Manual for "test_website"

Author: Kaleem Ullah (k.ullah@uva.nl)

Purpose

test_website is a Python flask web application. It tracks (1) the time spent by each visitor in the specified paths, and (2) whether a specified button ("Contact") is clicked or not by each visitor. It serves as an example web application for Computational Social Science students.

Dependencies

```
from flask import Flask, request, session, render_template, redirect
from datetime import datetime
from flask_sqlalchemy import SQLAlchemy
from flask_session import Session
```

The required packages are in the 'req.txt' file and can be installed using: `pip install -r req.txt`.

If the packages are not properly installed, you will see a 'ModuleNotFoundError'. For example: 'ModuleNotFoundError: No module named 'flask'

This is a summary of the functions we use in our application from each module:

flask:

i) Flask = Configures the application.

ii) session = Used to store session specific information.

iii) request = Contains the attribute `path` for every request that the user makes in Unicode.

iv) render_template = Used to render html templates stored in the 'templates' folder.

v) redirect = Used to redirect visitors to a specified route.

datetime:

i) datetime = Used to access the date and time after requests.

flask_sqlalchemy:

i) SQLAlchemy = Set up database to store information.

flask_session:

i) Session: Setting up flask sessions for each new visitor.

Configuring the application, session and databases

```
# Configure app
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db'

# Configure flask session
app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)

# Set up database and database models
db = SQLAlchemy(app)
```

The code used here is the standard for app configuration. The application is called "app" and it is a Flask object. The session is configured such that the files are stored in memory in the filesystem. The folder 'flask_session' is automatically generated once the application is executed and contains the data from user sessions. The database is initialized using SQLAlchemy.

Database model

```
# Database model for the continuous variable: time spent
class PageView(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    visitor_id = db.Column(db.String(10))
    page = db.Column(db.String(255))
    time_spent = db.Column(db.Integer)
    start_time = db.Column(db.DateTime)

# Database model for the binary variable: button click
class Button(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    visitor_id = db.Column(db.String(10))
    button = db.Column(db.Boolean)
```

After a database is set up using `db=SQLAlchemy(app)`, `db.Model` can be used to create tables in the database. The table name is the name of the class that is defined. Columns are added to the table using `db.Column`. In the arguments of the function, the type of data can be specified. The maximum length of the data that you expect can also be specified. For example: `visitor_id = db.Column(db.String(10))` creates a column called `visitor_id`, the column contains text of maximum length 10.

When a new table is added to the database, we need to use `db.create_all()` to add the tables to the database. This needs to be done in the application context, so we use:

```
# Create all the tables for the databases
with app.app_context():
  db.create_all()
```

Logging data

```
# Function to log data: this function saves the time spent on the previous page in
the database. Unit of time is seconds.
def log_data():
    try:
        time_spent = (datetime.now() - start_time).total_seconds()

        # First 3 seconds is the threshold to save the time spent in the database.
It is to eliminate recording repetitive page requests/reloads.
        if time_spent > 3:
            page_view = PageView(
                visitor_id=session.get('visitor_id'),
                page=previous_path,
                time_spent=time_spent,
                start_time=start_time)
            db.session.add(page_view)
            db.session.commit()
    except:
        pass
```

This is the function to log time spent data into the database. We calculate time spent by taking the difference of the timestamps taken now and when the user opens the page. The data has a lower bound of three seconds. This is an arbitrary threshold and can be changed given your data needs. The `PageView` object of the class db.model is created and added to the database.

Tracking time spent

```
# after_request decorator of Flask defines actions to be performed after each
request coming from the client-side.
@app.after_request
def track_time(response):
    global start_time
    global previous_path

    # Every time the user requests default route (/), time spent in the previous
path is recorded in the database with log_data().
    if request.path == '/':
        log_data()
        # Update start_time and previous_path
        start_time = datetime.now()
        previous_path = 'HomePage'

    # Every time the user requests /learn_more route, time spent in the previous
path is recorded in the database with log_data().
    if request.path == '/learn_more':
        log_data()
        # Update start_time and previous_path
        start_time = datetime.now()
        previous_path = 'Learn More'

    # Every time the user requests  /confirmation route, time spent in the 
previous path is recorded in the database with log_data().
    if request.path == '/confirmation':
        log_data()
        try:
            # Delete start_time and previous_path variables. Time spent on 
/confirmation route is not recorded.
            del start_time, previous_path
        except:
            pass
    return response
```

The function decorator `@app.after_request` is used with the `track_time()` function. This decorator runs the function after every request made by the user. A request here means requesting a webpage. This function basically runs at each webpage. The global variable `start_time` is used to keep track of time. The global variable `previous_path` is used to keep track of what the user visited before requesting the current page. `request.path` contains the information of the path of the website that the user requested.

Every time the user visits any page we log the data. If the user has not been on the site before, the `log_data` function will not throw any errors since we wrapped it in a try block.

The variables start_time and previous_path are removed at the "confirmation" page. If the user goes back to the "learn_more" page after the "confirmation" page, the program throws an error since the 'start_time' and 'previous_path' variables have been deleted. The try-except block is for error handling. This way the website will still run, even if the user does not follow your instructions. You can also make explicit exceptions depending on the error you expect.

Keep in mind that this code is written to collect individual level information and requires the user to visit the site in one session. If the user does not follow your instructions, you should see outliers in the dataset. You can use the data cleaning methods you learnt in the last semester to clean your data.

Creating and using routes

```
@app.route('/')
def index():
    # Getting the unique id from the home page URL. The unique URL will be 
generated by Qualtrics for each visitor.
    visitor_id = request.args.get('uid')
    # Add visitor_id to the session
    if visitor_id:
        session["visitor_id"] = visitor_id
    return render_template('index.html')


@app.route('/learn_more')
def learn_more():
    return render_template('learn_more.html')


@app.route('/confirmation')
def confirmation():
    return render_template('done.html')
```

All the routes are kept in the 'app.py' file in this example code. If you have many webpages, it is better to keep the routes in another file and import them in the 'app.py' using `from filename import *`.

To be able to track the time a user spends on the website, we first need to keep track of the user. Information in each new user session can be stored as a 'key, value' pair in the `session` object. The visitor_id is retrieved from the URL using `request.args.get()` function. In Qualtrics, you will assign this to be called "uid" and it will be added to the URL in this manner: `www.example.com\?uid=xxxxxxxxxx`. You will need to assign a randomly generated unique ID per visitor ("uid") in Qualtrics after which Qualtrics automatically adds it to the URLs that the users visit. Make sure the number of characters match the specifications of your database. In the database configuration above, the visitor_id is kept to 10 characters. Qualtrics will need to add 10 characters to the URL. This will be explained in detail in the Qualtrics workshop and you can consult the teaching assistants on how to do this.

If the user visits the website without the visitor_id assigned by Qualtrics, the `visitor_id` is None. Generated database will have an empty cell for visitor_id column. The session object remembers the `visitor_id` of the user. This way if the same user reloads the page, we know that it is not a unique visit.

Routing each webpage in the website can be done with the help of the `app.route()` decorator. The string inside this decorator can be considered as the name of the webpage. '/' refers to the index of the website. This is the homepage. If the website is 'www.example.com', the `app.route('/learn_more') refers to the webpage 'www.example.com/learn_more'.

The function runs when the specified route is requested by the user. There is a template made in HTML for each webpage which is stored in the 'templates' folder. For example, if the user opens the homepage, the `app.route('/')` is run, which allows the function `index()` to execute and the html file ('index.html') is returned using the function `render_template`.

Logging binary data

```

# /log_binary is the route that users are sent to when they click on the "Contact"
button.
# However, it is a dummy route which does not render a new template. It redirects
users to the Home Page.
# "Contact" button is added to provide an example structure for a button-click
data collection.
# button_tracking() function saves the visitor_id in the database if the visitor
clicked on the "Contact" button.
@app.route("/log_binary")
def button_tracking():
    try:
        button_click = Button(
            visitor_id=session.get('visitor_id'),
            button=True)
        db.session.add(button_click)
        db.session.commit()
    except:
        pass
    return redirect('/')
```

In addition to logging time spent, we also log binary data that keeps track of a button on the site. In our case, the button is named "Contact" which, when pressed by the user, runs the above function. The button click is added to another database table in our database as a binary variable. The user is then redirected to the homepage, so this button does not actually retrieve a contact page. We designed it this way to demonstrate an example prototyping choice enabling us to test the website without actually collecting sensitive data (e.g. email, name).

Concluding remarks

This code serves as a template which can be edited to meet your own specifications. Always test your code before deploying your website. This will help you know your limitations and improve your web application over time. You can add your own database tables with the required information in your use case. This might feel slightly overwhelming for some. If that is the case, you can always reach out for support.

Good Luck!