A Python Programmers' Guide to Dashboarding — Part 1

Introduction, Basic Components & Interactive Graphs

Drimik Roy Follow
Jan 7, 2019 · 7 min read

Written by: Drimik Roy & Namrata Chaudhary — January 7, 2019

This series uses **Dash by Plotly** as the underlying framework for Dashboarding.

Introduction

Dash by Plotly is a framework, build on top of Plotly.js, React, and Flask, that can be used to build web applications and modern UI elements like charts, tables, and interactive input and output functionalities where the user would engage with this platform through Python.

There are several sample dashboards along with documentation including various web components supported by Dash in its user guide (https://dash.plot.ly). However, because of its novelty, there aren't as many examples available to model and explore particular web elements such as multiple tabs, links to multiple pages, and other dashboard tools that facilitate an easy-to-use enterprise-wide dashboard.

In this blog series, we elaborate on how certain web elements available on Dash can be used to build a highly interactive and detailed dashboard with dynamic parameters and tabular sections with the goal of enabling the user to have the tools necessary to construct a multi-layered, organizational framework deployed as a web application. Moreover, we point out possible errors we encountered along with their workaround solutions that can be useful in writing an efficient and robust code to create the dashboard. Towards the end of the series we weigh Dash against some of the more established tools for dashboarding such as R Shiny and Bokeh.

Note: In all sample codes illustrated in the series, html.Div elements are contained within the app layout html element (explained in 'Components of Dash') and other functions (e.g. callbacks) are defined separately.

Components of Dash

Each web application powered by Dash is created using a code that can be split into three basic sections: the Layout, Interactivity functions, and the Main function, each of which is described below:

Layout

The layout section of the Dash code contains HTML & CSS components supported by Dash libraries and decides what the application looks like in the final output web page. Dash has the libraries `dash_core_components` and the `dash_html_components` which provide web elements that can be used in the layout. There is also the `graph_objs` library by plotly that gives the user access to additional graphical components. Moreover, as indicated previously, one can always build their own components using React.js and Javascript. The layout section is written as:

```
app = dash.Dash(__name__,
       external_stylesheets=<external_stylesheets>)

app.layout = html.Div(children=[ < Add your Web components here > ])
```

Below are the basic modules required for the implementation for all the graphical elements of the series:

```
import dash
import dash_core_components
import dash_html_components
import dash.dependencies
```

modules-dash-series.py hosted with ♡ by GitHub view raw

Main Modules for Graphical Elements

Below is a basic example of a simple line graph specified in the layout section of the Dash code:

```
html.Div(children=[
```

```
html.Div(children=[
        html.Div([
        dash_core_components.Graph(
                id='example-graph',
                figure={
                'data': [ {'x': data['date'],'y': data['Sales'],'type': 'line'} ],
                'layout': go.Layout(
                                xaxis={"title":"Time"}, yaxis={"title":"Sales"}) } )
                ], style={"display":"inline-block", "width":"35%"})
        ])
```

layout_example_1.py hosted with ♡ by GitHub view raw

Where `data` is a Pandas DataFrame containing the columns *`date`* and *`Sales`* used as the *x* and *y* axes, respectively.

Interactivity

Dash supports web applications with interactive components where users can modify input data of elements like graphs and tables. The backend code supporting such usability falls under the interactivity section and a basic structure for this consists of two sub-components.

i) `@app.callback()` function — where the *`input id`* of the dash element causing the change and the *`output id`* of the dash element altered as a result of the change are specified.

ii) The body of the function that needs to run every time an input is changed, needs to be defined.

Let us use the above example and modify the x-axis date range of the Sales graph using an input `Date Range picker` component:

```
dash_core_components.DatePickerRange( id='date-picker-range',
        start_date = datetime(2018, 5, 3), end_date=datetime(2018, 10, 29) )
```

layout_date_picker_range.py hosted with ♡ by GitHub view raw

Above is the layout of a date range selector elements supported by the `dash_core_components` library. Below we illustrate how inputs and outputs are specified in app.callback() function where function is defined to modify the x-axis (i.e. date axis) based on input date range.

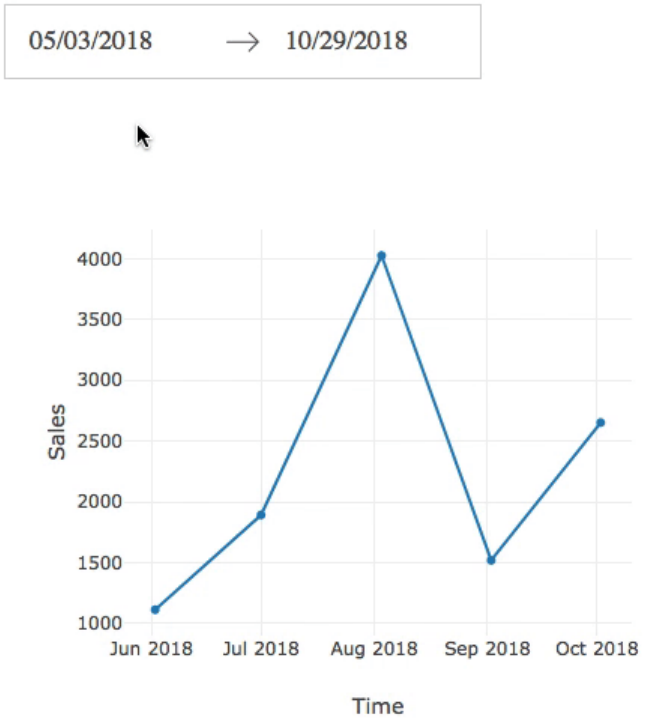
```
@app.callback(
    dash.dependencies.Output(component_id='example-graph', component_property='figure'),
```

```
    [dash.dependencies.Input(component_id='date-picker-range', component_property='start_date'),
    dash.dependencies.Input(component_id='date-picker-range', component_property='end_date')]
)

def update_output_div(start_date,end_date):
    return {
    'data': [go.Scatter(
                x=data[(data['date']>=start_date)&(data['date']<=end_date)]['date'],
                y=data[(data['date']>=start_date)&(data['date']<=end_date)]['Sales'])]
            ,
    "layout": go.Layout(
                xaxis={"title":"Time"}, yaxis={"title":"Sales"}
             )
        }
```

layout_output_div_callback_1.py hosted with ♡ by GitHub view raw

Output:



Tip: Handling Errors for Table and Graph Components

A often overlooked aspect of the Dash elements such as graphs and tables is that they are developed to display any previously held information *if an error has occurred*. A common issue is that the graphs continue to exhibit the information from the most previous callback if the current callback cannot execute as planned (perhaps due to an invalid input parameter).

This is misleading as the user on the page has no knowledge of this fact and thus, is lead to erroneous interpretations of data shown. A simple solution is to add error conditions in the callback function in the following ways:

- If an error occurs on a *graph*, then return `{'data': [], 'layout':[]}` which is an empty graph
- If an error occurs on a *table*, then return `{}` which is an empty table

As a result, the user is made aware about the presence of an issue by clearing any content previously held in these visual elements.

Main function

Like any other python code, the backend code that supports a web application powered by Dash also contains a main function, which can be used to call the server that launches the resulting application on a local url and port. If no port is specified, the default is reverted to `127.0.0.1:8050`. Below is an example:

```
# port 8051 chosen arbitrarily
if __name__ == '__main__':
        app.run_server(debug=True,port=8051)
```

layout_main.py hosted with ♡ by GitHub view raw

Advanced Dashboarding Ideas using Dash

1) Graphs and interactive tables

1.1) Color formatting the chart area

A graphical representation of a metric over time may require indications on the graph itself representing some division of the metric over specific time ranges (or any other metric on the x-axis). This indication is visually possible by having the said graph display different colors in the chart area for the specific time ranges on the x-axis. Dash allows for such a functionality by modifying the `layout` portion of a plotted graph. This is implemented using the `graph_objs` library by plotly.

The idea proposed is to add rectangular shapes of different colors in the chart area that have high transparency such that the data displayed and the categorical segment attributed by the color to each datum is easily visible. Below is an example implementation of a chart displaying sales trend over time, and we use the above mentioned concept to indicate multiple seasons, with the start and end dates of a season stored in a variable.

Variables of the x-axis, y-axis and indicate season ranges on the x-axis are defined:

```
colors = ['rgba(240,230,140,0.6)', 'rgba(175,238,238,0.6)', 'rgba(255,192,203,0.6)']
Xaxis  = ['2018-05-03','2018-06-02','2018-08-31','2018-10-30','2018-11-29','2018-12-29']
Yaxis = [3485, 1114,  1893,  4027,  1520,  2653]
Seasons = [('2018-05-03','2018-08-25'),('2018-08-25','2018-11-29'),('2018-11-29','2018-12-29')]
```

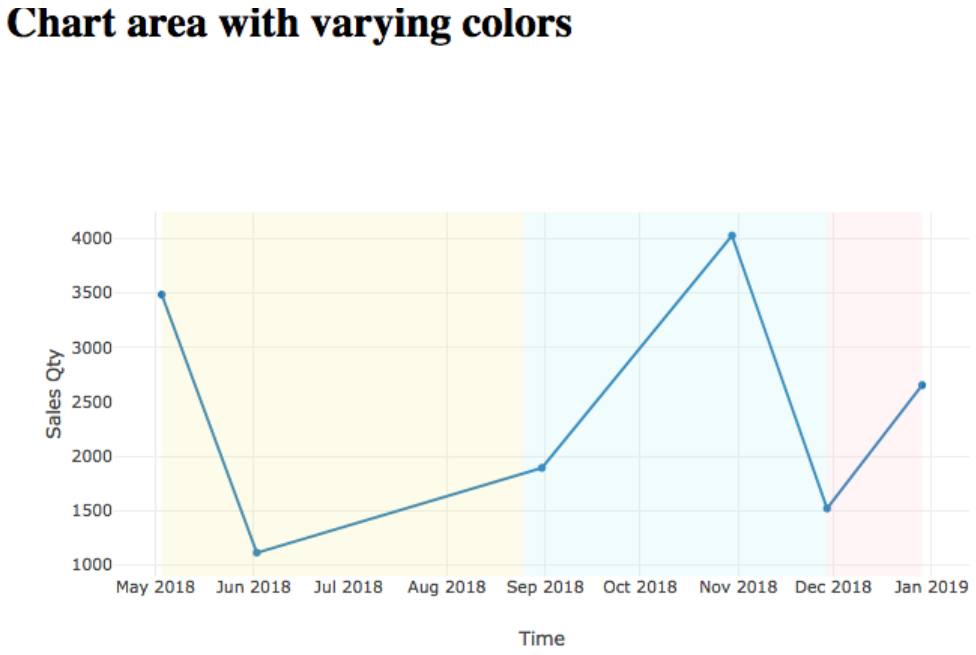
advanced_var_declarations.py hosted with ♡ by GitHub view raw

This is how the layout section of a graph can be modified to display the changing chart area color:

```
html.Div(children=[
        html.H1(children=' Chart area with varying colors'),
        dash_core_components.Graph(
        id='example-graph',
        figure={
                'data': [ {'x': Xaxis,'y': Yaxis,'type': 'line','name': 'Sales'} ],
            'layout': go.Layout(
                xaxis={"title":"Time"}, yaxis={"title":"Sales Qty"}, width=300,
                shapes=[
                    {
                        'type':'rect', 'xref':'x','yref':'paper',
                        'x0':exp[0],'y0':0,'x1':exp[1],'y1':1,
                        'fillcolor': colors[index%3],
                        'opacity': 0.30,
                        'line': {
                                'width': 0,
                        }
                    } for index,exp in enumerate(Seasons)]
           )
        }
    )
])
```

advanced_color_changing_impl.py hosted with ♡ by GitHub view raw

Consecutively, the output of this code indicates the three time ranges in separate colors in the chart area as shown below -



1.2) Optionally Visible Web Elements

Dash allows the option to display elements of the application based on a toggle switch, granting the user the ability to hide and show web divisions based on his or her desire.

How this works is through the use of two elements

1. **`html.Details([ … ])`** : what would be placed inside are the features of your dashboard that the user would like to be hidden from the layout and will be available for display based on the toggle, where the location of this element will be based on the user's app.layout.
2. **`html.Summary([`*`String`*`])`** : By providing a summary element after the `Details` element, an informative message is printed above the toggle on the application (the message being the *`String`*). This adds to the readability of your application and helps identify a switch (if there are many on the application).

As an example let us consider that in our previous example, we wanted to keep the graph with absolute sales optionally visible in relation to the rest of the app layout.

Sample use in a code:

```
1  html.Details([
2          html.Summary('Display sales quantity', style={"font-size":"22"}),
```

```
        html.Div([
        dash_core_components.Graph(
                id='example-graph1',
                figure={
                'data': [ {'x': Xaxis,'y': Yaxis,'type': 'line'} ],
                'layout': go.Layout(
                        xaxis={"title":"Time"}, yaxis={"title":"Sales Qty"}) } )
                ], style={"display":"inline-block", "width":"33%", "font-size":"12"})
    ]),

html.Div([
        dash_core_components.Graph(
        id='example-graph2',
        figure={
                'data': [ {'x': Xaxis,'y': Yaxis_revenue,'type': 'bar'} ],
                'layout': go.Layout(
                xaxis={"title":"Time"}, yaxis={"title":"Sales Revenue"}, width=150) } )
        ], style={"display":"inline-block", "width":"33%", "font-size":"12"})
```

advanced_details_summary_example.py hosted with ♡ by GitHub view raw

Output:

Through this tool, Dash once again emphasizes on user functionality and friendliness by providing in-built packages to improve organizational

factors of your application. One downside is that the hidden containers are not generated based on the click, but in fact are always continuously being modified if their input parameters are changing. The `Details` aspect just enables whether or not the data should be displayed.

1.3) Downloading content of a table into file

Dash incorporates a functionality to download any data from the backend code as a flat file onto your device. Through this, the user has the ability to not just view graphs and tables, but download the underlying data comprising them for further analysis. The download functionality is independent of the data shown on the web application and any sort of data that can be made available in the backend python code can be downloaded.

This functionality can be implemented by deploying a trigger on the dashboard that causes the file to get downloaded, and assigning that trigger element with a specific hyperlink. For the purpose of demonstration, we will be using a button element. Additionally, the backend code should have a function which gets called when the server is routed to that hyperlink (when the button is clicked), and executes the *`send_file`* function with the data to be downloaded passed as a parameter. This is a predefined function in the Flask library that sends the contents of a file to the client, which in our case is the person accessing the Dash-supported web application.

Example Code:

```
# app layout element
html.Div([
        html.A(html.Button('Download Graph Data'), href='/dash/data_download')
        ],style={"textAlign":"left","font-size":"20"})


# callback function
# note: 'directory' indicates the location of the raw_graph_data.csv file
@app.server.route("/dash/data_download")
def download_data():
        return flask.send_file(directory + 'raw_graph_data.csv',
                        mimetype='text/csv',
                        attachment_filename='raw_graph_data.csv',
                        as_attachment=True
        )
```

advanced_download.py hosted with ♡ by GitHub view raw

Output:

This concludes Part 1 of our series! To visit the subsequent parts, click below:

Part 2: Controls and Callbacks & Organizational Properties (Multi-Page App with Tabs and Links)

Part 3: Evaluation of Dash & Comparison to other Web Application Frameworks

Thanks to Namrata Chaudhary.

Python3 Web App Development Data Analysis Dash Plotly