A Comparison Between the Python Libraries Shiny and Plotly

By Ian McKechnie

Monday October 31st, 2022

**Abstract**

Shiny for Python (Shiny) and Dash (the data analysis tool made by Plotly) are both python libraries that allow you to write python code and have it displayed on the web so the user can view and analyze data. Dash was open sourced in 2017 whereas Shiny was launched in the summer in 2022. Both can accomplish good data visualization, but Dash allows greater data configuration and a more robust final application. This comes at a cost as Dash is more complex and has a steeper learning curve. Dash can also be a little slower as it is bigger and uses more system resources to run. Dash appears to scale better and offers a paid version for enterprises which includes more features than the open-source version. Dash looks to be the better choice *for now* as there is more support and there’s more features. Shiny is good for smaller projects and if you want them done a little quicker. Maybe in a year or two if more features get added and the community support improves and grows because right now there’s almost no answers outside the docs when you google questions for Shiny issues.

**Quick Facts**

**Activeness**: Both libraries are actively being maintained and updated with bug fixes and small features additions daily.

**Design:** Both are very well designed. Shiny is a little simpler so it’s faster to pick up. Dash is more complex but allows you to do more with it and configure more individual elements.

**Documentation:** Dash has better documentation. It’s larger and older so the documentation is more robust and length. Shiny is newer so its documentation is not as extensive. The documentation is easier to follow and more interactive but there is less of it for each individual component. There is also less community support for Shiny since it’s new so the documentation is almost the entirety of the resources you can consult to learn it and solve bugs. Dash has a community form where there are lots of questions and answers, so you have a better chance of finding your error and a solution for the issue being searched.

**Extensibility:** Dash is more extensible as you can pay for an enterprise version that adds lots of features to help scale.

**Performance:** Dash is a lot slower than Shiny. Shiny has less components and is faster which is good if people will be running it locally and don’t have fast computers.

**Popularity:** Dash is more popular and has more stars on GitHub. This is probably due to being larger, used more in industry, and being around longer. Shiny is smaller, but Shiny for R is fairly big (still smaller than Dash) so it will take time for that popularity to switch over to Shiny for python

**Stability:** Hard to say. Dash has 663 Open issues on GitHub whereas shiny has 63. So Shiny appears to be more stable but it’s also in its alpha version. I think people have not been using it enough to really find out all the errors in comparison to Dash, which is used by millions of people a month.

**Usability**: Shiny is fairly easy to pick up as the docs are readable in less than a couple hours. Dash’s is documentation much more extensive and could take you at least a day to fully read and comprehend each component. So Shiny has better immediate usability.

**Shiny for Python**

**Introduction**

Shiny is focused on creating graphs and images for web applications. It can be installed through pip. It has a lot of typical front end language features like headings and text features for displaying static text on websites. Done with the following methods:

Text

Description automatically generated

When you run Shiny it runs on <http://localhost:8000> on your browser. You can download a VSCode plug in and see a sample browser as seen in figure 1 below. This allows faster development.

Graphical user interface, application, Teams

Description automatically generated

Figure

Shiny is based on reactive programming. Reactive programming is where it only re-executes code based on if an input is modified.

**Reactive programming in Shiny**

There are three methods that you can choose from so when an input changes your Shiny application can react to the input. They are *reactive.Value, reactive.Calc, reactive.Effect*.

reactive.Calc

When you want a function to be calc reactive, you add the decorator @reactive.Calc above it (ex. figure 2)

Logo

Description automatically generated with low confidence

Figure 2

What reactive.calc does is cache its most recent value and returns that value until the calculation is invalidated (So only when a new input comes it that would change the output it will rerun the function). This means the amount of work the method needs to do is reduced since it doesn’t not need to recalculate what’s in the method every time the page is updated. This results in a faster and snappier page.

You can think of this as the method only returns cached results until the input changes, then it reruns and caches that new value that it’s displaying to the user.

reactive.Effect

This decorator is applied the same as reactive.Calc and it runs when the inputs change. This differs from reactive.Calc in two mains ways

1. Reactive.calc is used for its return value; its meant to be called from another reactive.calc or reactive.effect which uses the return value from the first. A reactive.effect does not have a return value and cannot be called like a function. Once created, it continues to exist, and it automatically executes after is has been invalidated.
2. As the name implies reactive.effect is used for its side effect. A side effect is when a function modifies state other than its return value. These side effects may include writing to the file, printing to console, or modifying a global variable.

reactive.object

They are Shiny objects that when their data changes, it causes reactive methods to change.

@output

There is also the @output decorator. This allows methods not decorated with reactive.effect to be reactive. An example is

@output

@render.text

**def** txt():

**return** f"The value of x is {input.x()}"

This returns a string that is based on the value of x which is an input from the user. It’s wrapped with a @render.text and @output decorators. Each time the reactive inputs change, it re-executes and puts the new string in the message queue.

**User interaction**

Shiny for Python has a set of features that allow the data to be interactive. So, the page being built can have interactive filters that the user can toggle, slide, and change to alter what and how data is being shown to the user. An example is shown below in figure 1. The code on the left implements an interactive slider. That slider has numbers which when selected, displays its square below in real time.

Graphical user interface, text, application

Description automatically generated

Figure

The list of interactive features are as follows

Graphical user interface, text, application

Description automatically generatedGraphical user interface, application

Description automatically generated

These inputs are not very complex to create. The entire set of inputs above is only 26 lines line including the text labels above each element. The inputs can be made reactive, so when a value is changed by the user on the GUI it will update other elements that use the data associated with it.

**How to get data**

There are a few ways to get data into Shiny. The first is through csv files. You can read in the csv file with pandas (a data analysis library in python) and then display the csv by filtering the pandas data frame with inputs from your user or hard coded filters before displaying it.

There’s also the option to upload and download files. You can also get data from APIs using the standard pandas’ csv\_reader method from python. Note, there are more ways to get data into Shiny, the ones listed above seemed most relevant to how GBAD gets and distributes data from what I have read so far.

**Noticeable Cons of Shiny**

The only documentation is on the Shiny for Python website. Since its introduction in the summer of 2022, there has not been a lot of external resource creation outside the Shiny website. This can make it hard to solve problems as if you have a bug that an example doesn’t show how to fix, it can be hard to find a solution to it.

**Noticeable Pros of Shiny**

This would be great for making interactive web pages for the GBAD website. If we wanted to show off the API more and allow researchers to comb through data quickly on the website, this would be a great tool. You can add all sorts of tools for user input to charge how the data is being displayed on the GUI to the user. This could be a great tool for showing off to users on the GBAD website.

It's also very quick to pick up. How you create backend and frontend elements is very intuitive and quick to pick up. If you have an intermediate understanding of Python, you could have a simple locally hosted Shiny App in a few hours with no previous Shiny experience.

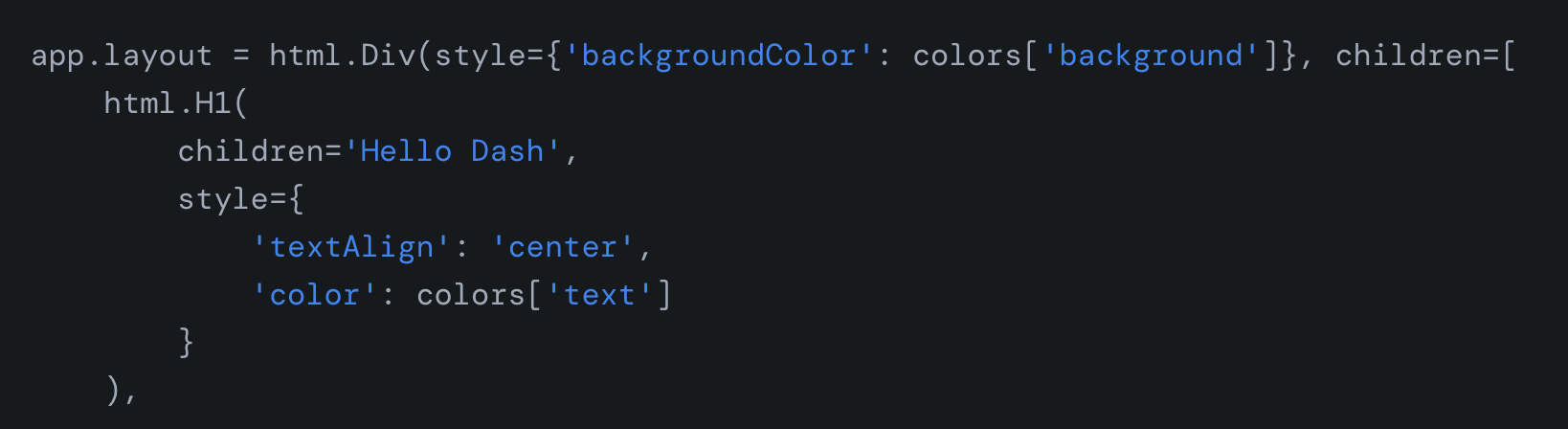
***Plotly***

**Introduction**

Plotly (the company) created the open-source *Dash* library for data analysis with python. There’s a paid enterprise version and free open-source version. Dash requires you to run and write the application in a similar manor to Shiny where you write your python code inside a file named app.js and run it. Dash then automatically deploys a local webpage where you can view your work.

**Styling the application**

Dash applications are styled in the usual html format with some minor syntax changes. There is a method in the Dash application called app.layout for which you pass an argument called *children* and set it equal to an array of html elements such as graphs, divs, headers, and more containing all the style you want to appear on your interface. Each of these elements are python elements that translate to html on the page. Each of these elements has children arguments of their own for each elements CSS allowing a lot of styling to be done on the app. Example in figure 3



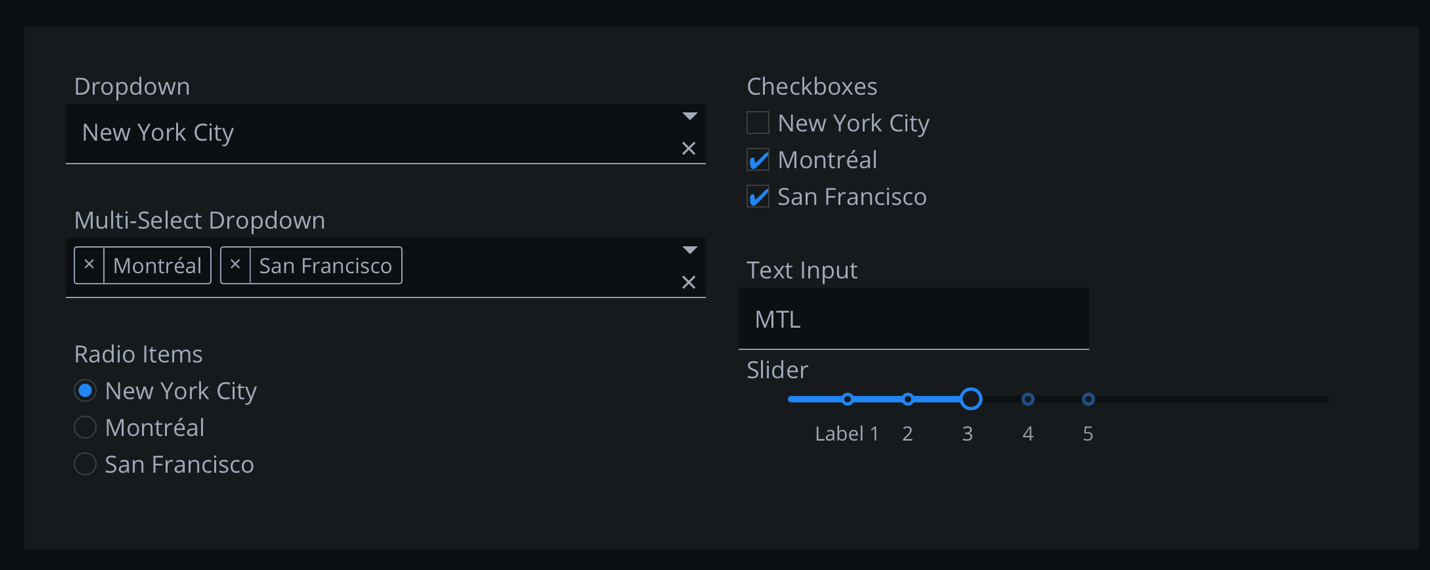
Figure

**Creating components**

All the visual components are shown using the app.layout as outlined above. To add graphs, tables, and other data visualizer tools, you can write functions to handle it. These functions can get and format the data, then return an html element visualizing the data. Then all that’s needed is to call that function from the app.layout and it will show on screen.

**Interactive components**

Dash also has a lot of similar interactive components to Shiny, figure 4 shows some of them.



Figure

**“Reactive Programming”**

Dash allows you to make your visual elements reactive to user input. You can add @app.callback tags above functions that you want to have interactive elements. The app callback’s accepts input and output arguments and those are used connect the input and output html elements. An example of this is

app.layout = html.Div([

dcc.Graph(id='graph-with-slider'),

dcc.Slider(

df['year'].min(),

df['year'].max(),

step=None,

value=df['year'].min(),

marks={str(year): str(year) for year in df['year'].unique()},

id='year-slider'

)

])

@app.callback(

Output('graph-with-slider', 'figure'),

Input('year-slider', 'value'))

def update\_figure(selected\_year):

filtered\_df = df[df.year == selected\_year]

fig = px.scatter(filtered\_df, x="gdpPercap", y="lifeExp",

size="pop", color="continent", hover\_name="country",

log\_x=True, size\_max=55)

fig.update\_layout(transition\_duration=500)

return fig

The outputs’ first argument is *graph-with-slider* which is the id of the graph element. This is the same id as the dcc.graph element being shown on the web page. The second is *figure* which is created by the update\_figure function and returned to the dash application. Finally, as the *value* of the doc slider changes, the update\_figure function is called with the *value* as an argument.

**Getting Data in Dash**

Can be done through pandas like Shiny. Dash does not appear to support NumPy.

**Noticeable Cons of Dash**

Dash appears to have a steeper learning curve that Shiny. So, it will take people longer to understand how to use Dash proficiently than Shiny. Especially if you are inexperienced with Python as there are a lot of different python features and elements being used in Dash and that could be overwhelming to beginners.

**Noticeable Pros of Dash**

You can do a lot more with Dash than you can with Shiny. This comes from it being older, so more features have been added. So, it may take longer to become proficient with Dash than Shiny but you can create more robust and complex programs with it.

There is also more room to grow with Dash since it has a paid enterprise tier. This could be useful in the future we need; more features, no code development, and powerful dashboards to handle pages created with Dash.