# Affordable Flight Ticket Planning

**Author** – Smit Patel

## Introduction

The project is called *Affordable Flight Ticket Planning* and was completed by one person. The app helps the user find the cheapest set of flights to take to reach from start to destination. The idea was inspired by countless websites online that come up with the cheapest air travel options while adding a lot of time and layovers in the process. While not always the best option, booking tickets this way is very cheap if one is fine with trading time and energy.

My vision for this project was to create a website that would ask the user to enter the start and destination cities and display the cheapest flight routes to get from start to end. Again, this is not always the best option but it was fun to create it and see how weird the routes get to make the costs cheaper.

Since I decided to create a Python web-app, the project includes a lot of 3rd party libraries. The main framework is Flask which is a Python library to create web applications with the latest standards and the Python programming language.

For the frontend, I used UI Kit to create the necessary form styling and indentations on the website and Google Maps API to draw the air routes on the map. To help with preprocessing, the Python wrapper for the Google maps API helped get coordinates for cities.

**How to run**

Running the code is simple. Open a terminal in the project folder and use the command ***pip install library-name*** to install the libraries. Links to the libraries and installation are given in the bibliography. After that, run the command “python app.py” to start the server. Then, visit the app at *localhost:5000* .

**Algorithm**

The algorithm used to calculate the cheapest route was Dijkstra’s shortest path algorithm. Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks (*wikipedia*.)

The algorithm works on a graph with weighted edges so I created a graph in Python using adjacency lists (the graph was sparse, so adjacency lists are better suited for this purpose.) The nodes were the cities and the weights were the cost of a direct flight from one city to another. The flight ticket costs were taken from Kayak.com and some costs were directly sourced from Google flight as Kayak.com does not cover all locations. After I had constructed the graph, I wrote Dijkstra’s algorithm to work on the undirected weighted graph.

**How it works**

When the server is started, Flask renders a file called *form_submit.html,* through the function form(). The html file is rendered with a list of cities that act as the starting and ending destinations.

The user is greeted with a form asking for starting and ending cities. Once the user enters the desired cities through the dropdown, they can click the Submit button to see the cheapest flight routes.

When the submit button is clicked, a POST method is initiated which calls the method *get_route()* in app.py. The values (name of cities) from the form are stored in the variables *start* and *end*.

If the user enters the same cities (i.e start == end), *error.html* is rendered which displays and error message. If not, the program calls the function shortestPath with the graph, and start and end as parameters. The function returns the shortest path as an array of cities based on the flight costs (edge weights) between two cities (nodes.)

The shortest path array is the passed on as an argument when Flask renders the map page. The map page has Javascript code which used Google Maps JS API to render the lines and markers on a map using the information given to it by Flask.

**Preprocessing the co-ordinates.**

To create a dictionary of city names as keys and their co-ordinates as values, I used the Google Maps API Python wrapper in the *getCityCoordinates.py* file. First, the city names were assembled in a list. Then the geocoding feature of the Google Maps API was used to convert the city names to their coordinates. A loop is set up to traverse through the list and geocode each element in the list. This is stored in a dictionary where the key is the city name and the value is the coordinate dictionary.

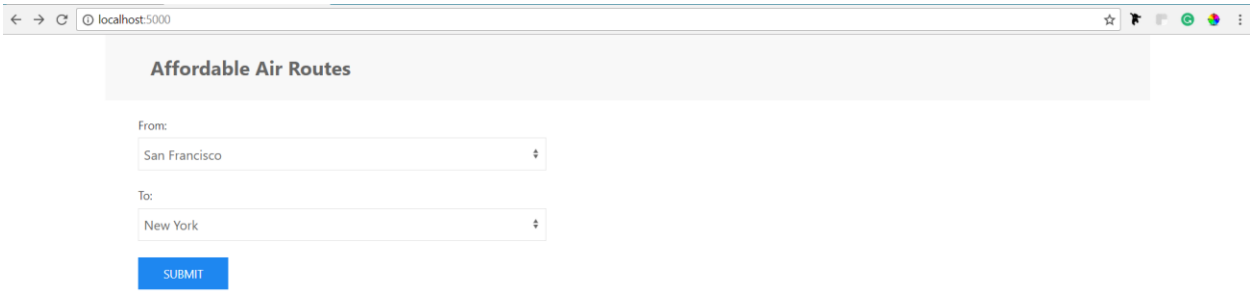
**Getting the Data**

As mentioned earlier, the flight tickets were taken from Kayak.com (and Google Flight were Kayak.com was not able to provide details.) I stored the results in an MS Excel spreadsheet and used the Python xlrd library to extract the data and store it in an array that was later used. The code for this is in the excel-conversion.py file.

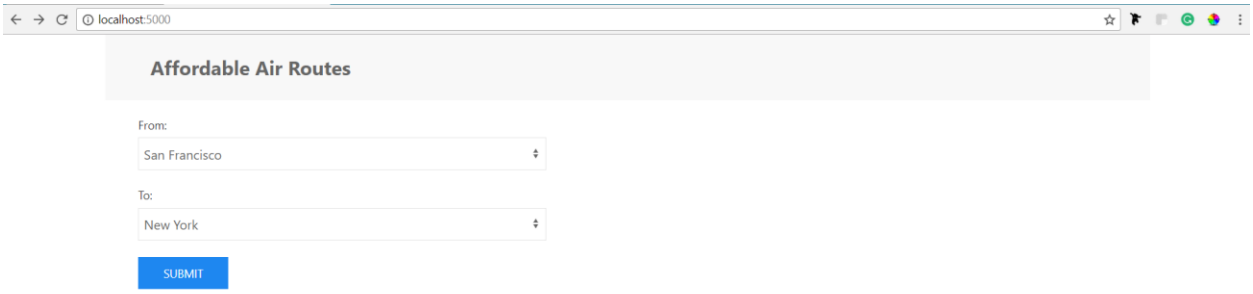
# Project Results

**Screenshots**

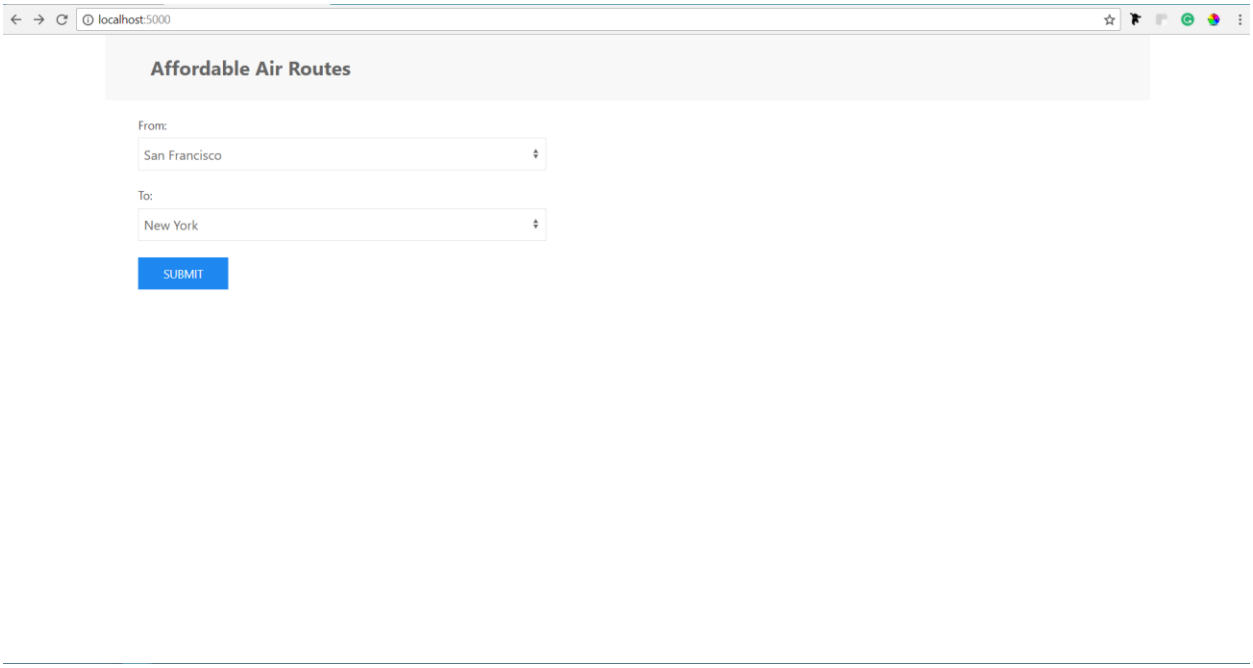
**1)** Input form




**2)** Result




**3)** Same cities input




**4)** Error message


**Please select different cities**

**Thoughts and Challenges**

After completing this project, I feel like I have a better understanding of Python and 3rd Party Python libraries in general. In class, I was introduced to the world of Python and learned specific features of the language. As such, the project seems like a natural extension of the class where I apply concepts learned in class like dictionaries and also learn new things about Python.

After completing this web app, I have realized that not only is it important to write and implement features of a program, but it is also important to have these features talking to each other properly. This is also important because in a somewhat complex program, each succeeding function needs data from the previous function. If one of the function fails to communicate properly, there is a negative domino effect that can cause problems.

For example, I had one part which displayed the routes on a map and one part which generated those routes. While these two features are integral to the project, they wouldn't serve their purpose unless they learned to communicate with each other.

The main challenge was also the main success. I am now much more confident in using 3rd party libraries for complex projects where different services are talking to each other.

Another big and complex challenge was to get the shortest path and the required co-ordinates and pass it on to the Google Maps JS API so that the API could create paths on the maps. It difficult as the data from the form page had to be passed on to the app.py page so that it could calculate the shortest path. Then the shortest path had to converted into a format (JSON) that the Maps JS API could understand. Once the conversion was done, it was difficult passing that information to the map display page and use it correctly.

While the project got complex at a few places and fairly easy at some parts thanks to Python, I learned a lot and look forward to creating more projects in the future.

**Future Improvements**

There are a few areas where I could have made the project better. The first one that immediately comes into mind is including the form and the map display page in one page. This would drastically improve the user experience and possibly speed.

Also, I feel that I could improve the data structures used in certain parts to make the program more efficient space wise. Since this project was a web app, it would be using a lot of shared resources and taking less space would help make the project better.

# Bibliography

**Libraries**


| | |
|---|---|
| UI Kit | https://getuikit.com |
| Xlrd | http://xlrd.readthedocs.io/en/latest/ |
| Google Maps Python Wrapper | https://github.com/googlemaps/google-maps-services-python |
| Flask | http://flask.pocoo.org |
| Google Maps JS API | https://developers.google.com/maps/documentation/javascript/ |


**Websites**


1) Kayak.com
2) google.com/flights
3) Wikipedia (for Dikjstra's algorithm.

Notes and homework assignments done in class.