Documentation for AssignmentPlanner

by Naing Lin Tun

13th December 2018

Contents

1 Introduction

AssignmentPlanner, by Team Mango, is a web-based application that offers users to organize their school assignments. It is built upon a popular web framework called Flask, which is simple and easy to use. Unlike normal planner applications such as Microsoft Planner, the features of *AssignmentPlanner* are tailored towards students. For instance, it enables the user to add information such as Course Number that normal planner applications do not offer. Through AssignmentPlanner, we aim to have a user-friendly platform where students can access their planner anywhere as long as they have access to the internet. This way, we hope the application allows students to easily, effectively, and efficiently record their assignments.

2 User Interface

Upon landing on the website's main page, the user will have an option to either create an account or login to their existing account. The user does not need third party addresses or accounts such as email or facebook account to create an account. Below is an image of what the landing page looks like for *AssignmentPlanner*:

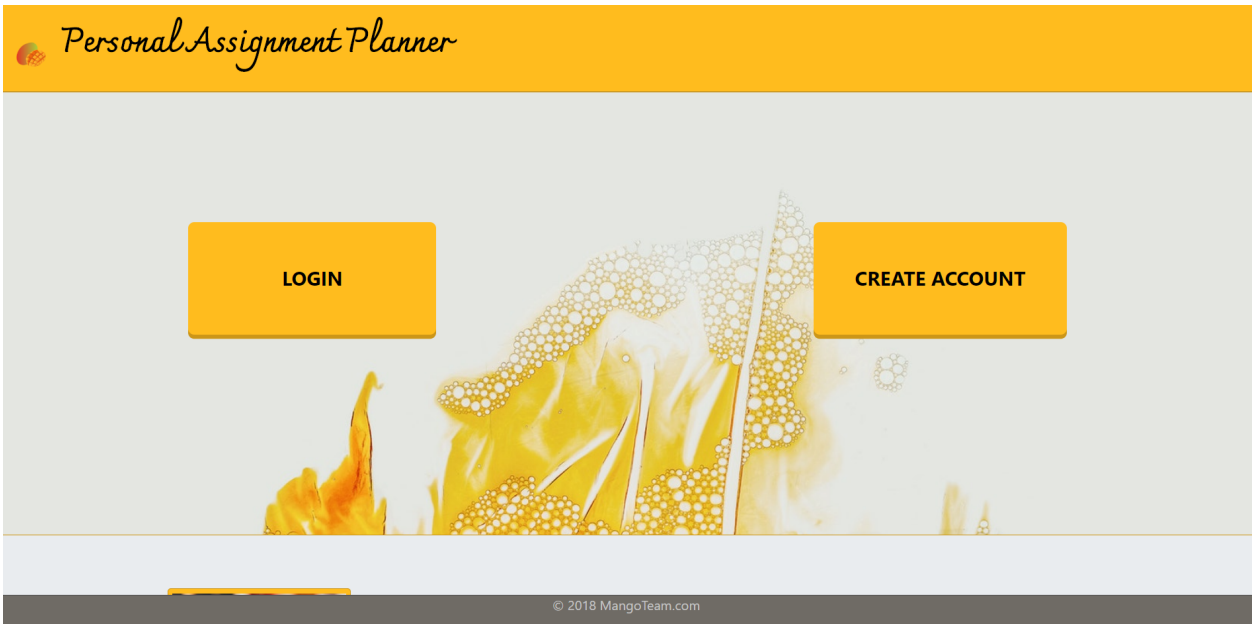

Figure 1. Root directory of the website

After successful creation of an account, the user is redirected to the login page. After the user has logged in, all the pages from here onwards will have a sticky navgiation bar which includes the following buttons:

- **Home**: this redirects the user to their dashboard or homepage

- **Add Assignment**: it redirects the user to the page where he can add assignment
- **View Schedule**: it shows the user the list of all assignments that he currently has
- **View Calender**: it redirects the user where he is prompted enter the date and year for the calendar he wishes to see. Then, a list of assignments due within that month and year are listed below.
- **Log out**: this logs out the user of current account and redirects the user back to the login page.
- **Create Account**: it lets the user to create an account

2.1 Homepage

The homepage of the website displays the user with an overview information about their planner. It states how many assignments in each priority category (Critical, High, Normal, Low) and the due dates. Below is the picture of the website's homepage

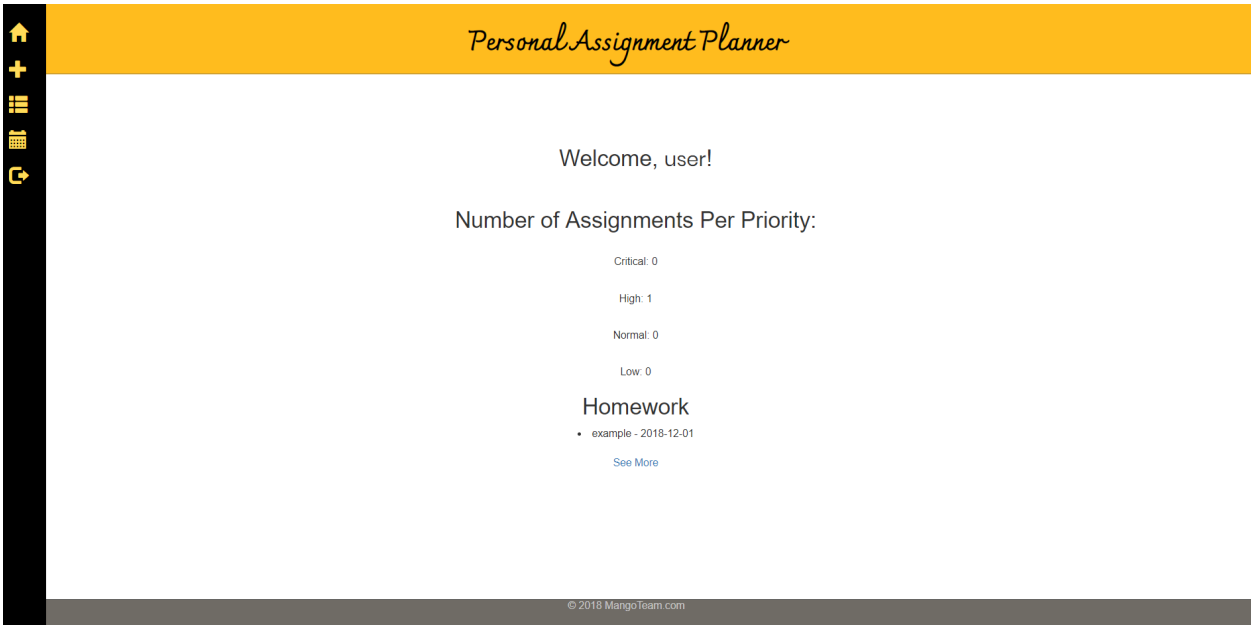

Figure 2. Homepage

2.2 Add Assignment Page

The `Add Assignment` button (a plus icon on the left) lets the user add the assignment entry. The user is presented with the following fields:

- **Title**: the user can add the title of the assignment, such as "Presentation about Cats"
- **Course**: the user can add course number such as "CS253"

- **Category**: the user can specify category such as presentation, report, exam, etc
- **Priority**: this presents the user with four priority levels in a drop down menu in ascending order `Critical`, `High`, `Normal`, `Low`. This way, the user can rank assignments in order of significance – a feature that usual planner apps don't offer. By default, the priority is set to `NORMAL`.
- **Due Date**: the user can either type in the date or select the date from the drop-down calender menu. Needless to say, due dates are very important for assignments.
- **Description**: this lets user add a brief description about the assignment or the note to themselves. For instance, the assignment title "Take Home Exam 2" might not be very indicative about what the assignment is. This is when the description field comes in handy. The user can add descriptions or notes such as "from Chapter 2-7" or "pdf submission instead of hand written".

Below is an image of what the `Add Assignment` page looks like:

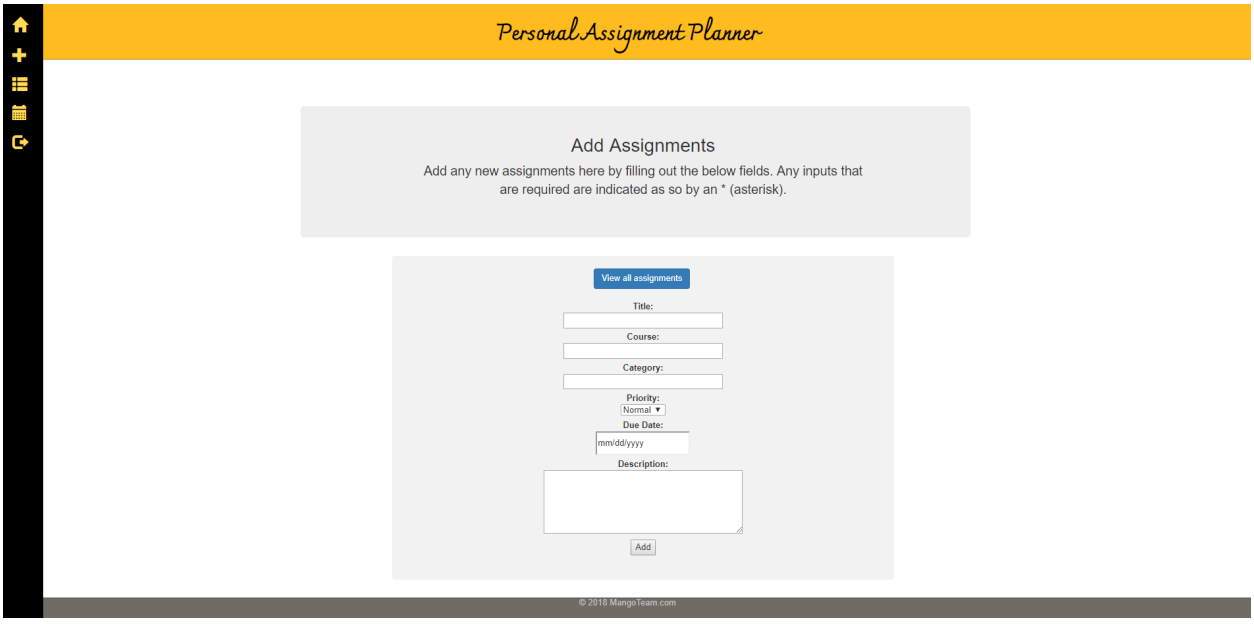

Figure 3. Add Assignment Page

Upon clicking the `Add` button, the new assignment is added, and the user is redirected to `View Schedule`, which is explained below.

2.3 View Schedule Page

The `View Schedule` page displays list of all assignments that the user has in a tabular format. Beside the title of each column, there are down and up arrows, which lets the user sort the assignments by ascending and descending order of that column. In addition, there

is a drop down menu which contains all the due dates of the assignments. The user can either choose all to view all assignments or choose one date to selectively view assignments that are due on that date only.

There are also three buttons beside each assignment entry, which has the following functionalists:

- **Edit**: it brings the user to the page that is similar to `Add Assignment`, but allows the user to edit their assignment data
- **Delete**: it lets the user delete the assignment from the same column
- **Full View**: sometimes, the user may have long texts in the `Description` field. `Show Assignment` page truncates those long text into a short one and appends with `...` at the end. If the user wants to view the full long description, they can click on the `Full View` button to do so

Below is an image of `View Schedule` page:

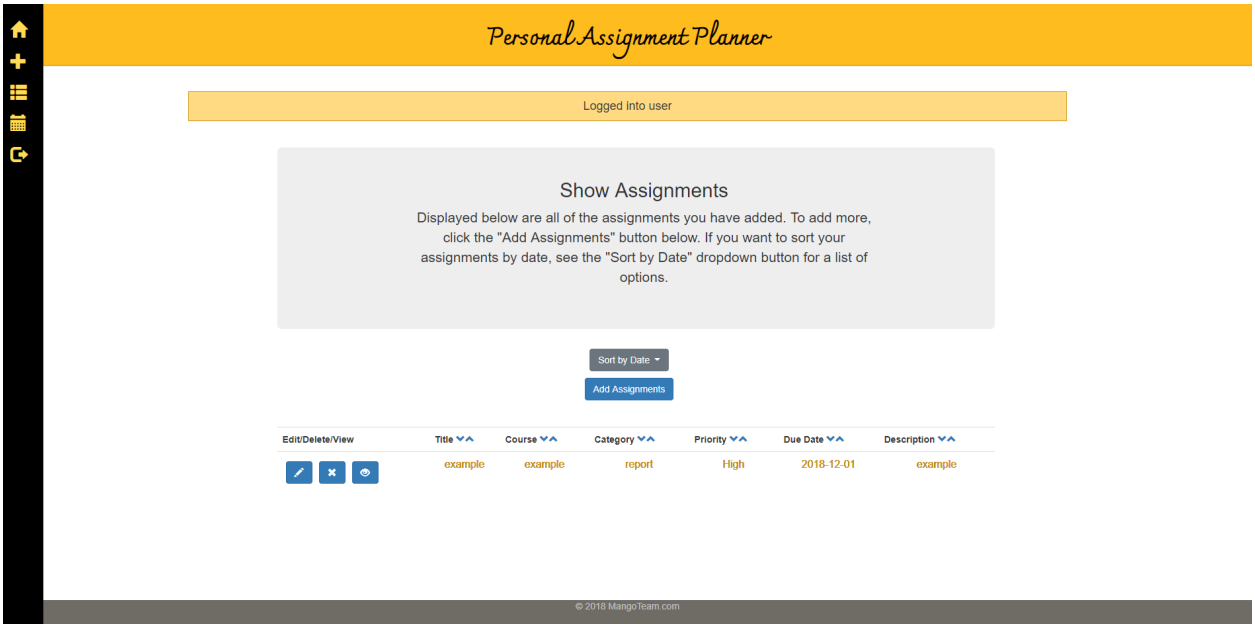

Figure 4. View Schedule Page

2.4 View Calendar Page

`View Calendar` page lets the user enter a numeric (month, year) and displays the calendar of that (month, year). Below the calendar is also the table of assignments whose due date are withink that month, year. Below is an image of what `View Calendar` looks like:

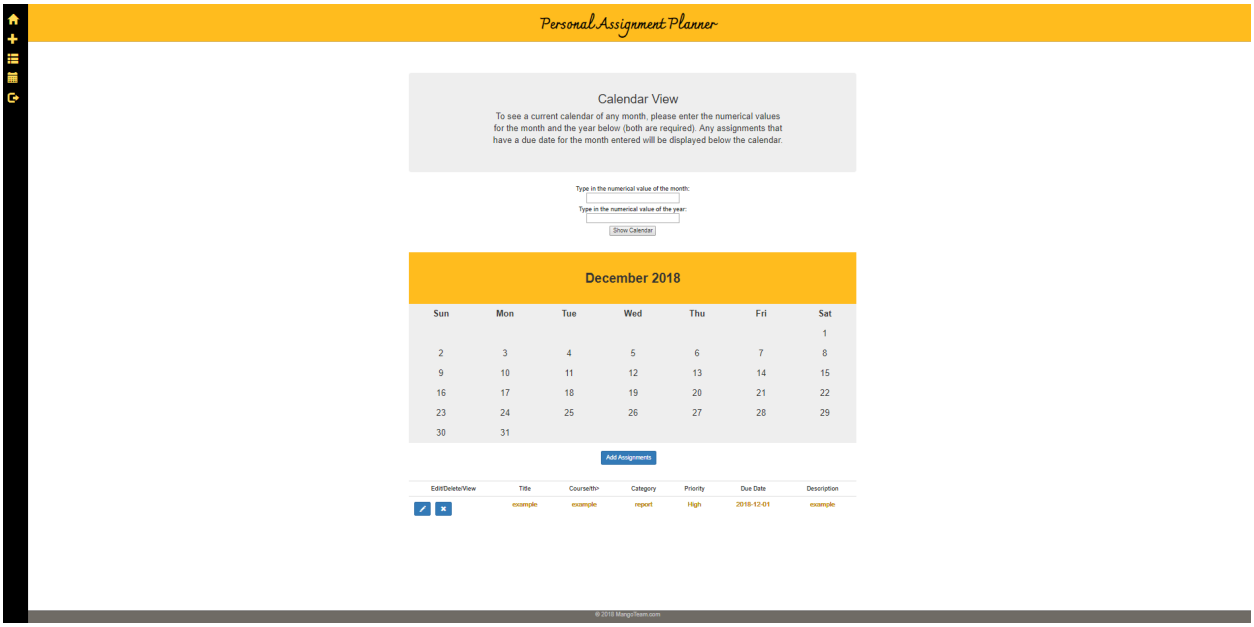

Figure 5. View Calendar Page

Users could find this feature useful when they want to filter out the assignment by (month, year), or if they want to know what assignments are due in a particular (month, year), they could easily find it out this way.

3 Installation and Dependencies

Because Flask web framework is based on Python, it is necessary to install Python in your local machine. We use Python3 for *AssignmentPlanner* codes. You can download different versions of Python from their official website.

Afterwards, installation of Flask is also necessary in order to use Flask framework. The official website for Flask mentions the installation steps, how to run Flask, and more. In short, we have to use the following command to install Flask: `pip install Flask`. Jinja templating is also used in our code. Jinja's official documentation provides a detailed description of syntaxes and more.

Needless to say, a browser is necessary in order to test and run a web-based application such as *AssignmentPlanner*. Our code is run and tested on modern browsers such as Chrome, Mozilla, and Safari (for MacOS). We have not tested our code with old browsers such as Internet Explorer, and we recommend testing and running *AssignmentPlanner* on modern browsers as well.

4 Overview of the architecture of *AssignmentPlanner*

4.1 Databases

AssignmentPlanner uses two tables: one for storing accounts; and another for storing assignments. The table to store accounts has column `id` as primary key, `username`, and `password`. The table for assignments has the following columns: `id` as primary key; `username`; `title` of the assignment; `course` number (such as CS253); `duedate`; and `description`. All these tables are implemented in `schema.sql` file in SQLite as follows:

```
drop table if exists accounts;
create table accounts (
id integer primary key autoincrement,
username text not null,
password text not null
);

drop table if exists assignments;
create table assignments (
id integer primary key autoincrement,
username text not null,
title text not null,
course text not null,
category text not null,
duedate text not null,
description text not null
);
```

The `username` column in `assignments` table is present for the purpose of connecting each account to their relative assignments. So, whenever the user adds an assignment (while logged in), the assignment is added to the `assignments` table along with his `username`. The data from `username` column never appears in any part of user interface; it is just for the sake of linking each user with their data.

4.2 Function Interfaces

As mentioned before, the backend of *AssignmentPlanner* is handled by Flask web framework written in Python. The applicaiton has the following functions:

- `create_account()`: it creates an account for the user, along error-checking such as the user cannot create an account with an already-existing username, and so on.

- `login_account()`: it enables the user to login to the account that he has created. It aso does error-checking against `accounts` database whether the input `username` and `password` matches.

- `logout()`: as the name says, this function is for logging out of the user's current account.

- `add_assignment()`: it lets the user add the assignment by filling in the data of the assignment and clicking on the `Add Assignment` button

- `show_assignment()`: by default, this will show all assignments in the `assignments` table with the `username` of the account that is currently logged in. "Sort arrow" buttons are also displayed at the top of each column header. Upon clicking those sort buttons, the respective sort features are also triggered, which is included in the `show_assignment()` code.

- `del_assignment()`: every assignment entry, that is displayed by `show_assignment()` function above, has its own delete button beside it. By clicking on that delete button, the corresponding assignment entry gets deleted.

- `edit_entry()`: every assignment entry also has an edit button beside it. Clicking on it redirects the user to the edit page, where they can edit the entries of their assignment. They can either save or discard the changes and get redirected back to `show_assignment()` page.

- `input_calender()`: the user can type in the month and year that he wishes to see. Then, a calender for that month and year is displayed, along with the table that is due for that month.

4.3 Link Directories

Since `AssignmentPlanner` has several subdiretories, it might be useful to list out all of them and their functionalities:

- `/`: this is just the landing page of the website and prompts the user to either login or create account.

- `/login`: log-in page.

- `/signup`: create account page.

- `/logout`: logs out the current user who is already logged in.

- `/add`: add assignment page. The user can fill fields such as title, course number, due date, etc.
- `/assignments`: the page to show all or sorted assignments.
- `/edit?editid=X`: edit the assignment entry whose ID is X.
- `/full_view?id=X` expand all the data fields of an assignment entry whose ID is X.
- `/calendar`: calendar page where the user can generate a calendar based on the month and year, along with the assignments that are due within that whole month.

4.4 Styling and Templating

As we can see from figures 2-5, the header "AssginmentPlanner", the navbar, and the footer remains consistent throughout all the website as soon as the user has logged in. Thus, the codes for those "common features" will be included in a `layout_template.html`, which is as follows: Other HTML files, in turn, extend the template file by putting the Jinja template code `{% extends "layout_template.html" %}` at the top of their files. Thus, many relevant Jquery links and CSS libraries are also called in `layout_template.html` so that its behavior maintains the same throughout the website. The full implementation of `layout_template.html` and its CSS can be found in `AssignmentPlanner` repository.

5 Function Definitions and Test Cases

5.1 Initializing Databases

The code to initialize and interact with the database is included at the start of `app.py` file as follows:

```
1  import os
2  import calendar
3  from sqlite3 import dbapi2 as sqlite3
4  from flask import Flask, request, session, g, redirect, url_for, abort, \
5      render_template, flash
6
7  # create our little application :)
8  app = Flask(__name__)
9
10 # Load default config and override config from an environment variable
11 app.config.update(dict(
12     DATABASE=os.path.join(app.root_path, 'planner.db'),
```

```
    DEBUG=True,
    SECRET_KEY='development key',
))
app.config.from_envvar('FLASKR_SETTINGS', silent=True)


def connect_db():
    """Connects to the specific database."""
    rv = sqlite3.connect(app.config['DATABASE'])
    rv.row_factory = sqlite3.Row
    return rv


def init_db():
    """Initializes the database."""
    db = get_db()
    with app.open_resource('schema.sql', mode='r') as f:
        db.cursor().executescript(f.read())
    db.commit()


@app.cli.command('initdb')
def initdb_command():
    """Creates the database tables."""
    init_db()
    print('Initialized the database.')


def get_db():
    """Opens a new database connection if there is none yet for the
    current application context.
    """
    if not hasattr(g, 'sqlite_db'):
        g.sqlite_db = connect_db()
    return g.sqlite_db


@app.teardown_appcontext
def close_db(error):
    """Closes the database again at the end of the request."""
    if hasattr(g, 'sqlite_db'):
        g.sqlite_db.close()
```

These codes have been adapted from Flask tutorial, steps 3-5. These tutorials provide explanations about each of these commands.

The `calendar` library is imported as well (line 2) because some part of the applications use calendar function, as we will see in Section 5.8.

5.2 Create, Login, and Logout account

In order for the website to function well, the three functions `create`, `login`, and `logout` must go hand in hand. In order for the user to `login`, he has to `create` account first. Once the user has logged in, he should also be able to `logout`. Thus, these three features are intertwined, and their implementations are elaborated below.

5.2.1 Create Account

Create, login, and logout functions are the first ones that we should implement because all other functions depend on these three functions. For instance, in order to add assignment, a user should be logged in first, but in order to login, the user have to create account first. All these three functios are written in Python using Flask web framework.

Below is the implementation of `create_account()`:

```
1  @app.route('/create_account', methods=['POST'])
2  def create_account():
3      db = get_db()
4      validate = db.execute('select username from accounts where
       ↪  username=?', [request.form['username']])
5      data = db.execute('select * from accounts')
6
7      if validate.fetchall():
8          flash('The username already exists. Try with another username')
9          return redirect(url_for('redirect_signup'))
10     else:
11         password = request.form['password']
12         re_password = request.form['password2']
13
14         if password != re_password:
15             flash('Passwords do not match. Try again.')
16             return redirect(url_for('redirect_signup'))
17         else:
18
19             db.execute('insert into accounts (username, password) values
               ↪  (?, ?)',
```

```
                    [request.form['username'], password])
            db.commit()
        flash('Account creation successful.')

    return redirect(url_for('redirect_login'))
```

This `create_account()` account function is called in `CreateAccount.html` file, which is the code for `/signup` page where users can create an account. A relevant excerpt of code from `CreateAccount.html` is as follows:

```
<form action="{{ url_for('create_account') }}" method="POST">

    <div class="input-group form-group">
        <div class="input-group-prepend">
            <span class="input-group-text"><i class="fas
             ↪ fa-user"></i></span>
        </div>
        <label for="Username1">Username:</label>
        <input type="text" name="username" class="form-control"
         ↪ id="Username1" placeholder="Enter Username" required>
    </div>

    <div class="input-group form-group">
        <div class="input-group-prepend">
            <span class="input-group-text"><i class="fas
             ↪ fa-user"></i></span>
        </div>
        <label for="Password1">Password:</label>
        <input type="password" name="password" class="form-control"
         ↪ id="Password1" placeholder="Enter Password" required>
    </div>

    <div class="form-group">
        <div class="input-group-prepend">
            <span class="input-group-text"><i class="fas
             ↪ fa-user"></i></span>
        </div>
        <label for="Password2">Re-enter Password:</label>
        <input type="password" name="password2" class="form-control"
         ↪ id="Password2" placeholder="Re-Enter Password" required>
    </div>

```

```
27      <div class="form-group">
28          <button type="submit" class="btn btn-primary">Submit</button>
29      </div>
30  </form>
```

So, what `CreateAccount.html` does is basically asking the user's input for `username`, `create_account()`, and `re-enter password`. Then, `create_account()` does all the necessary condition checkings for successful account creation. Since It checks for the following two conditions:

- **username already existed**: if the `username` that the user provided is already present in the `accounts` table, the user is notified about it and redirected back to the `/signup` page.
- **passwords do not match**: if the user enters different values for `password` and `re-enter password` field, the user is notified about it and redirected back to the `/signup` page.
- **account creation successful**: only if the user does not face any of the conditions above, his account creation will be successful

Since `create_account()`, upon successful account creation, will add new data into the database table `accounts`, it needs a `POST` method.

This `create_account()` function still needs to be tested for its corrections. Thus, a unit test is implemented for `create_account()` as follows:

```
1   def create(self, username, password1, password2):
2       return self.app.post('/create_account', data=dict(username=username,
        ↪  password=password1, password2=password2)
3           ,follow_redirects=True)
4
5   def test_create(self):
6       rv = self.create("user", "pw", "pw");
7       assert b"Account creation successful." in rv.data
8
9       rv = self.create("user", "pw", "pw");
10      assert b"The username already exists. Try with another username" in
        ↪  rv.data
11
12      rv = self.create("new_user", "pw", "pw_does_not_match")
13      assert b"Passwords do not match. Try again." in rv.data
```

Here, `create()` function is just to create an account, and `test_create()` tests for the correctness of `create_account()` function by checking the flash messages it produce. The test function tests for all possible cases mentioned above.

5.2.2 Login Account

After the user has created an account, he can begin logging in. Below is the implementation of `login_account()` function:

```
@app.route('/login_account', methods=['POST'])
def login_account():
    db = get_db()
    username = request.form['username']
    validate_account = db.execute('select username, password from accounts
    ↪  where username=?', [username])
    data = validate_account
    data = dict(data)

    if db.execute('select username, password from accounts where
    ↪  username=?', [username]).fetchall():
        password = request.form['password']

        if data.get(username) == password:
            session['logged_in'] = username
            flash('Logged into ' + username)
            return redirect(url_for('show_assignment'))

        else:
            flash('Wrong username and password. Try again')

    else:
        flash('Username does not exist')
    return redirect(url_for('redirect_login'))
```

The login function is connected to `Login.html` file, which is the code for `/login page`. A relevant section of the code from `Login.html` is the following:

```
<form action="{{ url_for('login_account') }}" method="POST" >
    <div class="input-group form-group">
        <div class="input-group-prepend">
```

```
4                <span class="input-group-text"><i class="fas
                  ↪  fa-user"></i></span>
5            </div>
6            <label for="Username1">Username</label>
7            <input type="text" name="username" class="form-control"
              ↪  id="Username1" placeholder="Enter username">
8        </div>
9
10       <div class="input-group form-group">
11           <div class="input-group-prepend">
12               <span class="input-group-text"><i class="fas
                  ↪  fa-key"></i></span>
13           </div>
14           <label for="Password1">Password</label>
15           <input type="password" name="password" class="form-control"
              ↪  id="Password1" placeholder="Enter password">
16
17       </div>
18
19       <div class="row align-items-center remember">
20           <input type="checkbox">Remember Me
21       </div>
22
23       <div class="form-group">
24           <button type="submit" class="btn btn-primary">Submit</button>
25       </div>
26   </form>
```

Similar to `CreateAccount.html`, `Login.html` is just asking the user for its input on `username` and `password`. The `login_account()` function pulls the data from `accounts` table to see whether the input `username` and `password` corresponds to one (and only one) of the rows in the table. It checks for the following three conditions:

- **username not in database**: if the username is not in `accounts` table, the user is notified about it and redirected back to `/login` page
- **username in database, but not the correct password**: it notifies the user that the username and password do not match. Then, the user is redirected back to the `/login` page
- **login successful**: only if the user login does not meet all the conditions above, login account will be successful.

The reason there are two different error conditions is to give the user more information about the error message that they receive. Here, `POST` method is used for security reasons. `GET` could have been used here and still works fine, but the inputs from `GET` methods are passed in URL. For instance, the user's login (username: admin; password: admin) will be sent as `/login?username=admin&password=admin` if `login_account()` is implemented with `GET` method. This is not secure for passwords because everyone looking at the screen could see the password, and also obtaining password via sniffing is also easier. With `POST` method, all the data are passed into HTTP messages, not URL.

In order to ensure that the user stays logged in after he has pressed "Login" button, the code uses Flask's inbuilt `session` features. Sessions are basically secure cookies that are used to indicate whether the client (user) is logged in or logged out. Flask's official documentation about sessions provides a more detailed description. The reason `session['logged_in']` is set to equal the input string `username` is because if the application wants to access the `username` of the current session (outside the scope of `login_account()` function) later on, it could do that by just calling `session['logged_in']` code. This is done in several functions such as `add_assignment()` (see 5.3), where it also adds `username` into the `accounts` table.

The unit test function for `logout_account()` is as follows:

```
1  def login(self, username, password):
2      return self.app.post('/login_account', data=dict(username=username,
       ↪  password=password), follow_redirects=True)
3
4  def test_login(self):
5      self.create("user", "pw", "pw")
6      rv = self.login("user", "pw")
7      assert b"Logged into user" in rv.data
8
9      rv = self.login("user_dne", "pw")
10     assert b"Username does not exist" in rv.data
11
12     rv = self.login("user", "wrong_pw")
13     assert b"Wrong username and password. Try again" in rv.data
```

The `login()` function is just for logging the user into the account. The `test_login()` function tests for all three possible cases as outlined above. Note that `self.create()` is called first before logging in because the user cannot login to the non-existent account; he has to create an account first in order to login.

5.2.3 Logout Account

After the user has logged in, of course, the user should be able to log out of their account (session) as well. The implementation of `logout` is relatively shorter and is as follows:

```
@app.route('/logout')
def logout():
    session.pop('logged_in', None)
    flash('You were logged out')
    return redirect(url_for('redirect_login'))
```

This `logout` function is also tied to the logout button on the navbar, where it gets triggered on click. The navbar implementation can be found in `layout_template.html`, and the code for logout button is as follows:

```
<a href="{{ url_for('logout') }}" title="Log Out">
    <span class="glyphicon glyphicon-log-out"></span>
</a>
```

Recall that when the user logs in, `login_account()` function sets $< key, value >$ pair to $<$ '*logged_in*', *username* $>$ and adds it to `session`. Thus, when the user logs out, that $<$ '*logged_in*', *username* $>$ pair should be removed (popped) from `session`, and that is exactly what `logout()` function does. In addition, it also lets the user know, via flash massage, that he is logged out and redirected to the login page.

The unit test implementation for logout is shown below:

```
def test_logout(self):
    self.create("user", "pw", "pw")
    self.login("user", "pw")
    rv = self.app.get('/logout', follow_redirects=True)
    assert b"You were logged out" in rv.data
```

Before the user can logout, the user first has to create an account and login to that account. That's why `self.create()` and `self.login()` are called first. Then, it checks whether the correct flash message is printed or not. Notice that there is `follow_redirects=True` when calling `/logout`. This is because the `logout()` function redirects the user back to the login page. Thus, the redirect part is necessary for the `self.app.get` call.

5.3 Add Assignment

As stated in Section 2.2, `Add Assginment` page lets the user add their assignments one at the time (while logged in). Now, we take a look at its implementation.

```
@app.route('/add', methods=['POST'])
def add_assignment():
    if not session.get('logged_in'):
        abort(401)

    db = get_db()
    db.execute('insert into assignments (username, title, course,
     ↪  category, priority, duedate, description) '
               'values (?, ?, ?, ?, ?, ?, ?)', [session['logged_in'],
                ↪  request.form['title'], request.form['course'],
                request.form['category'], request.form['priority'],
                 ↪  request.form['duedate'], request.form['description']])
    # request.form gets request in a post request
    # Puts the values from the show_entries.html form into the database
     ↪  as (title, category, text)
    db.commit()
    # Commits it to the database
    flash('New assignment was successfully saved.')
    return redirect(url_for('show_assignment'))
```

This `add_assignment()` function is linked to `AddAssignment.html`. Below is the section of relevant code from `AddAssignment.html`:

```
<form action="{{ url_for('add_assignment') }}" method="post">
    <dl>
        <dt>Title:
        <dd><input type="text" size="30" name="title">
        <dt>Course:
        <dd><input type="text" size="30" name="course">
        <dt>Category:
        <dd><input type="text" size="30" name="category">
        <dt>Priority:
        <dd><select name="priority">
            <option value="Critical">Critical</option>
            <option value="High">High</option>
            <option value="Normal" selected>Normal</option>
```

```
14              <option value="Low">Low</option>
15          </select>
16          <dt>Due Date:
17          <dd><input type="date" name="duedate">
18          <dt>Description:
19          <dd><textarea name="description" rows="5" cols="40"></textarea>
20          <dd><input type="submit" value="Add">
21      </dl>
22  </form>
```

What `AddAssignment.html` does is just asking user input for assignment data and upon clicking `Add` button, `add_assignment()` function gets called. The `abort(401)` is in `add_assignment()` function, line 4 just to make sure that the user cannot add an assignment while they are not logged in. The rest of the code is just basically adding the user input into the `assignments` table. When the user has finished adding the assignment, he is redirected to `View Assignment` page, where he can see all of his assignments. Details about `View Assignment` is described in next section and also in Section 2.3.

The test function for `add_assignment()` function is as follows:

```
1   # modified login() code from the following source
2   # http://flask.pocoo.org/docs/0.12/testing/
3   def add_entry(self, title, course, category, priority, duedate,
    ↪  description):
4       return self.app.post('/add',
5           data=dict(title=title, course=course, category=category,
            ↪  priority=priority,
6                     duedate=duedate, description=description),
                      ↪  follow_redirects=True)
7
8   def test_add_entry(self):
9       rv = self.add_entry('title1', 'CS253', 'None', 'High',
        ↪  '1111-11-11T11:11', 'D1')
10      assert b"Unauthorized" in rv.data
11
12      self.create("user", "pw", "pw")
13      self.login("user", "pw")
14      rv = self.add_entry('title1', 'CS253', 'None', 'High',
        ↪  '1111-11-11T11:11', 'D1')
15      assert b"title1" in rv.data
16      assert b"New assignment was successfully saved." in rv.data
```

First, the above code tests for the case when the user tries to add an assignment without logging into his account. Because of the `abort(401)` line in `add_assignment()` function, the page will print "Unauthorized" instead. Next, it tests for the case when the user logged in and add the assignment. If the input data appears on the page after adding it, then the `add_assignment()` is working correctly.

5.4 View Schedule

The `View Schedule` page, described in 2.3, displays all the assignment that the user currently has (while logged in). Below is its implementation:

```
@app.route('/assignments')
def show_assignment():

    if 'logged_in' in session:
        db = get_db()
        username = session['logged_in']

        if "duedate" in request.args:
            cur = db.execute('select * from assignments where username = ?
             ↪  and duedate = ? order by id desc',
                             [username, request.args["duedate"]])
            assignments = cur.fetchall()

        elif "arrange" in request.args:

            cur = db.execute('select * from assignments where username = ?
             ↪  order by {} ASC'.format(request.args["arrange"]),
                             [username])

            assignments = cur.fetchall()

        elif "sort" in request.args:
            cur = db.execute('select * from assignments where username = ?
             ↪  order by {} DESC'.format(request.args["sort"]),
                             [username])

            assignments = cur.fetchall()

        else:
```

```
27
28              cur = db.execute('select * from assignments where username = ?
                ↪  order by id desc', [username])
29              assignments = cur.fetchall()
30
31          cur = db.execute('select distinct duedate from assignments where
            ↪  username = ? order by duedate asc', [username])
32
33
34          duedates = cur.fetchall()
35          return render_template('ShowAssignments.html',
            ↪  assignments=assignments, duedates=duedates)
36
37      return render_template('Login.html')
```

Recall from 2.3 that `View Schedule` page also contains sorting functionalities: with up and down arrow at each column; and also "filtering by due date". All of these functions – in addition to displaying all assignment – are combined in `show_assignment` function. In this code, note that "arrange" means sort in ascending order, and "sort" means sort in descending order.

Line 7-10, 32 deals with "filtering by due date" function. It is linked to the following code from `ShowAssignments.html`

```
1   <ul class="duedates">
2       <div class="dropdown extrapadding">
3           <button class="btn btn-secondary dropdown-toggle" type="button"
            ↪  id="dropdownMenuButton" data-toggle="dropdown"
            ↪  aria-haspopup="true" aria-expanded="false">
4               Sort by Date
5           </button>
6           <div class="dropdown-menu" aria-labelledby="dropdownMenuButton">
7               <a class="dropdown-item" href="?">[all]</a>
8               {% for duedate in duedates %}
9                   {% if '' not in duedate %}
10                      <a class="dropdown-item" href="?duedate={{
                        ↪  duedate.duedate }}">{{ duedate.duedate }}</a>
11                  {% endif %}
12              {% endfor %}
13          </div>
14      </div>
15  </ul>
```

The following code, from `ShowAssignments.html`, is for showing the table of assignments:

```
{% if assignments %}
    <table class="table">
        {% if arrange == "title" %}
            <a href="./?arrange=title"></a>
        {% endif %}
        {% if arrange == "course" %}
            <a href="./?arrange=course"></a>
        {% endif %}
        {% if arrange == "category" %}
            <a href="./?arrange=category"></a>
        {% endif %}
        {% if arrange == "priority" %}
            <a href="./?arrange=priority"></a>
        {% endif %}
        {% if arrange == "duedate" %}
            <a href="./?arrange=duedate"></a>
        {% endif %}
        {% if arrange == "description" %}
            <a href="./?arrange=description"></a>
        {% endif %}

        {% if sort == "title" %}
            <a href="?sort=title"></a>
        {% endif %}
        {% if sort == "course" %}
            <a href="?sort=course"></a>
        {% endif %}
        {% if sort == "category" %}
            <a href="?sort=category"></a>
        {% endif %}
        {% if sort == "priority" %}
            <a href="?sort=priority"></a>
        {% endif %}
        {% if sort == "duedate" %}
            <a href="?sort=duedate"></a>
        {% endif %}
        {% if sort == "description" %}
            <a href="?sort=description"></a>
```

```
39          {% endif %}
40
41          <tr>
42              <th>Edit/Delete </th>
43
44              <th>Title <a href="/assignments?sort=title"><span
                ↪  class="glyphicon glyphicon-chevron-down
                ↪  arrow"></span></a><a
                ↪  href="/assignments?arrange=title"><span class="glyphicon
                ↪  glyphicon-chevron-up arrow"></span></a></th>
45              <th>Course <a href="/assignments?sort=course"><span
                ↪  class="glyphicon glyphicon-chevron-down
                ↪  arrow"></span></a><a
                ↪  href="/assignments?arrange=course"><span class="glyphicon
                ↪  glyphicon-chevron-up arrow"></span></a></th>
46              <th>Category <a href="/assignments?sort=category"><span
                ↪  class="glyphicon glyphicon-chevron-down
                ↪  arrow"></span></a><a
                ↪  href="/assignments?arrange=category"><span
                ↪  class="glyphicon glyphicon-chevron-up
                ↪  arrow"></span></a></th>
47              <th>Priority <a href="/assignments?sort=priority"><span
                ↪  class="glyphicon glyphicon-chevron-down
                ↪  arrow"></span></a><a
                ↪  href="/assignments?arrange=priority"><span
                ↪  class="glyphicon glyphicon-chevron-up
                ↪  arrow"></span></a></th>
48              <th>Due Date <a href="/assignments?sort=duedate"><span
                ↪  class="glyphicon glyphicon-chevron-down
                ↪  arrow"></span></a><a
                ↪  href="/assignments?arrange=duedate"><span class="glyphicon
                ↪  glyphicon-chevron-up arrow"></span></a></th>
49              <th>Description <a href="/assignments?sort=description"><span
                ↪  class="glyphicon glyphicon-chevron-down
                ↪  arrow"></span></a><a
                ↪  href="/assignments?arrange=description"><span
                ↪  class="glyphicon glyphicon-chevron-up
                ↪  arrow"></span></a></th>
50
51          </tr>
52
53          {% for assignment in assignments %}
```

```
54                {% if assignment.priority == "Critical" %}
55                    <tr class="critical">
56                        {% elif assignment.priority == "High": %}
57                    <tr class="high">
58                        {% elif assignment.priority == "Normal": %}
59                    <tr class="normal">
60                        {% elif assignment.priority == "Low": %}
61                    <tr class="low">
62                        {% else %}
63                    <tr>
64                {% endif %}
65            <td>
66                <form action="{{ url_for('edit_entry') }}" method="get"
                  ↪ class="inline">
67                    <button type="submit" class="btn btn-primary glyphicon
                      ↪ glyphicon-pencil"></button>
68                    <input type="hidden" value="{{ assignment.id }}"
                      ↪ name="editid">
69                </form>
70                <form action="{{ url_for('del_assignment') }}" method="post"
                  ↪ class="inline">
71                    <button type="submit" class="btn btn-primary glyphicon
                      ↪ glyphicon-remove"></button>
72                    <input type="hidden" value="{{ assignment.id }}"
                      ↪ name="id">
73                </form>
74                <form action="{{ url_for('full_view') }}" method="get"
                  ↪ class="inline">
75                    <button type="submit" class="btn btn-primary glyphicon
                      ↪ glyphicon-eye-open"></button>
76                    <input type="hidden" value="{{ assignment.id }}"
                      ↪ name="id">
77                </form>
78            </td>
79
80            <td>{{ assignment.title }}</td>
81            <td>{{ assignment.course }}</td>
82            <td>{{ assignment.category }}</td>
83            <td>{{ assignment.priority }}</td>
84            <td>{{ assignment.duedate }}</td>
85            <td>{{ assignment.description | truncate(25) | safe }}</td>
86            </tr>
```

```
87            {% else %}
88                <em>No assignment entries here so far.</em>
89                <br><br>
90            {% endfor %}
91
92        </table>
93    {% else %}
94        <em>You don't have any assignments currently.</em>
95        <br><br>
96    {% endif %}
```

Line 3-49 are for handling "sort arrows" and its functions. Note that there are 12 sorting arrows in total, and we do not want 12 different sorting functions. We also cannot pass `order by x`, where `x` is a variable, in SQL statement inside `show_assignments()` code. So, one option to get around it is to use Python string formating `.format()` and do `'... order by {} ...'.format(request.args[_some value_])`. Thus, we can just have a general sorting SQL statement (the `order by` code) with different variables passed in as `.format()`. That's why `.format()` is used for `show_assignment()` code.

Line 66-78 are for `Edit`, `Delete`, `Full View` buttons at each assignment row. The rest are for showing all assignments in a tabular form.

The test case for `show_assignment()` function is split into two parts:
`test_show_assignment()` tests whether all assignments are displayed in the table; and `test_sort()` tests whether *all* sorting arrows are working properly.

Below is the implementation of `test_show_assignment()`:

```
1   def test_show_assignment(self):
2       self.create("user", "pw", "pw")
3       self.login("user", "pw")
4
5       self.add_entry('title1', 'CS253', 'None', 'High', '1111-11-11T11:11',
        ↪  'D1')
6       self.add_entry('title2', 'CS253', 'None', 'High', '1111-11-11T11:11',
        ↪  'D2')
7       rv = self.app.get('/assignments')
8       assert b"title1" in rv.data
9       assert b"title2" in rv.data
```

This one is rather simple: first, we add a few assignments (while logged in); and secondly, we go to the `/assignments` route to see whether all assignments appear in the table. If it does, then `test_show_assignment()` will pass.

The implementation of `test_sort()` is rather long because there are 12 arrows in total, and the unit test has to exhaust through all cases. But the idea is simple: first, add a few posts with different column values; then, apply arranging/sorting function and checks whether all assignments appear in order. Its full implementation can be found in `planner_tests.py` in the repository.

5.5 Edit Assignment

As mentioned in the previous section, there is an `Edit` button beside each assignment entry. Upon clicking the button, it redirects the user to the edit page where he can modify or update the assignment data as desired. The implmenetation of edit feature is as follows:

```
1  @app.route('/edit', methods=['GET'])
2  def edit_entry():
3      db = get_db()
4      cur = db.execute('select * from assignments where id = ?',
       ↪  [request.args['editid']])
5      assignments = cur.fetchall()
6      return render_template('EditAssignment.html', assignments=assignments)
7
8
9  @app.route('/edit_assignment', methods=['POST'])
10 def update_entry():
11     if 'logged_in' in session:
12         db = get_db()
13         theid = request.form['id']
14         title = request.form['title']
15         course = request.form['course']
16         category = request.form['category']
17         priority = request.form['priority']
18         duedate = request.form['duedate']
19         description = request.form['description']
20         db.execute('update assignments set title = ?, course = ?, category
           ↪  = ?, priority = ?, duedate = ?, description = ?'
21             'where id = ?', [title, course, category, priority, duedate,
               ↪  description, theid])
22         db.commit()
23         # Commits it to the database
```

```
24            flash('New entry was successfully edited')
25            return redirect(url_for('show_assignment'))
26        return render_template('Login.html')
```

The `edit_entry()` function deals with redirecting the user to the edit page upon clicking the `Edit` button. Each `Edit` button has its value as the assignment's key, the `id`. So, when the user clicks on the `Edit` button, `edit_entry()` selects the assignment whose `id` matches with the button's value. Its implementation can be found in line 66-69 of `ShowAssignments.html` code mentioned above.

The `update_entry()` function handles user input data from the edit page and updates the assignment entry with the new data. After updating, the user is redirected back to the The `View Schedule` page. The corresponding HTML code from `EditAssignment.html` is shown below:

```
1   <form action="{{ url_for('update_entry') }}" method="post"
     ↪  class="update-entry">
2       <dl>
3           <dd><input type="hidden" value="{{ assignment.id }}"
             ↪  name="id"></dd>
4           <dt>Title:
5           <dd><input type="text" size="30" name="title" placeholder="{{
             ↪  assignment.title }}"
6                   value="{{ assignment.title }}">
7           <dt>Course:
8           <dd><input type="text" size="30" name="course" placeholder="{{
             ↪  assignment.title }}"
9                   value="{{ assignment.course }}">
10          <dt>Category:
11          <dd><input type="text" size="30" name="category" placeholder="{{
             ↪  assignment.title }}"
12                  value="{{ assignment.category }}">
13          <dd>
14              {% if assignment.priority == 'Critical' %}
15              <select name="priority">
16                  <option value="Critical" class="selected"
                     ↪  selected>Critical</option>
17                  <option value="High">High</option>
18                  <option value="Normal">Normal</option>
19                  <option value="Low">Low</option>
20              </select>
```

```
21
22              {% elif assignment.priority == 'High' %}
23              <select name="priority">
24                  <option value="Critical" class="selected"
                     ↪  selected>Critical</option>
25                  <option value="High" class="selected"
                     ↪  selected>High</option>
26                  <option value="Normal">Normal</option>
27                  <option value="Low">Low</option>
28
29              </select>
30
31              {% elif assignment.priority == 'Normal' %}
32              <select name="priority">
33                  <option value="Critical" class="selected"
                     ↪  selected>Critical</option>
34                  <option value="High">High</option>
35                  <option value="Normal" class="selected"
                     ↪  selected>Normal</option>
36                  <option value="Low">Low</option>
37              </select>
38
39              {% else %}
40              <select name="priority">
41                  <option value="Critical" class="selected"
                     ↪  selected>Critical</option>
42                  <option value="High">High</option>
43                  <option value="Normal">Normal</option>
44                  <option value="Low" class="selected" selected>Low</option>
45              </select>
46
47              {% endif %}
48          <dt>Due Date:
49          <dd><input type="date" size="30" name="duedate" placeholder="{{
             ↪  assignment.title }}"
50                     value="{{ assignment.duedate }}">
51          <dt>Description:
52          <dd><textarea name="description" rows="5" cols="40"
53                        placeholder="{{ assignment.description }}">{{
                           ↪  assignment.description }}</textarea>
54          <dd><input type="submit" value="Confirm Changes">
55      </dl>
```

```
56  </form>
```

The original data will be pre-populated for each fields, so that the user can see what the original data is on the same page (i.e. editing page). Not only that, the original data are also set as a "placeholder" in each input field, so that even if the user erases all the original data, he can still see what the original data is on the editing page (otherwise, the user would have to go back to `View Schedule` page, which is a hassle). The purpose of all these pre-populated and placeholder features is to improve the website's useability and user's friendliness.

The test case for `update_entry()` is implemented as follows:

```
1   def edit_entry(self, title, course, category, priority, duedate,
     ↪  description, id):
2       self.create("user", "pw", "pw")
3       self.login("user", "pw")
4       return self.app.post('/edit_assignment',
5           data=dict(title=title, course=course, category=category,
             ↪  priority=priority,
6                     duedate=duedate, description=description, id=id),
                      ↪  follow_redirects=True)
7
8   def test_edit_entry(self):
9       self.create("user", "pw", "pw")
10      self.login("user", "pw")
11
12      rv = self.add_entry('title1', 'CS253.1', 'None', 'High',
         ↪  '1111-11-11T11:11', 'D1')
13      assert b"title1" in rv.data
14
15      # test case for when title is edited as empty
16      rv = self.edit_entry('title1-edit', 'CS253.1-edit', 'None-edit',
         ↪  'Low', '2222-22-22T22:22', 'D1-edit', 1)
17      assert b"title1-edit" in rv.data
18      assert b"CS253.1-edit" in rv.data
19      assert b"None-edit" in rv.data
20      assert b"Low" in rv.data
21      assert b"2222-22-22T22:22" in rv.data
22      assert b"D1-edit" in rv.data
```

The `edit_entry()` function is for editing the assignment entry, and the `test_edit_entry()` is for checking whether the data actually gets edited or not. First, a

post is added (while logged in) and then, edited. If the assignment data gets updated, the unit test passes.

5.6 Delete Assignment

The `Delete` button lets the user delete a specific assignment. Similar to the `Edit` button, `Delete` button has the assigment's `id` as its value.

```
1  @app.route('/delete', methods=['POST'])
2  def del_assignment():
3      if 'logged_in' in session:
4          db = get_db()
5          db.execute('delete from assignments where id=?',
           ↪  [request.form['id']])
6          db.commit()
7          flash('Assignment has been deleted')
8          return redirect(url_for('show_assignment'))
9      return render_template('Login.html')
```

The relevant HTML code that is tied with `del_assignment()` function from `ShowAssignments.html` is as follows:

```
1  <form action="{{ url_for('del_assignment') }}" method="post"
   ↪  class="inline">
2      <button type="submit" class="btn btn-primary glyphicon
       ↪  glyphicon-remove" title="Delete"></button>
3      <input type="hidden" value="{{ assignment.id }}" name="id">
4  </form>
```

Just as other functions mentioned previously, `Delete` button does not let the user delete if the user has logged out of his account. For instance, when the user logs out of his account from another tab, he cannot access the `Delete` feature anymore. It will just redirects back to the login page. That's why the `if 'logged_in' in session` condition is added to ensure that the delete feature is executed only when the user is logged in. After that, the code from line 5 deletes the assignment entry whose `id` value equals the `Delete` button's value and redirects the user back to `View Schedule` page.

For the unit test, it is implemented as follows:

```
def delete_entry(self, delete):
    self.create("user", "pw", "pw")
    self.login("user", "pw")
    return self.app.post('/delete', data=dict(id=delete),
    ↪  follow_redirects=True)

def test_delete_entry(self):
    self.create("user", "pw", "pw")
    self.login("user", "pw")

    rv = self.add_entry('title1', 'CS253', 'None', 'High',
    ↪  '1111-11-11T11:11', 'D1')
    assert b"title1" in rv.data

    rv = self.add_entry('title2', 'CS253', 'None', 'High',
    ↪  '1111-11-11T11:11', 'D2')
    assert b"title2" in rv.data

    rv = self.add_entry('title3', 'CS253.3', 'None',
    ↪  'High','1111-11-11T11:11', 'D3')
    assert b"title3" in rv.data

    # delete second post
    rv = self.delete_entry(2)
    assert b"title2" not in rv.data

    # then, delete the top post (with id=3)
    rv = self.delete_entry(3)
    assert b"title3" not in rv.data
    assert b"title1" in rv.data

    # then, delete the remaining post (with id=1)
    rv = self.delete_entry(1)
    assert b"title1" not in rv.data
```

The `delete_entry()` object function call accepts 1 argument `id` as its parameter and deletes the assignment with that `id` value (while logged in). For testing function `test_delete_entry()`, a few posts are added first, so that we can check whether the assignment with the correct id is being deleted in the presence of multiple assignments. For this particular test function, three assignments are added first. Then, they are deleted in the order of second (with `id` 2), third (with `id` 3), first (with `id` 1) assignments. Everytime the

assignment gets deleted, the function checks whether the assignment's data is still displayed in the `View Schedule` page, which shows all the assignments (the `not in rv.data` code checks this). If not, the unit test passes.

5.7 Full View

As mentioned in Section 2.3, `View Schedule` page only shows the truncated text inside `Description` column if it is very long. The `Description` field only displays up to 25 characters, which can be seen in line 85 of the `View Schedule` html implementation mentioned above. But when the user wants to view the full text in the `Description`, this is where the `Full View` functionality comes in. Below is the implementation of `full_view()` function:

```
1  @app.route('/full_view', methods=['GET'])
2  def full_view():
3      if 'logged_in' in session:
4          db = get_db()
5          cur = db.execute('select * from assignments where id = ?',
           ↪  [request.args['id']])
6          assignments = cur.fetchall()
7          return render_template('full_view.html', assignments=assignments)
8      return render_template('Login.html')
```

Again, `full_view()` feature is allowed only when the user is logged into their account; otherwise, it will redirect the user back to the login page. Similar to the other two buttons `Edit` and `Delete`, `Full View` button also has the assignment `id` value assigned as its value. What the function basically does is that upon clicking the `Full View` button, it selects the assignment entry with `id` value that matches the button's value and displays full description in `/full_view` page. The corresponding HTML code in `full_view.html` file is the same as that of `ShowAssignments.html`, but without `truncate(25)` in line 85.

The unit test for `full_view()` is also similar to that of `show_assignment()`, but now, we only test for the case whether the long description is displayed fully without truncation and the dots ("..."). The implementation for the unit test is shown below:

```
1  def test_full_view(self):
2      self.create("user", "pw", "pw")
3      self.login("user", "pw")
4
```

```python
    self.add_entry('title1', 'CS253.1', 'None', 'High', '1111-11-11T11:11', 'DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD')
    rv = self.app.get('/full_view?id=1')
    assert b"DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD" in rv.data
    assert b"..." not in rv.data
```

First, we add a post with a very long description (¿ 25 characters) while logged into the account. Then, we test whether the whole description appears in the `full_view` page. If it does, the unit test passes.

5.8 Show Calender

As described in Section 2.4, the `View Calendar` allows the user to generate the calendar of a certain month and year, along with the assignments that are due in that month and year. Its implementation is as follows:

```python
@app.route('/calendar')
def display_calendar():
    if 'logged_in' in session:
        return render_template('Calendar.html')
    return render_template('Login.html')

@app.route('/showcalendar', methods=['GET'])
def input_calendar():
    if 'logged_in' in session:
        db = get_db()

        month = request.args['month']
        year = request.args['year']

        if len(month) == 1:
            # for cases like when user enters "1" for January, instead of "01"
            month = "0" + month

        like_str = "{}-{}-%".format(year, month)

        cur = db.execute("select * from assignments where username = ? and duedate like ? order by duedate ASC",
                         [session['logged_in'], like_str])
```

```
23
24          assignments = cur.fetchall()
25
26          if month == "":
27              flash("Month cannot be empty.")
28
29          else:
30              if (int(month) < 1) or (int(month) > 12):
31                  flash("Month should be between 1 and 12 inclusively.")
32
33          if year == "":
34              flash("Year cannot be empty.")
35
36          if month != "" and year != "":
37              mo = int(request.args['month'])
38              yr = int(request.args['year'])
39              myCal = calendar.HTMLCalendar(calendar.SUNDAY)
40              newCal = myCal.formatmonth(yr, mo)
41
42              return render_template('Calendar.html', calendar=newCal,
                ↪  assignments=assignments)
43
44          return redirect(url_for('display_calendar'))
45
46      return render_template('Login.html')
```

The relevant exerpt of the code from `Calendar.html`, where it asks user input for month and year, is as follows (displaying the table part is the same as that of `ShowAssignments.html`) :

```
1   <form action="{{ url_for('input_calendar') }}" method="GET">
2       <dl>
3           <dt>Type in the numerical value of the month:
4         <dd><input type="text" size="30" name="month">
5           <dt>Type in the numerical value of the year:
6         <dd><input type="text" size="30" name="year">
7         <dd><input type="submit" value="Show Calendar">
8       </dl>
9   </form>
```

Both `display_calendar()` and `input_calendar()` redirects back to the login page if the user attempts to access their directories without being in a login session. They also

render the template `Calendar.html`. But the difference between these two functions is that `display_calendar()` deals with more "redirecting" back to the `View Calendar` page without passing in any databases. Redirecting is necessary because as we will see later in the code, if the user types in an invalid value (such as month 13), it will redirect the user back to the calendar page. In this case, no passing of database is necessary.

The `input_calendar()` does necessary conditional checking, such as if the user enters 1 instead of 01 for the month January, it automatically converts single digit number to 01 (because that's how Python `calendar` library reads for the month). It also checks that month should be in the range of 0 to 12 inclusively. In addition, neither month nor year field can be empty; otherwise, `display_calendar()` is called for redirecting. Upon successful user input for month and year, we use the following function from `calendar` library:

- `calendar.HTMLCalendar(calendar.SUNDAY)`: it brings up the calendar in HTML table form, with the starting day of the week as Sunday.
- `.formatmonth(yr, mo)`: it inserts the calendar for the particular `yr` year and `mo` month.

Python's documentation on calendar library provides more detail about it.

The unit test for `input_calendar()` involves checking whether a correct calendar is output for a given month and year values, along with the assignments that are due in that month and year. The implementation of the unit test is as follows:

```
1   def test_calendar(self):
2       self.create("user", "pw", "pw")
3       self.login("user", "pw")
4
5       self.add_entry('title1', 'CS253', 'None', 'High', '2018-01-30', 'D1')
6       self.add_entry('title2', 'CS253', 'None', 'High', '2018-02-20', 'D1')
7       self.add_entry('title3', 'CS253', 'None', 'High', '2019-04-20', 'D1')
8
9       rv = self.app.get('/showcalendar?month=1&year=2018')
10      assert b"January 2018" in rv.data
11      assert b"title1" in rv.data
12      assert b"title2" not in rv.data
13      assert b"title3" not in rv.data
14
15      rv = self.app.get('/showcalendar?month=2&year=2018')
16      assert b"February 2018" in rv.data
17      assert b"title1" not in rv.data
18      assert b"title2" in rv.data
```

```
19        assert b"title3" not in rv.data
20
21        rv = self.app.get('/showcalendar?month=3&year=2018')
22        assert b"title1" not in rv.data
23        assert b"title2" not in rv.data
24        assert b"title3" not in rv.data
```

First, we add three post with different due (month, year) pairs. Then, for a given (month, year) pair, we test whether the correct header of the calendar is displayed. For instance, (1, 2018) should display "Januray 2018" as its calendar header. We will not be worrying about whether days are correctly stated in the calendar; we will just trust the `calendar` library. Afterwards, we test whether *only* assignments that are due in (month, year) appear in the table below. If so, then the unit test passes.

5.9 Some Extra Redirect Functions

A lot of redirects happen in this application due to error-handling or certain functions sharing the same destination route (for instance, `add_assignment()` and `update_entry()` both lead to `ShowAssignments.html` page) and asks the user to put the correct data again. Thus, there are still some redirect functions that are not covered above. They are stated below:

- `redirect_add_assignment()`: this function checks whether the user is currently logged in or not. If so, it leads to `AddAssignment.html`, where the user can add an assignment. If not, it redirects the user back to the login page.
- `redirect_opening()`: if the user is already logged in, the root directory `/` just redirects him to the `/homepage` directory. We do this because the user can either login or create account in the root directory. For the user that is already signed in, he neither needs to login nor create an account.
- `redirect_login()`: if the user is already logged in, the `/login` directory redirects the user back to `/homepage`. This is simply because a user that is logged in already cannot logged in again (at least using the same browser window).
- `redirect_signup()`: similar reasoning as `redirect_login()`. The person who already has an account and logged in already do not need to create a new account. Thus, `/signup` redirects the logged-in user back to the `/homepage`

Their implementations are also as follows:

```
1  @app.route('/')
2  def redirect_opening():
```

```
    if 'logged_in' in session:
        return redirect(url_for('display_homepage'))
    return render_template('OpeningPage.html')

@app.route('/login')
def redirect_login():
    if 'logged_in' in session:
        return redirect(url_for('display_homepage'))
    return render_template('Login.html')


@app.route('/signup')
def redirect_signup():
    if 'logged_in' in session:
        return redirect(url_for('display_homepage'))
    return render_template('CreateAccount.html')
```

We do not need test cases for these functions because they are just simply redirecting the user from one directory to another.

6 Rooms for Improvements

Certainly, *AssignmentPlanner* is far from perfect, and there are so many room for improvements that the application could use. Those improvements could come from certain features that our team initially planned to implement, but did not include it due to the limited resources that we had at the moment.

One enhancement that could be made to *AssignmentPlanner* is to add the reminder system. But that would require the user to link into some sort of third party applications such as facebook or email. For emails, there is a Flask-Mail documentation that discusses how to send SMTP emails to the clients (users). Then, the server could remind the user about the upcoming assignment that is due near. This could be a huge improvement to the application since it can help users stayed organized even better.

Another opportunity of improvement for *AssignmentPlanner* would be to add a time zone and due time. Since assignments are time sensitive, it is crucial to have the application operate on the time zone that the user wishes to. Adding the time zone will also expand the our targetted audiences on a very large scale, since users from all over the world can use the application now. Also, time zone can even further improve how the reminder system, mentioned above, works. For instance, the user might set "remind me 2 hours before the assignment is due", but the due time could be calculated using the server's local time instead

of the user's local time. Thus, time zone would be a really valuable feature for *Assignment-Planner*. Unfortunately, our team do not have knowledge or expertise to implement a time zone for now.

7 Conclusion

To sum up, *AssignmentPlanner* application enables the user to create account and store their assignment into their accounts. It offers features such as `add`, `view`, `sort` `edit`, and `delete` assignments. It also allows the user to generate a calendar for a given (month, year) pair and shows a list of assignments that are due in that (month, year). We employ agile software development process and test the functionalities of these features using unit tests as we implement them so that we could make our website bug-free as much as possible.