% if user.email == log.creator.email %}
    <form id="delete-form" action="{% url 'logs:detail' log.id %}" method="post">
            {% csrf_token %}
            <button type="submit"> Delete log </button>
        </form>
        <form id="edit-form" action="{% url 'logs:edit-log' log.id %}" method="get">
            <button type="submit">Edit log</button>
        </form>
    {% endif %}
</div>
```

Delete (D) operation of CRUD & Use of database

(Appendix 9)

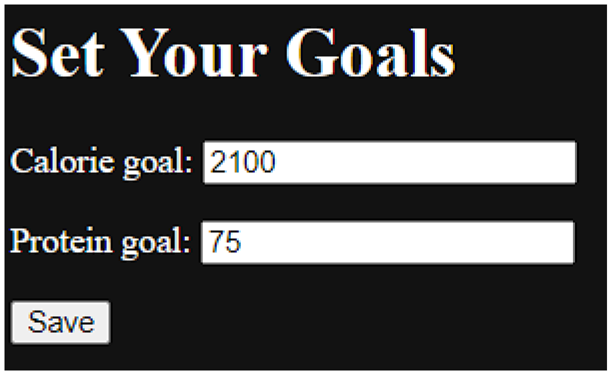
The system for adding, editing, and deleting meals in the application is a complete implementation of CRUD (Create, Read, Update, Delete) operations. The 'Create' operation is initiated when a user adds a new meal through a form on an HTML page, entering details like name, calories, and an optional image, which is then saved to the database. Users can 'Read' the meal details on a view page, displaying the entered information. For modifications, the 'Update' operation is used, where users edit meal details through a pre-filled form, with changes saved back to the database. The 'Delete' operation allows users to remove unwanted meal logs.

5. Calorie/Protein Total and Remaining: calculations displaying these values



Calorie Info	Protein Info
Calorie goal: 2100	Protein goal: 75
Current calories: 1111	Current protein: 11
Calories remaining: 989	Protein remaining: 64

(Appendix 11)

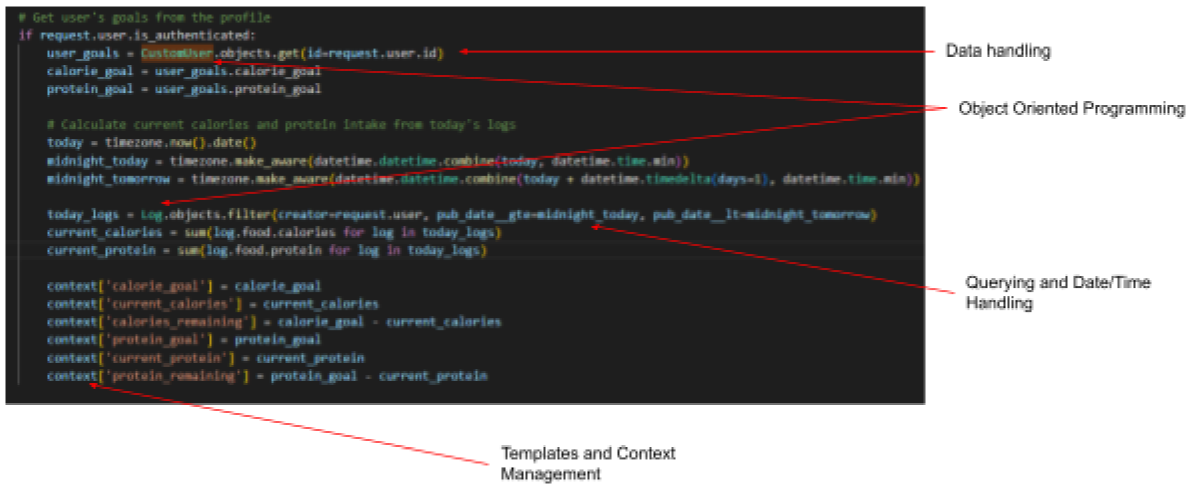


Set Your Goals

Calorie goal: 2100

Protein goal: 75

(Appendix 11)


Data handling
Object Oriented Programming
Querying and Date/Time Handling
Templates and Context Management

```python
# Get user's goals from the profile
if request.user.is_authenticated:
    user_goals = CustomUser.objects.get(id=request.user.id)
    calorie_goal = user_goals.calorie_goal
    protein_goal = user_goals.protein_goal

    # Calculate current calories and protein intake from today's logs
    today = timezone.now().date()
    midnight_today = timezone.make_aware(datetime.datetime.combine(today, datetime.time.min))
    midnight_tomorrow = timezone.make_aware(datetime.datetime.combine(today + datetime.timedelta(days=1), datetime.time.min))

    today_logs = Log.objects.filter(creator=request.user, pub_date__gte=midnight_today, pub_date__lt=midnight_tomorrow)
    current_calories = sum(log.food.calories for log in today_logs)
    current_protein = sum(log.food.protein for log in today_logs)

    context['calorie_goal'] = calorie_goal
    context['current_calories'] = current_calories
    context['calories_remaining'] = calorie_goal - current_calories
    context['protein_goal'] = protein_goal
    context['current_protein'] = current_protein
    context['protein_remaining'] = protein_goal - current_protein
```

(Appendix 10)

In the application, user-specific calorie and protein goals are calculated and displayed based on the individual's daily food intake, which is derived from their account information stored in the CustomUser model. The process begins by fetching the user's predefined calorie and protein targets. Then, to calculate the current day's intake, the system sums up the calories and proteins from food items logged by the user for that day, using Django's ORM to filter the records accurately. The actual intake values, current_calories and current_protein are aggregated from these logs and displayed using HTML and CSS.

Works Cited

"Calorie Calculator." *Calculator.net*, 2024, www.calculator.net/calorie-calculator.html. Accessed 10 Apr. 2024.

in. "Building a Website with Django & Python." *YouTube*, 2024, www.youtube.com/playlist?list=PLgCYzUzKIBE_dil025VAJnDjNZHHHR9mW. Accessed 10 Apr. 2024.