Course Of Study

Software Development

Course Title

Object Oriented and Functional Programming with Python (DLBDSOOFPP01)

Matriculation No

92107689

Tutor

Max Pumperla

Date

November 14, 2024

1. Introduction

In today's fast-paced world, maintaining good habits and breaking bad ones is essential for personal growth and productivity. A habit tracker app serves as a powerful tool to help users establish and maintain these habits by allowing them to track, manage, and analyze their routines effectively (Axelos, 2019). This project aims to create a simple, user-friendly, and functional habit tracking app that leverages modern web technologies to offer a smooth user experience and robust data management capabilities.

This habit tracking application has been developed using the Flask framework for its lightweight, efficient handling of web requests and modular design, ideal for building flexible web applications (Flask, n.d.). For the backend database, PostgreSQL has been chosen to store user data and habit records securely. PostgreSQL's advanced data handling capabilities and compatibility with Python make it an excellent choice for building scalable web applications (Psycopg2, n.d.).

Key Components and Technologies:

1. Flask: Provides a solid foundation for handling web requests, routing, and API management. Flask's extensibility allows for easy integration of various components like login management, templating, and database interactions (Flask, n.d.).
2. PostgreSQL: A powerful relational database, PostgreSQL is used for storing and managing user and habit data. By utilizing psycopg2 and cursor operations, this app ensures efficient database communication and querying (Psycopg2, n.d.).
3. Flask-Login: To ensure secure access to user-specific data, the app uses Flask-Login for handling user authentication, allowing users to log in and manage their personalized habit data securely (Salaets, 2020).
4. Jinja Templating: Jinja is used to dynamically render HTML templates, allowing for a responsive and interactive user interface. This framework enables the creation of a seamless user experience with efficient use of reusable templates (Schmidt, 2020).
5. psycopg2 with Cursor: This database adapter allows for efficient connection and interaction with PostgreSQL. The cursor functions handle database queries, data insertion, and retrieval in an organized, manageable way (Psycopg2, n.d.).

App Features:

- User Authentication: Users can securely log in and manage their own habits, ensuring privacy and a personalized experience.
- Habit Creation and Tracking: Users can define habits with specific schedules (e.g., daily, weekly) and track their progress over time.

- Analytics: The app includes an analytics module where users can view their habit streaks, analyze which habits they have maintained, and track their progress.

1. Implementation

1.1. Class: DataManager:

The DataManager class is designed to handle database interactions for habit tracking, including creating tables, inserting data, retrieving habit records, and updating habit completion statuses. It uses psycopg2 for connecting to a PostgreSQL database, and several helper methods to manage and retrieve data.

1.1.1. Method: Create_habits_table:

```
def create_habits_table(self):
    """create table if the table does not exist in the database"""
    try:
        with psycopg2.connect(**self.connection_params) as conn:
            with conn.cursor() as cursor:
                # Create a table
                create_table_query = f'''
                CREATE TABLE IF NOT EXISTS {self.habit_table_name} (
                    id SERIAL PRIMARY KEY,
                    habit_name VARCHAR(100) NOT NULL,
                    frequency VARCHAR(10) NOT NULL,
                    start_date DATE NOT NULL,
                    start_day INT NOT NULL,
                    start_month INT NOT NULL,
                    start_year INT NOT NULL,
                    is_completed BOOL NOT NULL,
                    completed_date DATE,
                    number_of_weeks INT NOT NULL,
                    day_of_days_in_year INT NOT NULL
                );
                '''
                cursor.execute(create_table_query)
                conn.commit()
    except Exception as e:
        print(e)
```

This method creates the main table (habit_tracking) for storing habit data if it doesn't already exist and ensures that the table structure aligns with the expected data types and structure.

1.1.2. Method: insert_data:

```
def insert_data(self, habit_name, frequency, start_date, start_day,     start_month,start_year,
is_completed,completed_date, number_of_week,day_of_days_in_years):
    """add habit into the db table"""
    try:
        with psycopg2.connect(**self.connection_params) as conn:
            with conn.cursor() as cursor:
                # Insert data
                insert_data_query = f"""
                INSERT INTO {self.habit_table_name} (habit_name, frequency, start_date, start_day,
start_month,
                start_year, is_completed, completed_date, number_of_weeks, day_of_days_in_year)
VALUES (%s, %s,
                %s, %s, %s, %s, %s, %s, %s);"""
                cursor.execute(insert_data_query, (habit_name, frequency, start_date,  start_day,
start_month, start_year, is_completed, completed_date,number_of_week,
day_of_days_in_years))
                conn.commit()
                return "Success"
    except Exception as e:
        print(e)
        if type(e).__name__ == 'UndefinedTable':
            return "UndefinedTable"
```

This method inserts a new record into the habit_tracking table with the provided data. The parameters correspond to each column in the table, allowing for detailed tracking of each habit's attributes. It commits the transaction to ensure the data is saved.

1.1.3. Method : retrieve_current_habits

```
def retrieve_current_habits(self, frequency):
    """retrieve active habits. frequency is used to specify and filter the period habit eg. daily or
weekly"""
    try:
        with psycopg2.connect(**self.connection_params) as conn:
            with conn.cursor(cursor_factory=RealDictCursor) as cursor:
```

```
                if frequency.lower() == "daily":
                    # Retrieve all data
                    cursor.execute(f"""SELECT * FROM {self.habit_table_name}  WHERE frequency =
            %s AND day_of_days_in_year = %s AND is_completed = %s""",
                                   (frequency, day_of_days_in_year, False))
                    rows = cursor.fetchall()
                    return rows
                elif frequency.lower() == "weekly":
                    cursor.execute(f"""SELECT * FROM {self.habit_table_name}  WHERE frequency =
            %s AND
                    number_of_weeks = %s AND is_completed = %s""", (frequency, number_of_weeks,
            False))
                    rows = cursor.fetchall()
                    return rows
                else:
                    cursor.execute(f"SELECT * FROM {self.habit_table_name} ;")
                    rows = cursor.fetchall()
                    return rows
            except ProgrammingError as e:
                return "Table not found"

            except Exception as e:
                if type(e).__name__:
                    return "UndefinedTable"
```

This method retrieves current (incomplete) habits filtered by frequency (daily or weekly). It filters based on the is_completed status and checks against the current day or week. If no frequency filter is provided, it retrieves all habits in the table.

1.1.4. Method: retrieve_completed_habits:

```
def retrieve_completed_habits(self, frequency):
    """when checkout, the 'is_completed ' table column is updated to TRUE, this will filter and get the values"""
    try:
        with psycopg2.connect(**self.connection_params) as conn:
            with conn.cursor(cursor_factory=RealDictCursor) as cursor:
                if frequency.lower() == "daily":
                    # Retrieve all data
                    cursor.execute(f"""SELECT * FROM {self.habit_table_name}  WHERE frequency = %s AND
```

```
day_of_days_in_year = %s AND is_completed = %s""",
                        (frequency, day_of_days_in_year, True))
            rows = cursor.fetchall()
            return rows
        elif frequency.lower() == "weekly":
            cursor.execute(f"""SELECT * FROM {self.habit_table_name}  WHERE frequency = %s AND
            number_of_weeks = %s AND is_completed = %s""", (frequency, number_of_weeks, True))
            rows = cursor.fetchall()
            return rows
        else:
            cursor.execute(f"SELECT * FROM {self.habit_table_name} ;")
            rows = cursor.fetchall()
            return rows
    except Exception as e:
        if type(e).__name__:
            return "UndefinedTable"
```

Similar to retrieve_current_habits, this method retrieves completed habits based on frequency. It filters by the current day or week and only retrieves habits with is_completed set to True.

1.1.5. Method: update_completed_habit:

```
def update_completed_habit(self, is_completed, completed_date, habit_id):
    """this will update and 'is_completed' field to TRUE"""
    with psycopg2.connect(**self.connection_params) as conn:
        with conn.cursor() as cursor:
            update_query = f'''
            UPDATE {self.habit_table_name}
            SET is_completed = %s, completed_date = %s
            WHERE id = %s;
            '''
            cursor.execute(update_query, (
                is_completed, completed_date, habit_id))
            conn.commit()
            return "Success"
```

Updates the status of a specific habit to completed. This method sets is_completed to True and updates completed_date for the habit with the specified habit_id.

1.1.6. Method : create_streak_completed_table:

```
def create_streak_completed_table(self):
    """create streak table if the table does not exist in the database"""
    try:
        with psycopg2.connect(**self.connection_params) as conn:
            with conn.cursor() as cursor:
                # Create a table
                create_table_query = f"""
                CREATE TABLE IF NOT EXISTS {self.streak_complete_table} (
                    id SERIAL PRIMARY KEY,
                    frequency VARCHAR(10) NOT NULL,
                    start_date DATE NOT NULL,
                    start_day INT NOT NULL,
                    start_month INT NOT NULL,
                    start_year INT NOT NULL
                );
                """
                cursor.execute(create_table_query)
                conn.commit()
    except Exception as e:
        print("Error creating table:", e)
```

Creates a table (streak_completed) for tracking streaks if it doesn't exist. This table logs the frequency and start date information of completed streaks, which helps in calculating and analyzing habit streaks over time.

1.1.7. Method: insert_into_streak_table:

```
        def insert_into_streak_table(self, frequency, start_date, start_day, start_month, start_year):
            """insert data into the streak table"""
            try:
                with psycopg2.connect(**self.connection_params) as conn:
                    with conn.cursor() as cursor:
                        insert_data_query = f"""
                            INSERT INTO {self.streak_complete_table} (frequency, start_date, start_day,
    start_month,
                            start_year) VALUES (%s, %s, %s, %s, %s);"""
                        cursor.execute(insert_data_query, (frequency, start_date, start_day, start_month,
```

```
                                        start_year))
                    conn.commit()
                    return "Success"
                except Exception as e:
                    print(e)
                    if type(e).__name__ == 'UndefinedTable':
                        return "UndefinedTable"
```

Inserts a new entry into the streak_completed table. This method records a completed habit streak with associated date and frequency information, which is useful for analyzing consistency in habit completion.

1.1.8. Method: retrieve_streak_data:

```
def retrieve_streak_data(self, frequency):
    """Retrieve streak data"""
    try:
        with psycopg2.connect(**self.connection_params) as conn:
            with conn.cursor(cursor_factory=RealDictCursor) as cursor:
                if frequency.lower() == "daily":
                    cursor.execute(f"""SELECT * FROM {self.streak_complete_table}  WHERE frequency = %s""",
                                   (frequency,))
                    rows = cursor.fetchall()
                    return rows
                elif frequency.lower() == "weekly":
                    cursor.execute(f"""SELECT * FROM {self.streak_complete_table}  WHERE frequency = %s """,
                                   (frequency,))
                    rows = cursor.fetchall()
                    return rows
                else:
                    cursor.execute(f"SELECT * FROM {self.streak_complete_table} ;")
                    rows = cursor.fetchall()
                    return rows
    except Exception as e:
        if type(e).__name__:
            return "UndefinedTable"
```

Retrieves data from the streak_completed table filtered by frequency (daily or weekly). If no specific frequency is specified, it retrieves all data from the table. This method is essential for

calculating habit streaks and tracking the longest consecutive days or weeks of completed habits.

1.2. Class : Habit:

The Habit class is designed to interact with the DataManager and Users classes to manage habit tracking functionalities. It allows creating, retrieving, and updating habit data while also managing user registration and streak data.

1.2.1. Method: create_all_table:

```
def create_all_table(self):
    """create and register users if the table and user does not exist"""
    self.data.create_habits_table()
    self.data.create_streak_completed_table()
    self.user.create_user()
    self.user.register_user(username="obinna@gmail.com", password="qwerty")
```

This method creates the necessary tables for habit tracking and streaks if they don't already exist. It also registers a default user by calling create_user and register_user methods in the Users class. This setup ensures that the habit tracking application has the required database structures and a sample user.

1.2.2. Method: get_current_habits

```
def get_current_habits(self, frequency):
    """get the habits based on frequency. eg if 'weekly' is assigned to frequency, weekly habits will be
    displayed"""
            return self.data.retrieve_current_habits(frequency)
```

Retrieves habits that are currently active (incomplete) based on the specified frequency (daily or weekly). This method calls retrieve_current_habits from DataManager to filter and return habits based on the specified frequency, enabling users to see only relevant active habits.

1.2.3. Method: get_completed_habits

```
def get_completed_habits(self, frequency):
    """get the habits based on frequency. eg if 'weekly' is assigned to frequency, weekly habits will be
    displayed"""
    return self.data.retrieve_completed_habits(frequency)
```

Retrieves habits that have been completed based on the specified frequency (daily or weekly). This method calls retrieve_completed_habits from DataManager to filter and return completed habits according to the given frequency.

1.2.4. Method: create_habit

```
def create_habit(self, frequency, habit_name, day_of_days_in_year, number_of_weeks):
    """create habit"""
    return self.data.insert_data(completed_date=None, is_completed=False, frequency=frequency,
                                 habit_name=habit_name, start_date=today,
                                 start_day=day, start_month=month,
                                 start_year=year, day_of_days_in_years=day_of_days_in_year,
                                 number_of_week=number_of_weeks)
```

Creates a new habit entry in the database. It uses insert_data from DataManager to insert a habit record with the specified parameters. This method defines the frequency and name of the habit, along with start date, day, month, year, and other tracking information.

1.2.5. Method: checkout_habit

```
def checkout_habit(self, date, habit_id):
    """checkoff and call db update method"""
    return self.data.update_completed_habit(is_completed=True, completed_date=date, habit_id=habit_id)
```

Marks a habit as completed by updating the is_completed flag and setting the completed_date to the specified date. It calls update_completed_habit from DataManager, which updates the completion status of the habit with the given habit_id.

1.2.6. Method: insert_complete_streak

```
def insert_complete_streak(self, frequency, start_day=day, start_year=year, start_month=month,
start_date=today):
    """add data in the complete streak"""
    return self.data.insert_into_streak_table(frequency=frequency, start_day=start_day,
                                              start_year=start_year, start_month=start_month,
                                              start_date=start_date
                                              )
```

Inserts a completed streak entry into the streak tracking table. This method calls insert_into_streak_table in DataManager and logs the frequency and date when the streak was completed. This method is used to track successful habit completion streaks over time.

1.2.7. Method: get_streak_data:

```
def get_streak_data(self, frequency):
    """method to call db to get streak data"""
    return self.data.retrieve_streak_data(frequency=frequency)
```

Retrieves streak data based on the specified frequency (daily or weekly). This method calls retrieve_streak_data in DataManager, which queries the streak table to return data related to the user's completion streaks for the given frequency.

1.3. Class: Analysis:

The Analysis class is designed to analyze habit completion streaks, focusing on both daily and weekly streaks. It has attributes to store the current and longest streaks for daily and weekly completions.

Attributes

- weekly_current_streak: Tracks the current weekly streak of consecutive weeks.
- weekly_longest_streak: Tracks the longest weekly streak recorded.
- daily_current_streak: Tracks the current daily streak of consecutive days.
- daily_longest_streak: Tracks the longest daily streak recorded.

1.3.1. Method: calculate_daily_streak:

```
def calculate_daily_streak(self, dates):
                                # Sort dates in ascending order
    dates = sorted(dates)
        longest_streak = 0
        current_streak = 1  # Start streak count at 1 for the first day
        # Iterate over dates to find consecutive days
        for i in range(1, len(dates)):
            # Check if the current date is consecutive to the previous date
            if dates[i] == dates[i - 1] + timedelta(days=1):
                current_streak += 1
            else:
                # Update longest streak and reset current streak
                longest_streak = max(longest_streak, current_streak)
                current_streak = 1  # Reset for a new streak
        # Final check to update longest streak if it ends on the last date
        longest_streak = max(longest_streak, current_streak)
        self.daily_longest_streak = longest_streak
```

```
    self.daily_current_streak = current_streak
    return longest_streak
```

This method calculates the longest streak of consecutive days based on the dates provided.

1. Sort Dates: The dates list is sorted to ensure chronological order.
2. Loop Through Dates: It iterates through each date to check for consecutive days:
    - If the next date is exactly one day after the previous date, it increments the current_streak.
    - If the next date is not consecutive, it updates the longest_streak and resets current_streak.
3. Final Check: After the loop, it performs a final comparison to update the longest_streak if the longest streak ends on the last date in the list.
4. Update Attributes: Sets self.daily_longest_streak and self.daily_current_streak with the calculated values.

Return: Returns the longest daily streak.

1.3.2. Method: calculate_weekly_streak

```
def calculate_weekly_streak(self, dates):
    # Convert dates to (year, week number) tuples and sort them
    weeks = sorted({(date.year, date.isocalendar()[1]) for date in dates})
    longest_streak = 0
    current_streak = 1  # Start streak count at 1 for the first week
    # Iterate over weeks to find consecutive weeks
    for i in range(1, len(weeks)):
        # Check if the current week is consecutive to the previous week
        prev_year, prev_week = weeks[i - 1]
        curr_year, curr_week = weeks[i]
        # Check if they are consecutive weeks
        if (curr_year == prev_year and curr_week == prev_week + 1) or \
                (curr_year == prev_year + 1 and prev_week == 52 and curr_week == 1):
            current_streak += 1
        else:
            # Update longest streak and reset current streak
            longest_streak = max(longest_streak, current_streak)
            current_streak = 1  # Reset for a new streak
    # Final check to update longest streak if it ends on the last date
    longest_streak = max(longest_streak, current_streak)
```

self.weekly_current_streak = current_streak

self.weekly_longest_streak = longest_streak

return longest_streak

This method calculates the longest streak of consecutive weeks based on the dates provided.

1. Convert to Week Tuples: It first converts each date into a tuple of (year, week_number) and sorts them.
2. Loop Through Weeks: Iterates through the weeks to check for consecutive weeks:
    - If the next week is consecutive (the same year and the next week number or transitioning from week 52 of the previous year to week 1 of the current year), it increments current_streak.
    - If the next week is not consecutive, it updates longest_streak and resets current_streak.
3. Final Check: After the loop, it performs a final comparison to update the longest_streak if necessary.
4. Update Attributes: Sets self.weekly_longest_streak and self.weekly_current_streak with the calculated values.

Return: Returns the longest weekly streak.

2. Flask Routes

1. Login Route (/):
    - Displays the login form.
    - Validates user credentials and logs in a user.
2. Daily Habit Route (/daily_habit):
    - Shows the daily habits, both current and completed.
    - Calculates and displays current and longest daily streaks.
    - Automatically adds a new entry to the streak table at the end of the day if all habits are completed.
3. Weekly Habit Route (/weekly_habit):
    - Shows weekly habits, both current and completed.
    - Calculates and displays current and longest weekly streaks.
    - Automatically adds a new entry to the streak table if all weekly habits are completed at the end of the week (Sunday night).
4. Dashboard Route (/dashboard):
    - Displays a simple dashboard as the main page after logging in.
5. Add Habit Routes (/add_daily_habit and /add_weekly_habit):

- Allows users to add new habits by submitting a form with the habit's name and frequency (daily or weekly).
- Redirects back to the habit page upon success.

6. Check-Off Habit Route (/checkoff):
   - Updates the status of a habit to "completed" for the current day/week.
   - Redirects to the appropriate habit view (daily or weekly).

7. Streak Logic:
   - Daily Streak: If all daily habits are completed, the app inserts a new entry in the streak table to maintain streak data.
   - Weekly Streak: If all weekly habits are completed by the end of Sunday, it inserts a new entry for the weekly streak.

3. Conclusion

Flask-based habit tracking application serves as a useful tool for users to manage and analyze their daily and weekly habits. The app successfully incorporates several key features, including user authentication, habit creation and tracking, and streak calculation. By utilizing PostgreSQL for data persistence, Flask-Login for secure user sessions, and structured classes for habit, user, and analysis management, this application provides a foundational system for habit tracking that is functional and accessible.

This project demonstrates a well-organized architecture with Flask and showcases Python's flexibility in building practical applications. The current implementation allows users to monitor their habits over time, building streaks for continued motivation. Additionally, the separation of concerns across different classes makes the code maintainable and extensible, paving the way for future improvements.

3.1. Suggestions for Improvement

While the habit tracking app is fully functional, there are areas where enhancements could further enrich user experience and improve functionality:

3.1.1. Monthly Habit Tracking:

The current application supports daily and weekly habits. Adding support for monthly habits would provide users with more flexibility and a wider range of tracking options, catering to users with longer-term goals.

3.1.2. Preventing Empty Habit Entries:

Currently, users can add habits with empty or blank names, which could lead to confusing entries in the habit list. Adding a validation step to prevent empty habit entries would improve the clarity of the habit list and overall user experience.

3.1.3. Improved Streak Tracking and Analytics:

Expanding the analytics module to include additional insights, such as average streak lengths, streak trends, and missed habit tracking, could provide users with a more in-depth understanding of their progress over time.

3.1.4. Habit Notifications and Reminders:

Introducing a notification system to remind users of their daily or weekly habits would help users maintain consistency. This could be achieved by integrating push notifications or email reminders.

Reference

Axelos. (2019). *ITIL Foundation, ITIL 4 edition*. The Stationery Office.

Flask. (n.d.). *Flask documentation*. Retrieved from https://flask.palletsprojects.com/en/2.0.x/

Psycopg2. (n.d.). *Psycopg documentation*. Retrieved from https://www.psycopg.org/docs/

Salaets, C. (2020). *The strategic importance of IT in modern business environments*. Journal of Information Systems, 18(2), 35-42.

Schmidt, H. (2020). *Service value systems in ITIL 4: A recipe for effective IT service management*. ITIL Journal, 7(1), 22-34.