Transform Pandas to JSON Flare for D3 Visualizations

Justin Chae
Dec 17, 2020 · 7 min read ★

How to transform a Pandas DataFrame to JSON with flare-like hierarchy to produce D3 Sunburst visualizations.

Photo by Jeremy Bishop on Unsplash

Data is king but colors are cool!

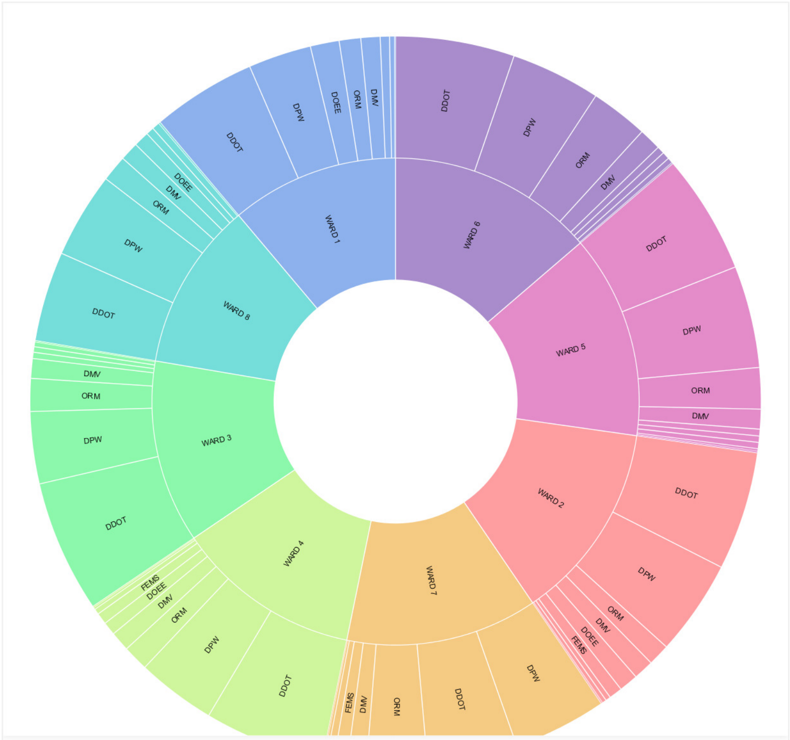
If your data and insights are worth telling, the right colors and design can make or break the crucial connection to your audiences. As a result, whether you are trying to win a competition or just trying to turn in a class assignment, chances are that colors can help make your case.

As I learned from personal experience, tricky data transformations on the back-end often put the desired visualization out of reach. For example, despite various libraries in Pandas, I was surprised to discover there is not a clear-cut way to transform a DataFrame to the exact JSON format required to make D3 work. When short on time, instead of mucking around with code, a seemingly easy option is to click, copy, and paste data but this type of manual work is error-prone and does not scale. Fortunately, I've figured out at least one of the problems and explain how to get over the hump in this article.

In this Article

I had a problem with data transformation to D3, got help from the Internet, solved the problem, and posted a solution to Stack Overflow. But, while the actual code to get the job done is important, the workflow and considerations to prepare the data are just as important — can't have one without the other.

As a result and to illustrate, this article contains a start to finish workflow on real data with code and explanations to transform a Pandas DataFrame to a JSON file that you can deploy immediately to a zoomable sunburst chart in D3.



@justinhc... 311 Service Requests in Washington... Observabl

The Data

As for the data (it is *always* about the data), this article leverages a Pandas DataFrame with data on average service times for Washington, D.C. 311 service calls. Link to GitHub with a derivative copy of the source data: https://github.com/justinhchae/p2d3. I originally solved this problem for a different project but decided to create this sample project to help others who may have similar data transformation issues form Python to D3.

Getting Started

There are some projects where it makes sense to describe hierarchy in data. For instance, if tracking sales data, we might start at the national-level, then work down in the hierarchy to the local city. When going back up the hierarchy, we can aggregate sales data from many cities into their respective states, many states into their regions, and then combine regional data to the national-level.

For D.C. 311 data, let's consider the top-level nodes to be each of the 8 Wards that comprise the city, the next level as each of the city agencies that service 311 requests, and then finally, the actual value of service request times. The data under analysis is the average length of time to complete each type of service request.

```
# necessary libraries for this project
import pandas as pd
import numpy as np
import os
import json
```

Get Data

The source data is from D.C.'s OpenData page, this project's data is based on a derivative located in an accompanying GitHub repo here. Terms of use: *Data retrieved from DC Data Catalog (http://data.dc.gov/).*

```
# set full path to csv containing data
filename =
'311_City_Service_Requests_in_2020.zip'
folder = 'data'
path = os.environ['PWD'] + os.sep +
os.sep.join([folder, filename])

# set columns to read (skip the rest)
usecols = ['WARD', 'ORGANIZATIONACRONYM',
'SERVICECODEDESCRIPTION',
'SERVICEORDERSTATUS', 'SERVICEORDERDATE',
'RESOLUTIONDATE']

# set WARD as a category type
dtype = {'WARD':'category'}

parse_dates = ['SERVICEORDERDATE',
'RESOLUTIONDATE']

# read in the csv params set above
df = pd.read_csv(path, usecols=usecols,
dtype=dtype, parse_dates=parse_dates)
```

Parse Date Data

How long does it take, on average, to close a 311 service call in any given Ward? To find the answer, let's take the difference of the service request date and the service resolution date.

```
# subtract times to produce service times
for each record
df['service_time'] = df['RESOLUTIONDATE'] -
df['SERVICEORDERDATE']

# divide times with numpy to produce times
as hours
df['service_time'] = df['service_time'] /
np.timedelta64(1, 'h')

# filter records where the service status
is closed
df['SERVICEORDERSTATUS'] =
df['SERVICEORDERSTATUS'].str.lower()
df = df[(df['SERVICEORDERSTATUS'] ==
'closed')]

# drop date columns that we no longer need
parse_dates.append('SERVICEORDERSTATUS')
df.drop(columns=parse_dates, axis=1,
inplace=True)

# return df with four columns (three
categories and one value)
df = df[['WARD', 'ORGANIZATIONACRONYM',
'SERVICECODEDESCRIPTION', 'service_time']]
```

Group and Clean DataFrame

It is worth noting that the preceding steps are only necessary to create some sample data for the next couple of sections. For instance, the data and analysis that lead up to this point will vary from project to project. However, I built this thing to parse out three levels of data. As a result, whatever you do, to make this conversion work, slice and dice your dataframe to have three columns of categories and one column of values. In this case, I chose to aggregate over the averages in service times by ward, service organization, and service description.

```
# drop missing values
df.dropna(thresh=2, inplace=True)

# append WARD to the front of every value
in the column
df['WARD'] = 'WARD ' +
df['WARD'].astype(str)

# group by and aggregate by average, drop
missing records
group = ['WARD', 'ORGANIZATIONACRONYM',
'SERVICECODEDESCRIPTION']
df = df.groupby(group)
[['service_time']].mean().dropna(axis=0,
how='any')

# reset index and return a dataframe ready
for transformation
df = df.reset_index()
```

Convert DataFrame to D3 JSON Flare

If you download my *Pandas to D3* (p2d3) library from GitHub, you can simply import and call the function.

```
from p2d3.pandas_to_d3 import PandastoD3

convert = PandastoD3()
convert.p2d3(df)

# returns json file ready for D3
```

However, to demonstrate how this works, consider the following code snip. While troubleshooting this problem, a friend made me realize that the key is to conditionally append nodes at each level. For example, only create a node and its children if the node has not been added. If a node is already added, instead of making a duplicate, append the nodes children. Then, do the same thing with the next child node.

```
# initialize a flare dictionary
flare = {"name": "flare", "children": []}

# iterate through dataframe values
for row in df.values:
    level0 = row[0]
    level1 = row[1]
    level2 = row[2]
    value = row[3]

    # create a dictionary with all the row
data
    d = {'name': level0,
          'children': [{'name': level1,
                        'children':
[{'name': level2,

'value': value}]}]}
    # initialize key lists
    key0 = []
    key1 = []

    # iterate through first level node
names
    for i in flare['children']:
        key0.append(i['name'])

        # iterate through next level node
names
        key1 = []
        for _, v in i.items():
            if isinstance(v, list):
                for x in v:
                    key1.append(x['name'])

    # add the full row of data if the root
is not in key0
    if level0 not in key0:
        d = {'name': level0,
              'children': [{'name': level1,
                            'children':
[{'name': level2,

'value': value}]}]}
        flare['children'].append(d)
    elif level1 not in key1:

        # if the root exists, then append
only the next level

        d = {'name': level1,
             'children': [{'name': level2,
                           'value':
value}]}

        flare['children']
[key0.index(level0)]['children'].append(d)

    else:

        # if the root exists, then only
append the next level

        d = {'name': level2,
             'value': value}

        flare['children']
[key0.index(level0)]['children']
[key1.index(level1)]['children'].append(d)
```

Important to note here that this is not a perfect technical solution. It only works for a specific case of three levels and the nested for loops and conditionals slow down the runtime. However, it works exactly as I need and I probably won't need to run it on a massive dataset. An improved solution should leverage faster built-in functions, some recursion, and probably a lot less code!

Return Data

```
# save to some file

with open('flare.json', 'w') as outfile:
json.dump(flare, outfile)

print(json.dumps(flare, indent=2))
```

A sample of the output.

```
# sample output of the transformed data
Writing Flare to JSON
{
  "name": "flare",
  "children": [
    {
      "name": "WARD 1",
      "children": [
        {
          "name": "DCRA",
          "children": [
            {
              "name": "DCRA - Illegal
Construction",
              "value": 210.40070746527792
            },
            {
              "name": "DCRA - Vacant
Private Property Inspection",
              "value": 635.8518215710985
            }
          ]
        },
        {
          "name": "DDOT",
          "children": [
            {...
```

ObservableHQ

Once you have a JSON file in the format you need, ObservableHQ makes it crazy easy to make a super slick visualization.

- Go here: https://observablehq.com/@d3/zoomable-sunburst
- Fork the notebook
- Click on the […] and then click on file attachments
- Replace the existing file with your new JSON file
- Bask in the glory of your new sunburst chart

Conclusion

We often work in tables, more specifically, Pandas DataFrames to analyze data. However, to display our results and connect to others, transformation to new formats such as JSON are often required. To leverage a great platform like ObservableHQ to produce interesting

D3 visualizations, we have to go a little further than the given libraries that come with Pandas.

The p2d3 library that I created and explained in this article is a derivative work of two other sources including my own Stack Overflow post. However, my code adds some object-oriented abstraction and provides for one additional layer of hierarchy. Better yet, it has worked for me on three distinct datasets and works well enough to shortcut some error-prone manual labor. If I given more time, a good next step will be to make the code more

Data Science   Python   D3js   Pandas   Visualization