

Lab #5

Due: 11/28/22 @ 11:59pm

Content Covered

HTTP Requests in Python, Bottle Server in Python, HTML, AJAX in JavaScript

Overview

Labs 4 and 5 will be a bit different than the previous 3. In labs 4 and 5 you will build a complete web application which can serve as a model for how to structure your project.

Each lab guides you through the process of writing a small web application. The web application produces a visualization in a browser front end, of data provided by a back end web server. Together they bring together all the elements of the course we've covered so far, including but not limited to reading and writing files, representing data using data structures, processing data (e.g. filtering, combining), using libraries, and setting up a simple web server.

Successful completion requires that a number of small components be assembled in a way that they can all work together. I do not expect you will be able to complete this without some guidance. **Do not be shy to take advantage of the Piazza forum and office hours to discuss issues you run into.**

Lab 4 will handle the application code (the code that actually does the work: reading/writing files, manipulating data, etc). Lab 5 will handle the web server backend, front end visualization, and integration of all the pieces into one coherent WebApp.

Getting Started

1. Sign into replit.com
2. **DO NOT CREATE A NEW REPL!** The code for this Lab should be written in the same REPL you used for Lab #4. Remember, the Lab #4 code should be in the `App.py` file.
3. In this lab we will process data from the [311 Service Requests](#) data set available via [OpenData Buffalo](#). This data set has over 800,000 data records. For ease of testing, you can work with an abbreviated version of this data set with a little over 1,000 data records available [here](#).
 - a. During your testing, it may be useful to create even smaller versions of the file to use. You can do so by editing the file in any spreadsheet application to get a feel for the data, and modify/shrink down the file.
4. You must also add the following file to your repl: [ajax.js](#)

Editing Your Code

In this lab you will be defining a number of functions across multiple files to perform various tasks in your web application. Each description below will specify which file to define your functions in, and any restrictions on naming, variables, outputs, etc.

Submission

This Lab is broken into two parts, A and B. Part A will be autograded, and Part B will be graded manually. The Autolab submission for Part A will open up no later than 11/14/22 @ 5:00pm.

Your final submission should include everything for both parts A and B. Autolab will grade Part A out of 50, and we will manually grade Part B out of 50.

Your final submission for this lab will be a .zip file downloaded from replit.com and submitted to [Autolab](#). You may submit as many times as you like, but only your last submission will count for your final grade.

Submissions close at 11:59pm on 11/28/22 and no late submissions will be accepted.

Basic 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, found [here](#).

In this homework you will write code to process the data which will be needed (in Part B) to produce visualizations with the plot.ly library in JavaScript:

1.1 [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.

Part A (autograded)

This section is long – but it is long because it spells out in detail what each function needs to do and how. Take your time, and ask questions if you get stuck.

Application Code (add to App.py)

App.departments

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

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 substrings:

```
>>> 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

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

$low \leq open_year(d) < high$

App.data_by_subject

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.

To avoid problems when submitting to AutoLab:

- 1) do NOT add parameters for `data_by_subject`
- 2) do NOT name the parameter of this function `ALL_DATA`
- 3) do NOT assign a value to `ALL_DATA` inside this function
- 4) you may assign a value to `ALL_DATA` outside of any function, so that you can call these functions for your own testing - the autograding code will overwrite any value you assign to the variable before running its tests.

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.

Compute the percentage long these lines:
*int(100*round(deptCount/totalCount, 2))*

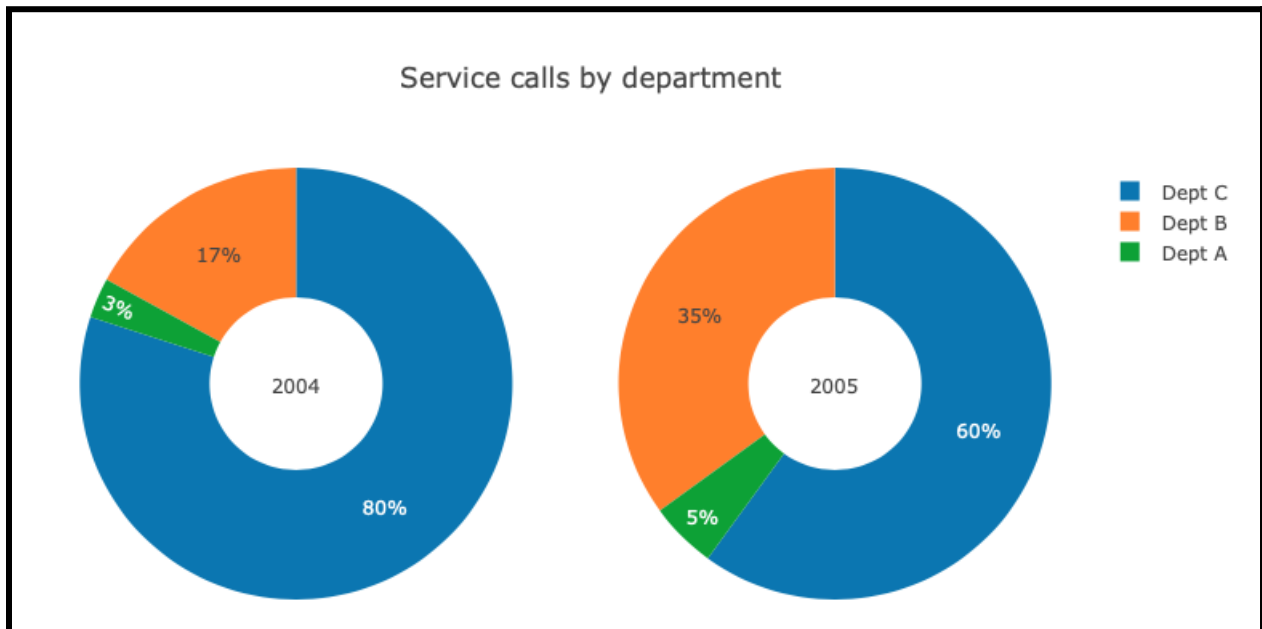
Use the functions defined in HW4, as well as the `filterYear` function, to collect together the relevant data items to determine these counts (e.g. `filterYear` will let you grab the service requests for a given year, and `keepOnly` will let you pull out the service requests for a specific department).

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. **Years which do not have any relevant data should be omitted from the list of dictionaries.** 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 part B data like this will be used to create donut chart similar to this:



Web Server Code (add to Server.py)

Define a Bottle web server so it has four 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 blob 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.

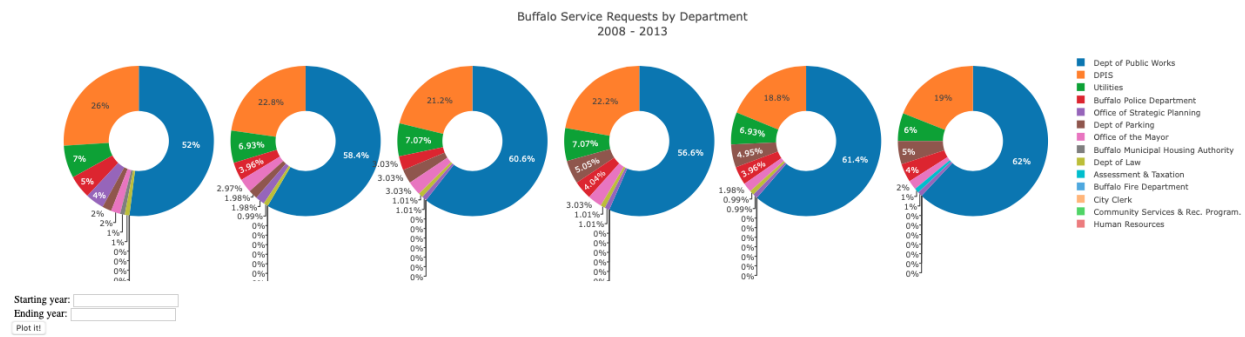
Part B (graded manually)

In this part you will write code to visualize the data which your code in part A produced.

If you did not complete the requirements for HW4 or HW5 part A, do that first. For part B you need to write two files (index.html and front_end.js), ensure that one file (ajax.js) is included in your repl (and in your submission), and make one small modification to your Server.py file.

In the end you will have a small web application that will be able to display a graph like this:

Donut



Define main.py

Your main.py file should now be defined with the following content:

```
import bottle
import Server

bottle.run(host="0.0.0.0", port=8080, debug=True)
```

If you still have testing code you wish to run to verify the HW4 and HW5 Part A functionality you can leave that in the file and comment out the

```
bottle.run(host="0.0.0.0", port=8080, debug=True)
```

line of code. When you are ready to try out your whole web application you should comment in the above line, and comment out any other testing code you may have in main.py.

Define index.html

The contents of `index.html` will define a user interface (UI) for your web application. A UI has two primary functions: to present information to a user, and accept input from a user.

The *head* element

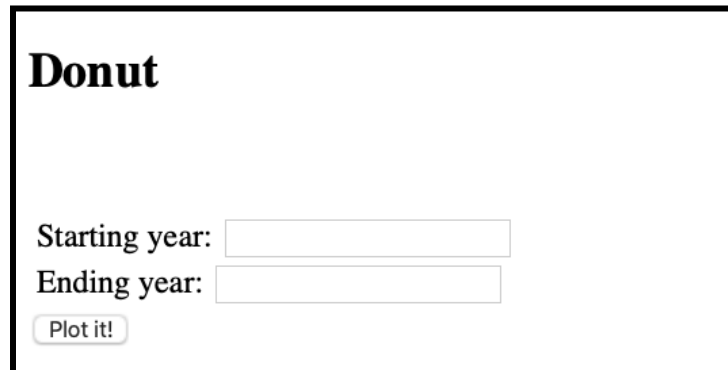
Of course our `index.html` file must also request various JavaScript code files. These files will be requested using `<script>` elements in the `<head>` of the HTML page. Javascript code must be requested from your web server along both the `ajax.js` and `front_end.js` routes.

The Plotly library must also be requested from `https://cdn.plot.ly/plotly-latest.min.js`

The *body* element

The two presentation elements will be a `<div>` element for the donut chart.

When the application starts up it should present a page that looks like this:



The screenshot shows a web form with a black border. At the top left, the word "Donut" is written in a large, bold, black serif font. Below it, there are two text input fields. The first is labeled "Starting year:" and the second is labeled "Ending year:". Below these two fields is a button with rounded corners and a light gray background, containing the text "Plot it!" in a small, black, sans-serif font.

The Donut chart

The user can enter a starting year and an ending year in the two input text fields. Clicking on the "Plot it!" button just after the two input text fields triggers an AJAX POST request to be sent to the back end server along the `'/donut'` route, along with a dictionary of the form

```
{"year_start":startYear , "year_end":endYear}
```

Where `startYear` is an integer corresponding to the year typed in the "Starting year:" text field, and `endYear` is an integer corresponding to the year typed in the "Ending year:" text field.

Note: For the purposes of this homework your code does not need to do any error checking on the values entered: if the user enters nonsensical values your application does not need to give an error message, though it must not crash either. After entering nonsensical values, entering valid values and clicking "Plot it!" should cause the application to behave correctly.

For example, entering the following for the donut plot,

Starting year:

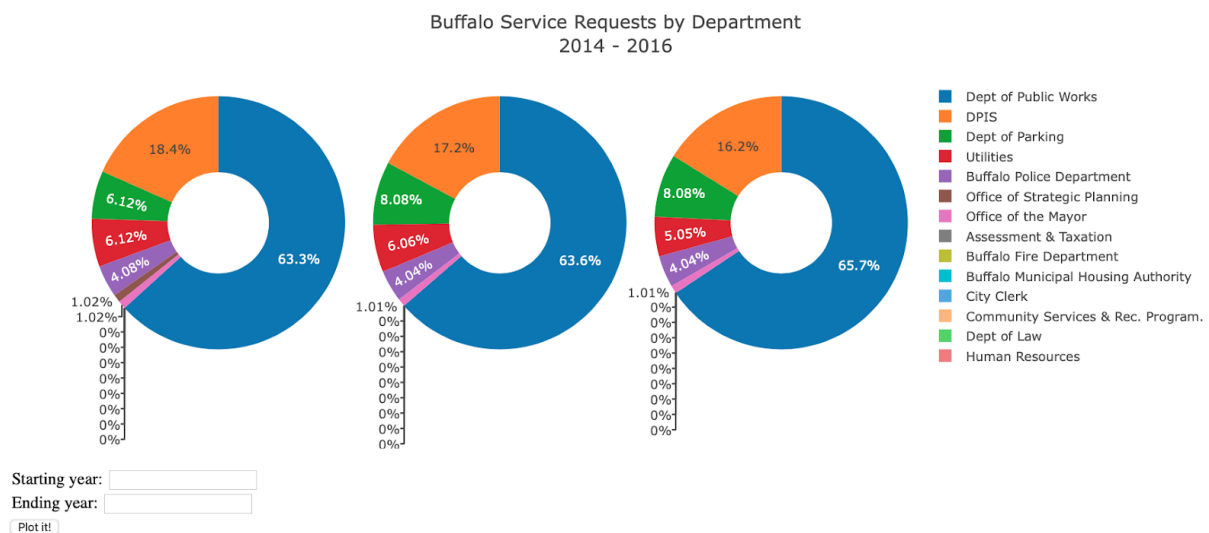
Ending year:

and clicking "Plot it!" should send this dictionary

```
{"year_start":2014 , "year_end":2016}
```

as part of the POST request to the server, which must respond with data to produce a graph that looks something like this:

Donut



Notice that the two text fields have been cleared.

The callback function for this POST request must call `Plotly.newPlot` with the data provided by the server, as well as some layout information. Figuring out a good layout with plotly can be tricky, so here's the layout we used above, assuming that `data` is the data that the server sent:

```
let s = data[0]["name"]
let f = data[data.length-1]["name"]
let layout = {
  title: 'Buffalo Service Requests by Department<br>'+s+" - "+f,
  showlegend: true,
  grid: {rows: 1, columns: data.length}
};
```

Define front_end.js

This file must define functions that provide the functionality necessary for the front end. Review the code of the Chat and Music Rater applications (posted on the website) as well as relevant lecture slides for information on using AJAX, callback functions, and using the Plotly library.

You will need to define two functions to create the donut chart.

You need one function that will be called when the "Plot it!" button is clicked. This function will grab the user inputs from the input element(s), clear the input element(s), and formulate an appropriate AJAX POST request to the web server, including specifying the callback function.

Note: the body of this function is short, 3 to 4 lines. Of course if you like to space out your code and use more variables than we did it may be a little longer...but the basic message is don't overcomplicate things!

You will also need a callback function. The callback function must unpackage the data received from the server, formulate an appropriate layout specification (as described above), and invoke Plotly to draw the graph.

Note: the body of this function is a little longer, maybe 10 to 15 lines, depending on how you write it. About five of those lines are for the layout (given to you above). And again, if you like to space out your code and use more variables than we did it may be a little longer...but the basic message is don't overcomplicate things!

Useful Suggestions and Advice

- **Think before you code** - think about the input to the function, and the desired output, and plan out how you would solve the problem in your head or on paper before you start to code. Having an idea of where you are going before you start typing can make the coding process smoother.
- **Write your own tests** - each problem describes an example test case or two that you can use to check if your implementation is correct. But they are not sufficient to guarantee you have a completely correct implementation. It can help to think through and write additional test cases (even before you code) to make sure your functions are behaving as you expect them to. This will also allow you to more quickly test your code as you go, rather than having to wait for Autolab to be open, and to go through the entire submission process just to get feedback.
- **Break down bigger problems into smaller ones** - if you are stuck on a problem, try solving a smaller piece of the problem first. You can even write it as a separate function, and test it out. Then once you have the small pieces working, building the solution to the bigger problem from your smaller solutions should be easier.
- **Ask for help** - during lab sessions, in office hours, and on Piazza. If you do choose to ask questions on Piazza **do not post your own code in public posts**. If you are going to make your question public, make sure it is general/conceptual in nature. Otherwise, make your question private to just the instructors.