

Dashboards in Python for Beginners and Everyone Else using Dash



Eric Kleppen

Follow

Jan 6 · 13 min read ★



I love sharing and visualizing data as much as analyzing it, so I was enthusiastic about a project at work that gave me an excuse to finally learn Dash. Dash is a framework for Python written on top of Flask, Plotly.js, and React.js, and it abstracts away the complexities of each of those technologies into easy to apply components. Anyone who has a little Python and HTML experience will feel like Dash empowers them to create customized and interactive web-based dashboards without breaking a sweat. To quote the documentation's Introduction,

“Dash is simple enough that you can bind a user interface around your Python code in an afternoon.”

If you want to build something cool without being bogged down by the nuts and bolts under the hood, this is the guide for you. If you're new to Dash, read on friend!

If you're already familiar with Dash and want to see the final layout, scroll to the bottom or find the code on my Github.

bendgame/DashApp

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

github.com



After wrapping my head around the core concepts, creating a simple dashboard felt like a “plug and play” experience. Dash can utilize Bootstrap CSS too, which makes styling and page-layout even easier to piece together.

Although I like Dash so far, it uses a lot of dictionaries and lists making it is easy to lose your place. Once you get the hang of the patterns, it doesn't feel bad. It can feel tricky to keep track of where your lists and dictionaries end at the beginning, so my advice is to use a lot of comments to keep track of your place.

I wont cover hosting the application in this article, but I will show you how to create an app and organize into multiple files, making it easier to manage and enhance. Dash and Plotly are vast and powerful tools, and this is only a taste of what they can do. If you really want to impress your friends, dig into the documentation and push yourself to design something amazing. Now onto some code!

Dash User Guide

Dash User Guide & Documentation

dash.plot.ly

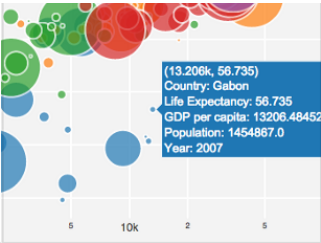
Plotly Python Graphing Library



Plotly's Python graphing library makes interactive, publication-quality graphs.

Examples of how to make line plots

plot.ly



Necessary Installations

Run these pip install commands to make sure you have the correct libraries:

```
pip install pandas
pip install dash
pip install dash-bootstrap-components
pip install plotly
```

The Wine Data Dashboard

I am using the wine review dataset from Kaggle. I have already cleaned the data and saved it to a sqlite3 database. I cover the data transformations in a previous article. This is the basic dashboard I'll show how to create. We will create an app that has two tabs and two filters:

Wine Dash

Filters

☐ Only rating >= 95

Price Slider

\$500

\$1000

\$1500

\$2000

\$2500

\$3000

Tab one

country	description	rating	price
Portugal	This is ripe and fruity, a wine that is smooth while still structured. Firm tannins are filled out with juicy red berry fruits and freshened with acidity. It's already drinkable, although it will certainly be better from 2016.	87	15
Portugal	This wine is light in tannins and ripe in fruit, with a delicious red-berry character. Drink this attractive wine from 2019.	85	11
Portugal	Towards the western end of the Douro vineyards, Avidagos is a fine place for table wines. This young, ripe and rounded wine is made to be drunk young. It has great black fruits and acidity that are linked by the firm tannin background. The wine will be ready from the end of 2016.	87	15
Portugal	This Avidagos vineyard in the Corgo River Valley is one of the oldest in the Douro. The wine, from a parcel of old vines, is hugely powerful and dense. With its dark tannins, concentrated black fruits, it could be too much. Somehow it	93	65

Tab two

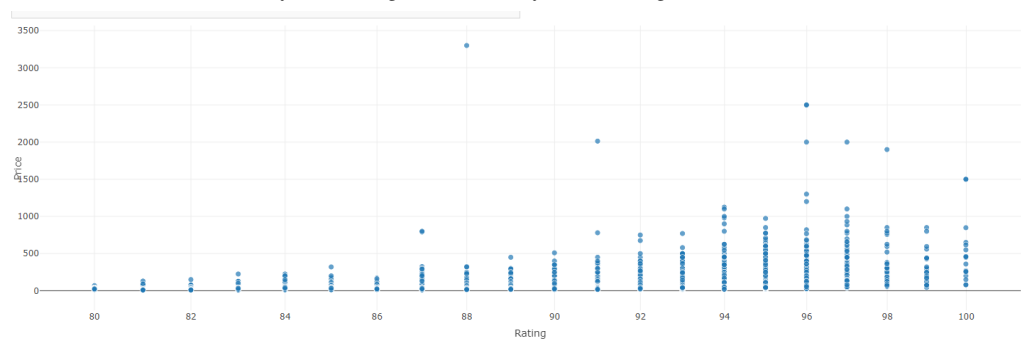
province	title	variety	winery	color
Douro	Quinta dos Avidagos 2011 Avidagos Red (Douro)	Portuguese Red	Quinta dos Avidagos	red
Douro	Quinta dos Avidagos 2015 Lote 138 Red (Douro)	Portuguese Red	Quinta dos Avidagos	red
Douro	Quinta dos Avidagos 2014 Avidagos Red (Douro)	Portuguese Red	Quinta dos Avidagos	red
Douro	Quinta dos Avidagos 2011 Lenuma Red (Douro)	Portuguese Red	Quinta dos Avidagos	red

Wine Dash

Filters

Data Table

Scatter Plot



The first tab is the raw data in a filterable and sort-able data table.

Tab two is a responsive scatter plot of the data points by price and rating.

The data can be filtered using the filters in the side panel.



Dash Concepts to Know

Before we get into dash-boarding, let me explain a couple things...If you're already familiar with Dash and just want my layout template, skip this part... Dash apps are primarily composed of two parts:

Layout

Callbacks

Layout

The layout is made up of a tree of components that describe what the application looks like and how users experience the content. If you're skilled with JavaScript and React.js, you can build your own components; however, Dash ships with multiple component libraries like `dash_core_components`, `dash_html_components`, and Dash DataTable. The `dash_html_components` library has a component for nearly every HTML tag. The `dash_core_components` library includes higher-level interactive components like buttons, input fields, and dropdowns. Dash DataTable makes it easy to integrate interactive data tables into your application.

Callbacks

Callbacks are what hold the logic for making Dash apps interactive. This will be covered in much more detail when I show you the code, but for now, just understand that Callbacks are Python functions that are automatically called whenever an *input* component's property changes. It is possible to chain callbacks, making one change trigger multiple updates throughout the app.

Callbacks are made up of Inputs and Outputs. The functionality works through the `app.callback` decorator. Inputs and Outputs are simply the properties of a component that a user can interact with. For example, an *Input* could be the option you select from a droplist and an *Output* could be a visualization.

A Simple Example

I'll show you how easy it is to make a simple dashboard before we get into the complex example. Name the file `index.py`.

Import Dependencies and Data

I'm reading from a sqlite database, but you can read a csv file using pandas if you're not familiar with sqlite.

```
import dash
import dash_bootstrap_components as dbc
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output
import dash_table

import pandas as pd
import sqlite3

import plotly.graph_objs as go

conn =
sqlite3.connect(r"C:\Users\MTGro\Desktop\coding\wineApp\db\wine_data
.sqlite")
c = conn.cursor()

df = pd.read_sql("select * from wine_data", conn)
df = df[['country', 'description', 'rating',
'price', 'province', 'title', 'variety', 'winery', 'color']]
df.head(1)
```

	country	description	rating	price	province	title	variety	winery	color
0	Portugal	This is ripe and fruity, a wine that is smooth...	87	15.0	Douro	Quinta dos Avidagos 2011 Avidagos Red (Douro)	Portuguese Red	Quinta dos Avidagos	red

First row of the dataset

Configure the Dash App

To configure the app to run, you'll need to include this bit of boilerplate code. Notice this is where the external style sheet is set. Since I am using Bootstrap CSS, I set it like using `[dbc.themes.BOOTSTRAP]`

```
app = dash.Dash(__name__, external_stylesheets =
[dbc.themes.BOOTSTRAP])
server = app.server
app.config.suppress_callback_exceptions = True
```

Add the Layout

Remember, the first pillar of the Dash app is the tree of components that form the *Layout*. Since the app has two tabs, I'll start by rendering the tabs in the layout. An HTML `<div>` Tag defines a division or a section in an HTML document. [Learn more about html here](#).

```
#set the app.layout
app.layout = html.Div([
    dcc.Tabs(id="tabs", value='tab-1', children=[
        dcc.Tab(label='Tab one', value='tab-1'),
        dcc.Tab(label='Tab two', value='tab-2'),
    ]),
    html.Div(id='tabs-content')
])
```

Take note of the *id* set for **dcc.Tabs** and the *values* set for **dcc.Tab**. Those are the *Input* for the callback I'll write to make the tabs interactive and return a layout. As the Output for the callback, I will use the *id* value "tabs-content" from **html.Div(id='tabs-content')**. If I were to run the app without the callback, it would look like this:

Tab one

Tab two

Two tabs, no content

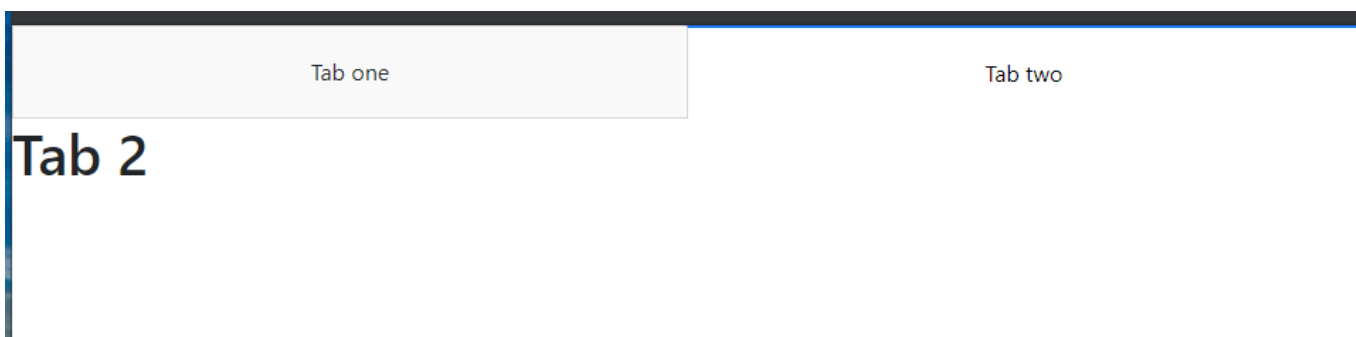
Add Callbacks

Let there be interaction! The values I specified as Inputs and Output are properties of particular components. This is where Dash provides the magic... to quote their documentation:

Whenever an input property changes, the function that the callback decorator wraps will get called automatically. Dash provides the function with the new value of the input property as an input argument and Dash updates the property of the output component with whatever was returned by the function.

```
#callback to control the tab content
@app.callback(Output('tabs-content', 'children'),
              [Input('tabs', 'value')])
def render_content(tab):
    if tab == 'tab-1':
        return html.H1('Tab 1')
    elif tab == 'tab-2':
        return html.H1('Tab 2')
```

This callback uses the *value* from the **dcc.Tab** to determine which html value to return as the child for *id* “tabs-content”



Tab 2

Add Data to the App

Now that the layout functionality is in place, add data to the tabs. Tab one contains table data, which displays in the app using the Dash component library [DataTable](#). Since the dataset contains over 100k rows, I decided to use backend paging and filtering using Pandas. This looks like a lot of code, but keep in mind, most of it come directly from the Dash documentation. Dive into the documentation to learn more about styling and functionality!

The DataTable Layout

```
@app.callback(Output('tabs-content', 'children'),
               [Input('tabs', 'value')])
def render_content(tab):
    if tab == 'tab-1':
        return html.Div(dash_table.DataTable(
            id='table-sorting-filtering',
            columns=[
                {'name': i, 'id': i, 'deletable':
True} for i in df.columns
            ],
            style_table={'overflowX': 'scroll'},
            style_cell={
                'height': '90',
                # all three widths are needed
                'minWidth': '140px', 'width':
'140px', 'maxWidth': '140px',
                'whiteSpace': 'normal'
            },
            page_current= 0,
            page_size= 50,
            page_action='custom',

            filter_action='custom',
            filter_query='',

            sort_action='custom',
            sort_mode='multi',
            sort_by=[]
        ))
```


The Scatter Plot Layout

The scatter plot is produced by using the [Dash Core Component Graph](#). It is also possible to use Plotly with the component. I'll show how to do that in the more advanced example at the end.

```
elif tab == 'tab-2':
    return html.Div([
        dcc.Graph(
            id='rating-price',
            figure={
                'data': [
                    dict(
                        y = df['price'],
                        x = df['rating'],
                        mode = 'markers',
                        opacity = 0.7,
                        marker = {
                            'size': 8,
                            'line': {'width': 0.5, 'color': 'white'}
                        },
                        name = 'Price v Rating'
                    )
                ],
                'layout': dict(
                    xaxis = {'type': 'log', 'title': 'Rating'},
                    yaxis = {'title': 'Price'},
                    margin = {'l': 40, 'b': 40, 't': 10, 'r': 10},
                    legend = {'x': 0, 'y': 1},
                    hovermode = 'closest'
                )
            )
        )
    ])

```

The DataTable Callbacks

This is boilerplate code taken from the documentation for the Dash DataTable Component library. Additional inputs can be added to callback to apply side panel controlled filtering.

```
operators = [['ge ', '>='],
             ['le ', '<='],
             ['lt ', '<'],
             ['gt ', '>'],
             ['ne ', '!='],

```

```

        ['eq ', '='],
        ['contains '],
        ['datestartswith ']]

def split_filter_part(filter_part):
    for operator_type in operators:
        for operator in operator_type:
            if operator in filter_part:
                name_part, value_part = filter_part.split(operator,
1)
                name = name_part[name_part.find('{') + 1:
name_part.rfind('}')]

                value_part = value_part.strip()
                v0 = value_part[0]
                if (v0 == value_part[-1] and v0 in ('"', "'", '`')):
                    value = value_part[1: -1].replace('\\"' + v0, v0)
                else:
                    try:
                        value = float(value_part)
                    except ValueError:
                        value = value_part

# word operators need spaces after them in the filter string,
# but we don't want these later
                return name, operator_type[0].strip(), value

    return [None] * 3

@app.callback(Output('table-sorting-filtering', 'data'),
[Input('table-sorting-filtering', "page_current"),
Input('table-sorting-filtering', "page_size"),
Input('table-sorting-filtering', 'sort_by'),
Input('table-sorting-filtering', 'filter_query')])

def update_table(page_current, page_size, sort_by, filter):
    filtering_expressions = filter.split(' && ')
    dff = df
    for filter_part in filtering_expressions:
        col_name, operator, filter_value =
split_filter_part(filter_part)

    if operator in ('eq', 'ne', 'lt', 'le', 'gt', 'ge'):
        # these operators match pandas series operator method names
        dff = dff.loc[getattr(dff[col_name], operator)
(filter_value)]
        elif operator == 'contains':
            dff = dff.loc[dff[col_name].str.contains(filter_value)]
        elif operator == 'datestartswith':
            # this is a simplification of the front-end filtering logic,
            # only works with complete fields in standard format
            dff =
dff.loc[dff[col_name].str.startswith(filter_value)]

```

```

if len(sort_by):
    dff = dff.sort_values(
        [col['column_id'] for col in sort_by],
        ascending=[
            col['direction'] == 'asc'
            for col in sort_by
        ],
        inplace=False
    )

page = page_current
size = page_size
return dff.iloc[page * size: (page + 1) *
size].to_dict('records')

```

Run the App

```

if __name__ == "__main__":
    app.run_server()

```

Once the code is complete, run the app through the console using *python index.py*

```

(base) C:\Users\MTGro\Desktop\coding\dash\wineDash>python index.py
Running on http://127.0.0.1:8050/
Debugger PIN: 998-712-481
* Serving Flask app "app" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
Running on http://127.0.0.1:8050/
Debugger PIN: 292-371-739

```

python index.py

Tab one					Tab two		
country	description	rating	price	province	title	vari	
filter data...							
Portugal	This is ripe and fruity, a wine that is smooth while still structured. Firm tannins are filled out with juicy red berry fruits and freshened with acidity. It's already drinkable, although it will certainly be better from 2016.	87	15	Douro	Quinta dos Avidagos 2011 Avidagos Red (Douro)	Portuguese	
Portugal	This wine is light in tannins and ripe in fruit, with a delicious red-berry character. Drink this attractive wine from 2019.	85	11	Douro	Quinta dos Avidagos 2015 Lote 138 Red (Douro)	Portuguese	
	Towards the western end of the Douro vineyards.						

Portugal	Avidagos is a fine place for table wines. This young, ripe and rounded wine is made to be drunk young. It has great black fruits and acidity that are linked by the firm tannin background. The wine will be ready from the end of 2016.	87	15	Douro	Quinta dos Avidagos 2014 Avidagos Red (Douro)	Portuguese
----------	--	----	----	-------	--	------------

Simple Tab Dash App

. . .

Adding the Sidepanel and File Structure

The app is functional, but as you can tell, expanding the number of tabs and callbacks will make the code feel long and hard to manage. Instead, break the content down into separate files.

The File Structure

I structured the files in a way that made it easy to add additional tabs and functionality without bloating a single file or impacting the layout of the other tabs. Need to add a new tab? EASY! Just add a new Tab file and update the index.py file to look for the new tab.

```

Top_Folder
|_ app.py
|_ index.py
|_ Layouts
    |_ __init__.py
    |_ SidePanel.py
    |_ Tab1.py
    |_ Tab2.py
    |_ Tab3.py

|_ Database
    |_ db.sqlite
    |_ transformations.py

```

App.py

Since I am adding callbacks to elements that don't exist in the app.layout as they are spread throughout files, I set **suppress_callback_exceptions = True**

```
import dash
import dash_bootstrap_components as dbc
app = dash.Dash(__name__, external_stylesheets =
[dbc.themes.BOOTSTRAP])

server = app.server

app.config.suppress_callback_exceptions = True
```

Index.py

This is the file that will be run in the console. The `app.layout` returns `sidepanel.layout`. It also contains majority of the callbacks used in the app. The functionality for the sidepanel is written into the callbacks.

```
import dash
import plotly
import dash_core_components as dcc
import dash_html_components as html
import dash_table
from dash.dependencies import Input, Output
import dash_bootstrap_components as dbc

import sqlite3
import pandas as pd

from app import app
from tabs import sidepanel, tab1, tab2
from database import transforms

app.layout = sidepanel.layout

@app.callback(Output('tabs-content', 'children'),
              [Input('tabs', 'value')])
def render_content(tab):
    if tab == 'tab-1':
        return tab1.layout
    elif tab == 'tab-2':
        return tab2.layout

operators = [['ge ', '>='],
             ['le ', '<='],
             ['lt ', '<'],
             ['gt ', '>'],
             ['ne ', '!='],
             ['eq ', '='],
```

```

        ['contains '],
        ['datestartswith ']]

def split_filter_part(filter_part):
    for operator_type in operators:
        for operator in operator_type:
            if operator in filter_part:
                name_part, value_part = filter_part.split(operator,
1)
                name = name_part[name_part.find('{') + 1:
name_part.rfind('}')]

    value_part = value_part.strip()
    v0 = value_part[0]
    if (v0 == value_part[-1] and v0 in ('"', "'", '`')):
        value = value_part[1: -1].replace('\\"' + v0, v0)
    else:
        try:
            value = float(value_part)
        except ValueError:
            value = value_part

# word operators need spaces after them in the filter string,
# but we don't want these later
return name, operator_type[0].strip(), value

return [None] * 3

@app.callback(
    Output('table-sorting-filtering', 'data')
    , [Input('table-sorting-filtering', "page_current")
    , Input('table-sorting-filtering', "page_size")
    , Input('table-sorting-filtering', 'sort_by')
    , Input('table-sorting-filtering', 'filter_query')
    , Input('rating-95', 'value')
    , Input('price-slider', 'value')
    ])
def update_table(page_current, page_size, sort_by, filter,
ratingcheck, prices):
    filtering_expressions = filter.split(' && ')
    dff = transforms.dff
    print(ratingcheck)

    low = prices[0]
    high = prices[1]

    dff = dff.loc[(dff['price'] >= low) & (dff['price'] <= high)]

    if ratingcheck == ['Y']:
        dff = dff.loc[dff['rating'] >= 95]
    else:
        dff

```

```

for filter_part in filtering_expressions:
    col_name, operator, filter_value =
split_filter_part(filter_part)

if operator in ('eq', 'ne', 'lt', 'le', 'gt', 'ge'):
    # these operators match pandas series operator method
names
    dff = dff.loc[getattr(dff[col_name], operator)
(filter_value)]
    elif operator == 'contains':
        dff = dff.loc[dff[col_name].str.contains(filter_value)]
    elif operator == 'datestartswith':
        # this is a simplification of the front-end filtering
logic,
        # only works with complete fields in standard format
        dff =
dff.loc[dff[col_name].str.startswith(filter_value)]

if len(sort_by):
    dff = dff.sort_values(
        [col['column_id'] for col in sort_by],
        ascending=[
            col['direction'] == 'asc'
            for col in sort_by
        ],
        inplace=False
    )

page = page_current
size = page_size
return dff.iloc[page * size: (page + 1) *
size].to_dict('records')

```

DataBase > transforms.py

The transforms file contains any transformations the data needs to go through before being called into the app.

```

import dash
import dash_bootstrap_components as dbc
import pandas as pd
import sqlite3
from dash.dependencies import Input, Output

conn =
sqlite3.connect(r"C:\Users\MTGro\Desktop\coding\wineApp\db\wine_data
.sqlite")
c = conn.cursor()

```

```
df = pd.read_sql("select * from wine_data", conn)

df = df[['country', 'description', 'rating', 'price', 'province',
'title', 'variety', 'winery', 'color', 'varietyID']]
```

Tabs > sidepanel.py

This is where the power of Bootstrap CSS comes in. The layout grid makes it easy to control the look of the layout. There are three main layout components in *dash-bootstrap-components*: `Container`, `Row`, and `Col`.

```
import dash
import plotly
import dash_core_components as dcc
import dash_html_components as html
import dash_bootstrap_components as dbc
import dash_table
import pandas
from dash.dependencies import Input, Output

from app import app

from tabs import tab1, tab2
from database import transforms

df = transforms.df
min_p=df.price.min()
max_p=df.price.max()

layout = html.Div([
    html.H1('Wine Dash')
    ,dbc.Row([dbc.Col(
        html.Div([
            html.H2('Filters')
            , dcc.Checklist(id='rating-95'
            , options = [
                {'label':'Only rating >= 95 ', 'value':'Y'}
            ])
            ,html.Div([html.H5('Price Slider')
                ,dcc.RangeSlider(id='price-slider'
                    ,min = min_p
                    ,max= max_p
                    , marks = {0: '$0',
                        500: '$500',
                        1000: '$1000',
                        1500: '$1500',
                        2000: '$2000',
                        2500: '$2500',
```



```

                                3000: '$3000',
                                }
                                , value = [0,3300]
                                )

                                ])

                                ], style={'marginBottom': 50, 'marginTop': 25,
'marginLeft':15, 'marginRight':15})
                                , width=3)

,dbc.Col(html.Div([
                                dcc.Tabs(id="tabs", value='tab-1', children=[
                                dcc.Tab(label='Data Table', value='tab-1'),
                                dcc.Tab(label='Scatter Plot', value='tab-2'),
                                ])
                                , html.Div(id='tabs-content')
                                ]), width=9)])

])

```

Tabs > Tab1

This is the DataTable from the example at the beginning of the article. The callbacks are in the index. The callbacks include functionality for filtering using the sidepanel components.

```

import dash
import plotly
import dash_core_components as dcc
import dash_html_components as html
import dash_bootstrap_components as dbc
import dash_table
import pandas as pd
from dash.dependencies import Input, Output

from app import app
from database import transforms

df = transforms.df

PAGE_SIZE = 50

layout =html.Div(dash_table.DataTable(
                                id='table-sorting-filtering',
                                columns=[
                                {'name': i, 'id': i, 'deletable':
True} for i in

```

```

df[['country','description','rating','price','province','title','variety','winery','color']]
],
style_table={'height':'750px',
             'overflowX': 'scroll'},

style_data_conditional=[
    {
        'if': {'row_index': 'odd'},
        'backgroundColor': 'rgb(248,
248, 248)'
    }
],
style_cell={
    'height': '90',
    # all three widths are needed
    'minWidth': '140px', 'width':
'140px', 'maxWidth': '140px', 'textAlign': 'left'
    , 'whiteSpace': 'normal'
}
, style_cell_conditional=[
    {'if': {'column_id': 'description'},
    'width': '48%'},
    {'if': {'column_id': 'title'},
    'width': '18%'},
]
, page_current= 0,
page_size= PAGE_SIZE,
page_action='custom',

filter_action='custom',
filter_query='',

sort_action='custom',
sort_mode='multi',
sort_by=[]
)
)

```

Tabs > Tab2

Instead of using the Dash Core component Graph, I am using Plotly's Scattergl to get better performance visualizing the dataset.

```

import dash
import dash_core_components as dcc
import dash_html_components as html
import dash_bootstrap_components as dbc
import pandas as pd

```

```

import plotly.graph_objs as go
from dash.dependencies import Input, Output
import dash_table
from app import app
from database import transforms

df = transforms.df

layout = html.Div(
    id='table-paging-with-graph-container',
    className="five columns"
)

@app.callback(Output('table-paging-with-graph-container',
    "children"),
    [Input('rating-95', 'value')
    , Input('price-slider', 'value')
    ])

def update_graph(ratingcheck, prices):
    dff = df

    low = prices[0]
    high = prices[1]

    dff = dff.loc[(dff['price'] >= low) & (dff['price'] <= high)]

    if ratingcheck == ['Y']:
        dff = dff.loc[dff['rating'] >= 95]
    else:
        dff

    trace1 = go.Scattergl(x = dff['rating']
        , y = dff['price']
        , mode='markers'
        , opacity=0.7
        , marker={
            'size': 8
            , 'line': {'width': 0.5, 'color':
'white'}}
        , name='Price v Rating'
    )

    return html.Div([
        dcc.Graph(
            id='rating-price'
            , figure={
                'data': [trace1],
                'layout': dict(
                    xaxis={'type': 'log', 'title': 'Rating'},
                    yaxis={'title': 'Price'},
                    margin={'l': 40, 'b': 40, 't': 10, 'r': 10},
                    legend={'x': 0, 'y': 1},

```

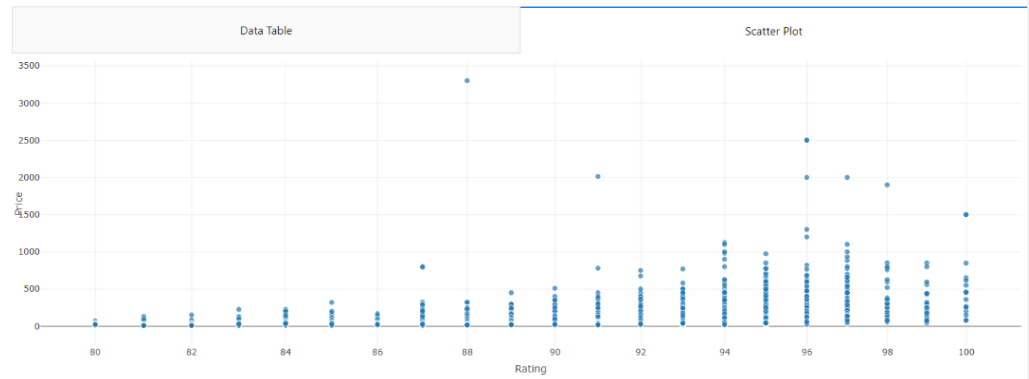
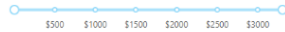
```
    hovermode='closest'
    )
    }
  )
]
```

Wine Dash

Filters

☐ Only rating >= 95

Price Slider



Thanks for checking out my beginners guide to creating a Dash app. You can find the code on github here:

<https://github.com/bendgame/DashApp>

Python

Data Science

Programming

Business

Dash

Medium

About Help Legal