# HW5

[HW4-HW5-HW6 Overview]

## Overview

In this homework you will write the code for the web server which will be at the heart of your web application.

Recall that your code will process data from 311 Service Requests data set available via OpenData Buffalo. For this homework I again suggest you work with the abbreviated version of this data set, containing a little over 1,000 data records: 311_Service_Requests_Abbreviated.csv

In this homework you will write code to process the data which will be needed (in HW6) to produce two visualizations:

### Donut chart

The donut chart will show the percentage of all service requests associated with each department (indicated in the "SUBJECT" field), within a given range of years.

### Scatter plot

The scatter plot will show the number of requests (Y axis) by duration (X axis), per department (data series), in a given year.

## Requirements

This section is long - read it all the same. It is long because it spells out in detail what each function needs to do, and how.

### Application code (add to App.py)

The application code must include the functionality written for HW4. It will be augmented with the following functions:

#### `App.departments` function

Define the function `departments(`*list_of_dictionaries*`)` so it returns a list of all the department names (which are in the "SUBJECT" field) in *list_of_dictionaries*, sorted by alphabetical order. One way to implement this function is to use a standard accumulation pattern and only add a value to the list if it is not already in the list; once the accumulation is complete the list can be sorted using the natural sort.

## `App.open_year` function

Define the function `open_year(`*dictionary*`)` so it returns the year in which a request was opened. In other words, it extracts the 4-character date part of the "OPEN DATE" field.

This can be done using a slice (see below), as follows. For example:

```
> a = "05/04/2016 12:21:00 PM"
> year = a[6:10]
> print(year)
2016
```

A slice is described in the strings section of the Python tutorial, which assumes `word` has been assigned the value `'Python'`:

> In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain substring:
>
> ```
> >>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
> 'Py'
>
> >>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
> 'tho'
> ```

## `App.filterYear` function

Model this function after the `App.filterRange` function. Define the function ~~`filterYear(`*list_of_dictionaries*`, `*key*`, `*low*`, `*high*`)`~~ `filterYear(`*list_of_dictionaries*`, `*low*`, `*high*`)` so it returns a list of dictionaries containing only those dictionaries *d* from *list_of_dictionaries* for which

~~*low <= open_year(d["OPEN DATE"]) < high*~~
*low <= open_year(d) < high*

## `App.date` function

Define the function `date(`*string*`)` so it returns a `datetime.date` object corresponding to the date given in the argument string; this will be the value in either the "OPEN DATE" or "CLOSED DATE" field in a dictionary.

For example:

`date("04/23/2016 11:37:00 AM")` must return the `datetime.date(2016,04,23)` object.

`date("06/14/2016 8:27:00 AM")` must return the `datetime.date(2016,06,14)` object.

This function will prove useful in conjunction with the `duration` function defined in HW4.

## `App.data_by_subject` function

***Clarification added on 11-12-2020:***

*For this function assume that the complete data set (as a list of dictionaries) is stored in a variable named `ALL_DATA`. For testing purposes you can read the contents of a CSV file using the `readCSV` function and assign the returned value to that variable.*

This function must return the data needed by Plotly to display a donut chart, as described above. The argument to this function will be a dictionary in this format:

```
{ "year_start": integer, "year_end": integer }
```

For example, here are some possible dictionaries:

```
{ "year_start": 2004, "year_end": 2012 }

{ "year_start": 2018, "year_end": 2018 }
```

For each year in the range (remembering that both endpoints are included) prepare a dictionary in the following general format:

```
{
        "values": [3, 17, 80],
        "labels": ['Dept A', 'Dept B', 'Dept C'],
        "domain": {"column": 0},
        "name": '2004',
        "hole": .4,
        "type": 'pie'
      }
```

The values are determined for each department by counting how many service requests there were for that department in the given year, and then dividing by the total number of service requests in the year. Express this percentage as an integer between 0 and 100.

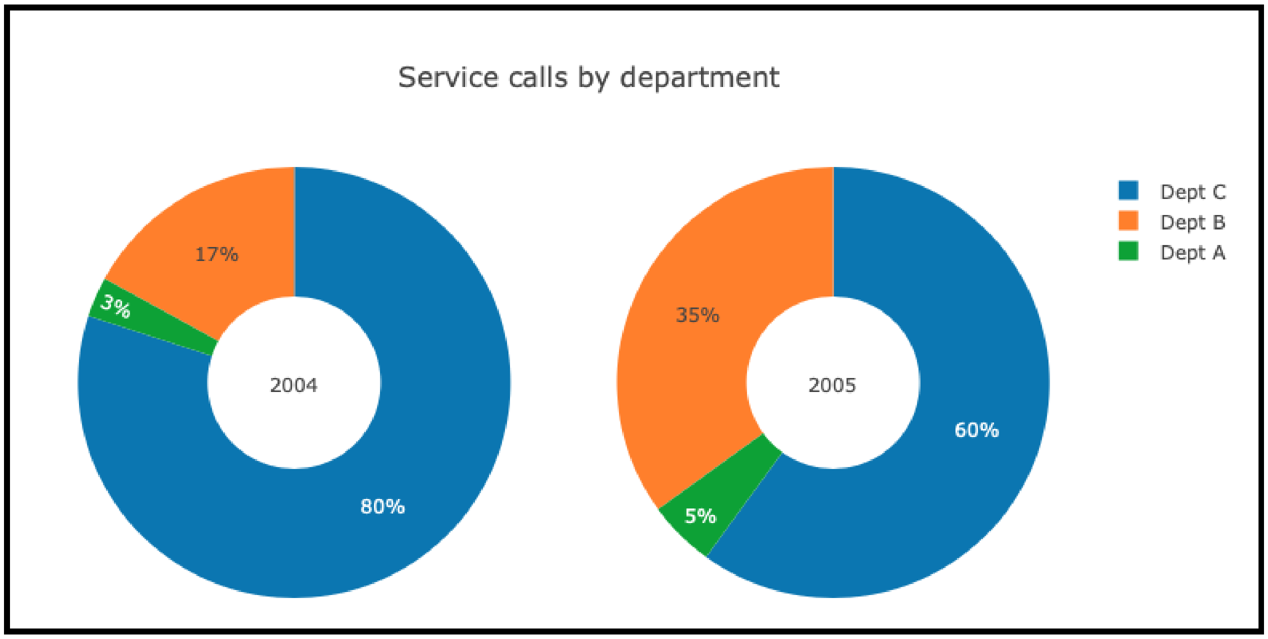
Use the functions defined in HW4, as well as the `filterYear` function, to collect together the relevant data items to determine these counts (e.g. `filterYear` will let you grab the service requests for a given year, and `keepOnly` will let you pull out the service requests for a specific department).

So that the same departments are represented in every dictionary, use the `App.departments` function to get a list of departments for the entire data set.

The value returned will be a list of these dictionaries, one per year in range, where the 'name' key is paired with the year. For example:

```
[
      {
        "values": [3, 17, 80],
        "labels": ['Dept A', 'Dept B', 'Dept C'],
        "domain": {"column": 0},
        "name": '2004',
        "hole": .4,
        "type": 'pie'
      }
,
      {
        "values": [5, 35, 60],
        "labels": ['Dept A', 'Dept B', 'Dept C'],
        "domain": {"column": 1},
        "name": '2005',
        "hole": .4,
        "type": 'pie'
      }
]
```

In HW6 data like this will be used to create donut chart similar to this:



## `App.data_by_subject_duration` function

***Clarification added on 11-12-2020:***

> *For this function assume that the complete data set (as a list of dictionaries) is stored in a variable named `ALL_DATA`. For testing purposes you can read the contents of a CSV file using the `readCSV` function and assign the returned value to that variable.*

This function must return the data needed by Plotly to display a scatter plot, as described above. The argument to this function will be a dictionary in this format:

```
{ "year": integer }
```

For example, here are some possible dictionaries:

```
{ "year": 2004 }

{ "year": 2018 }
```

For the indicated year prepare a dictionary for each department in the following general format:

```
{
```

```
          "x": [1,3,7],
          "y": [20,2,1],
          "name": 'Dept A',
          "mode": 'markers',
          "type": 'scatter',
        }
```

The "x" points denote service request durations (in days between the opening and closing of the request - call the `App.duration` function on appropriate `datetime.date` objects). The "y" points denote how many service requests there were of that duration for the given department.

Use the functions defined in HW4, as well as the `filterYear` function, to collect together the relevant data items to determine these counts (e.g. `filterYear` will let you grab the service requests for a given year, and `keepOnly` will let you pull out the service requests for a specific department).

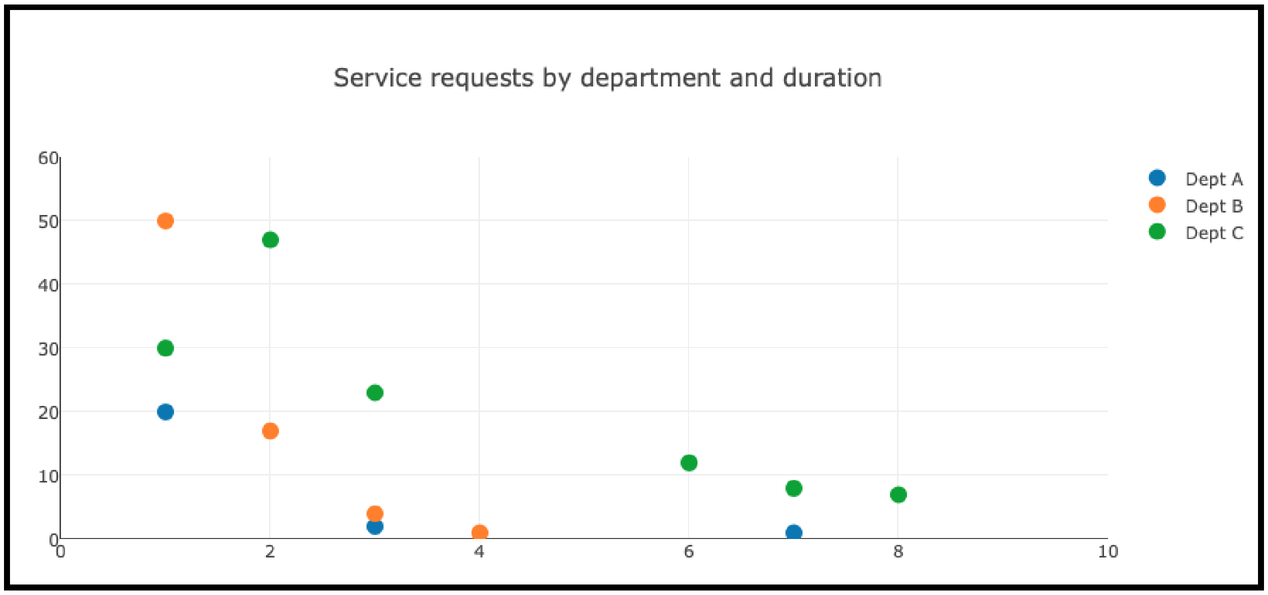
For a given dictionary `d`, the `date` function can be applied to `d["OPEN DATE"]` and `d["CLOSED DATE"]` to construct `datetime.date` objects suitable for use in a call to the `duration` function that you defined in HW4.

Use the `App.departments` function to get a list of departments for the entire data set.

The value returned will be a list of these dictionaries, one per department, where the 'name' key is paired with the department. For example:

```
[
        {
          x: [1,3,7],
          y: [20,2,1],
          mode: 'markers',
          type: 'scatter',
          name: 'Dept A',
          marker: { size: 12 }
        }
,
        {
          x: [1,2,3,4],
          y: [50,17,4,1],
          mode: 'markers',
          type: 'scatter',
          name: 'Dept B',
          marker: { size: 12 }
        }
,
      {
        x: [1,2,3,45,6,7,8],
        y: [30,47,23,16,12,8,7,3],
        mode: 'markers',
        type: 'scatter',
        name: 'Dept C',
        marker: { size: 12 }
      }
]
```

In HW6 data like this will be used to create scatter plot similar to this:



## Web server code (add to Server.py)

Define a Bottle web server so it has five routes as detailed below. Remember that each route is specified by an `@bottle.route` or an `@bottle.post` annotation, and that an annotation is associated with a function which handles requests along the indicated route. These functions *must* be named as indicated below if the autograder for HW5 is to award points for them. **Do NOT include the bottle.run function call in Server.py!**

(1) The "/" route must serve up a single HTML page as a static file. The function associated with this route must be named `serve_html`. The name of the HTML file must be `index.html`.

You may name the remaining *routes* as you wish - you should make them meaningful in the context of the application - but remember that the functions must be named as indicated.

(2) A  route to serve up the front end JavaScript code as a static file.  The function associated with this route must be named `serve_front_end_js`.  The name of the JavaScript file must be `front_end.js`.

(3) A route to serve up the AJAX JavaScript code as a static file.  The function associated with this route must be named `serve_AJAX`.  The name of the JavaScript file must be `ajax.js`.

(4) A POST route to serve up the data for a donut chart as a JSON string. The function associated with this route must be named `serve_donut`.  Remember that this is handling a POST request, and that a POST request includes a data payload.  The `serve_donut` function must:
   - recover the JSON blob that carries the data,
   - convert the JSON blog to Python data,
   - call the `App.data_by_subject` function with that data as argument, and
   - return the data that function provides, encoded as a JSON string.

(5) A POST route to serve up the data for a scatter plot as a JSON string.The function associated with this route must be named `serve_scatter`.  Remember that this is handling a POST request, and that a POST request includes a data payload.  The `serve_scatter` function must:
   - recover the JSON blob that carries the data,
   - convert the JSON blog to Python data,
   - call the `App.data_by_subject_duration` function with that data as argument, and
   - return the data that function provides, encoded as a JSON string.