Dash User Guide and Documentation

Table of Contents

What's Dash?

Introduction to Dash

Dash is a productive Python framework for building web applications.

Written on top of Flask, Plotly.js, and React.js, Dash is ideal for building data visualization apps with highly custom user interfaces in pure Python. It's particularly suited for anyone who works with data in Python.

Through a couple of simple patterns, Dash abstracts away all of the technologies and protocols that are required to build an interactive web-based application. Dash is simple enough that you can bind a user interface around your Python code in an afternoon.

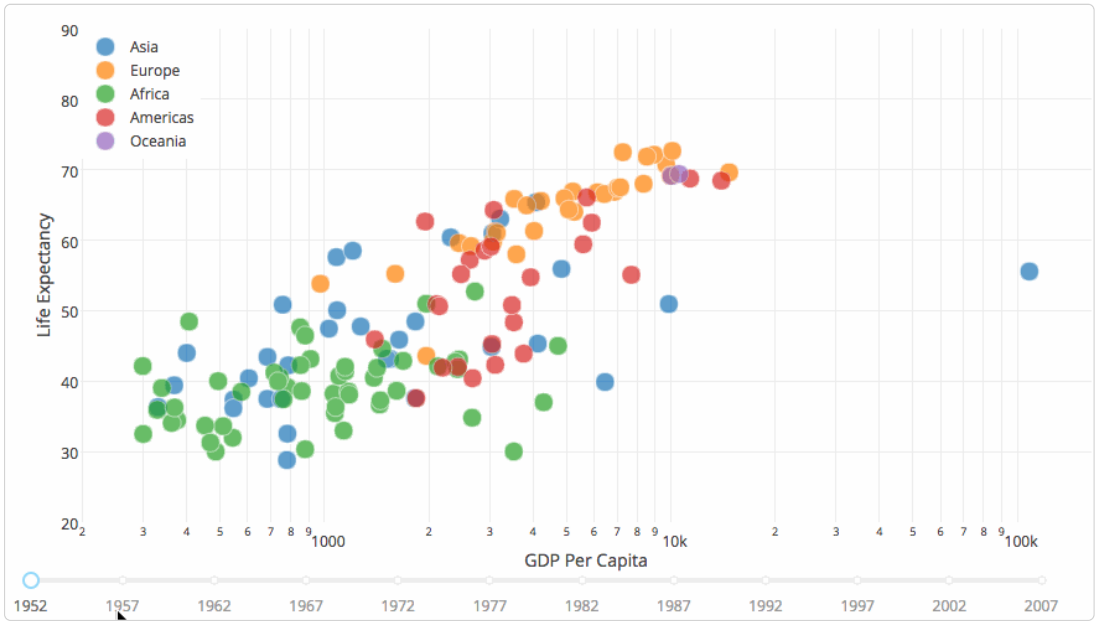
Dash apps are rendered in the web browser. You can deploy your apps to servers and then share them through URLs. Since Dash apps are viewed in the web browser, Dash is inherently cross-platform and mobile ready.

There is a lot behind the framework. To learn more about how it is built and what motivated Dash, watch our talk from Plotcon below or read our announcement letter.

Dash is an open source library, released under the permissive MIT license. Plotly develops Dash and offers a platform for easily deploying Dash apps in an enterprise environment. If you're interested, please get in touch.

Dash Gallery

Getting Started

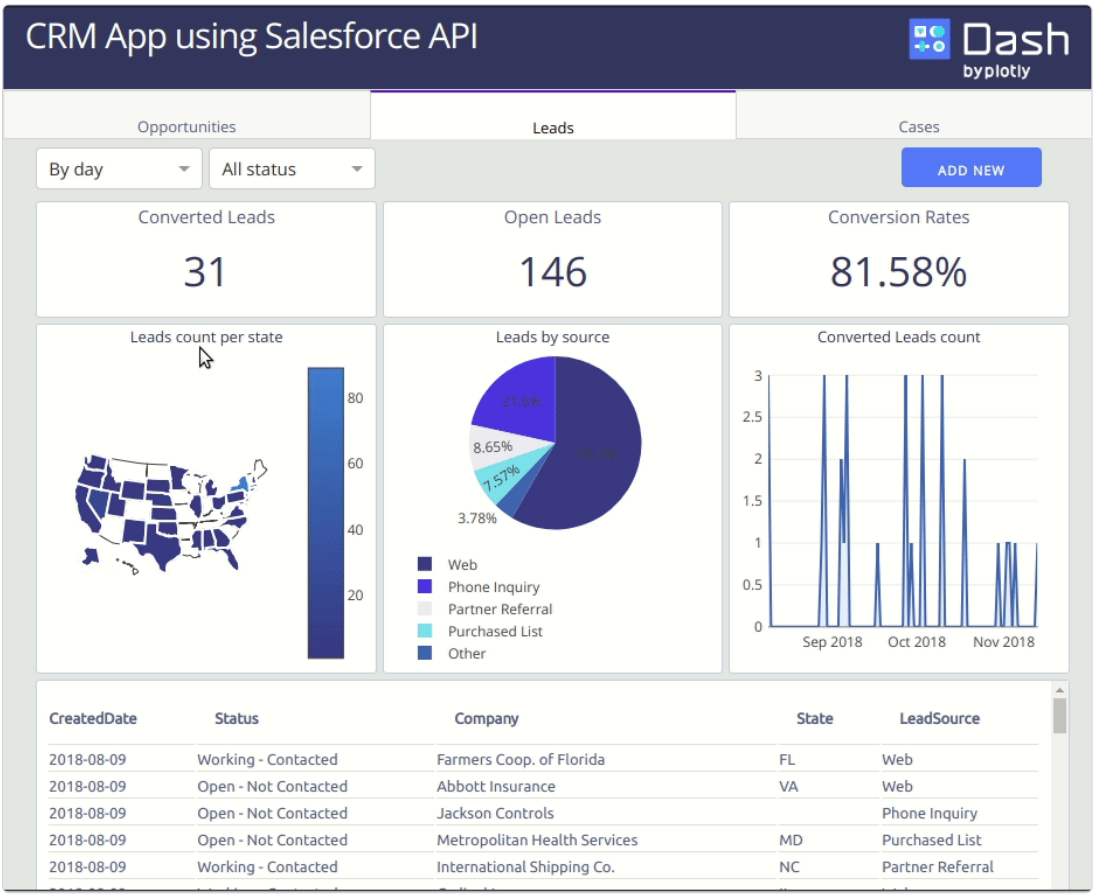


View the getting started guide

The Dash Getting Started Guide contains many applications that range in complexity.

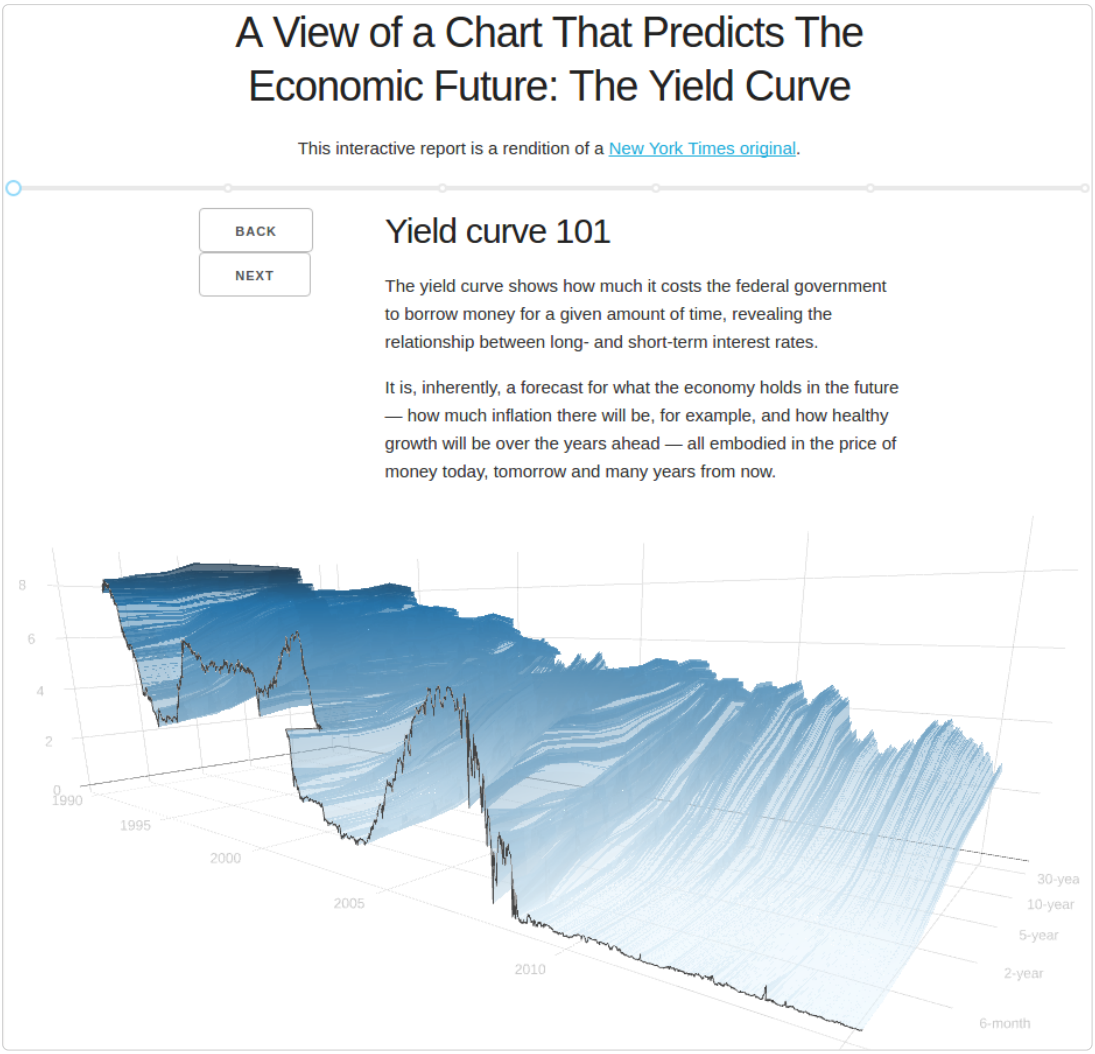
The first interactive app that you'll create combines a `Slider` with a `Graph` and filters data using a Pandas `DataFrame`. The `animate` property in the `Graph` component was set to `True` so that the points transition smoothly. Some interactivity is built into the `Graph` component, including hovering over values, clicking on legend items to toggle traces, and zooming into regions.

Finance



Salesforce Dashboard | Source code

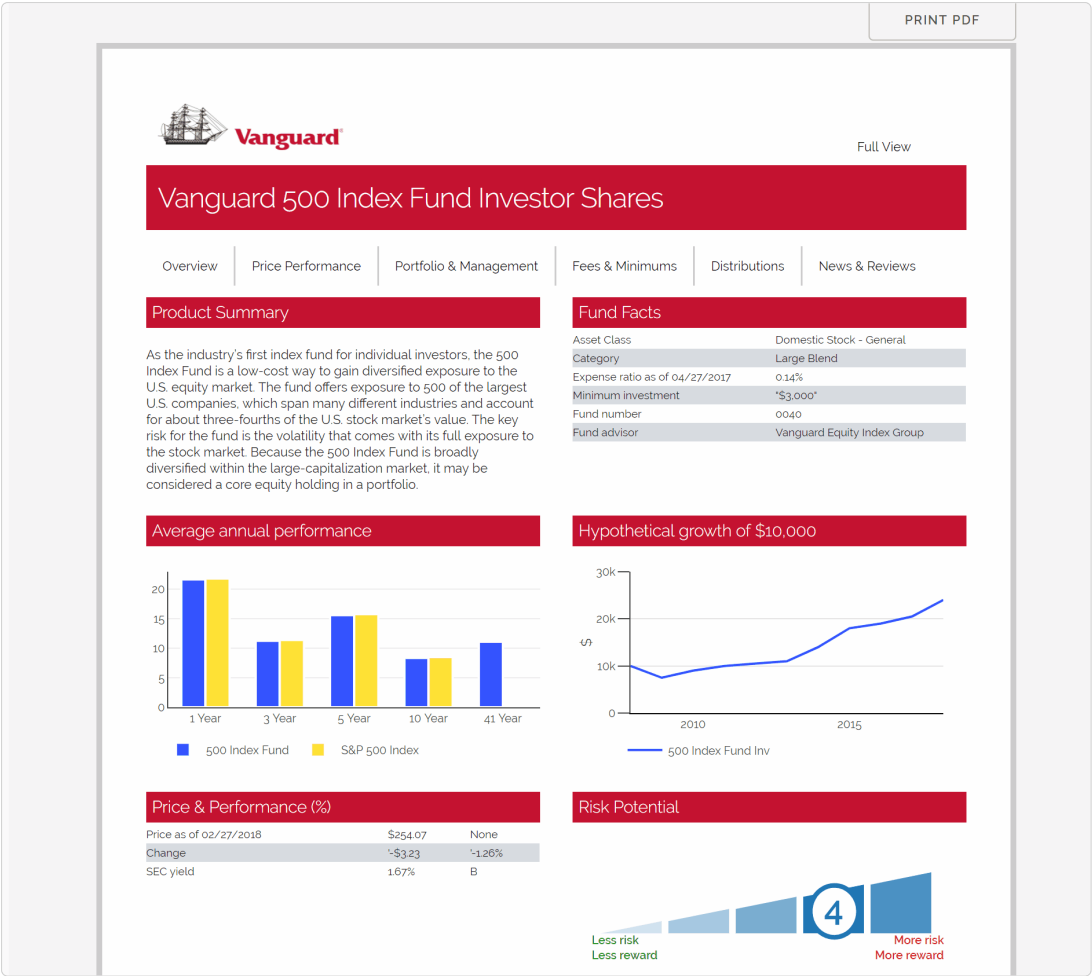
This app uses Salesforce API in order to implement a custom CRM dashboard. The API is used via the module Simple-Salesforce and allows you to retrieve and to push data.



3-D Yield Curve | Source code

This Dash app adapts the New York Times' excellent report: A 3-D View of a Chart That Predicts The Economic Future: The Yield Curve.

Dash comes with a wide range of interactive 3-D chart types, such as 3-D scatter plots, surface plots, network graphs and ribbon plots. View more 3-D chart examples.

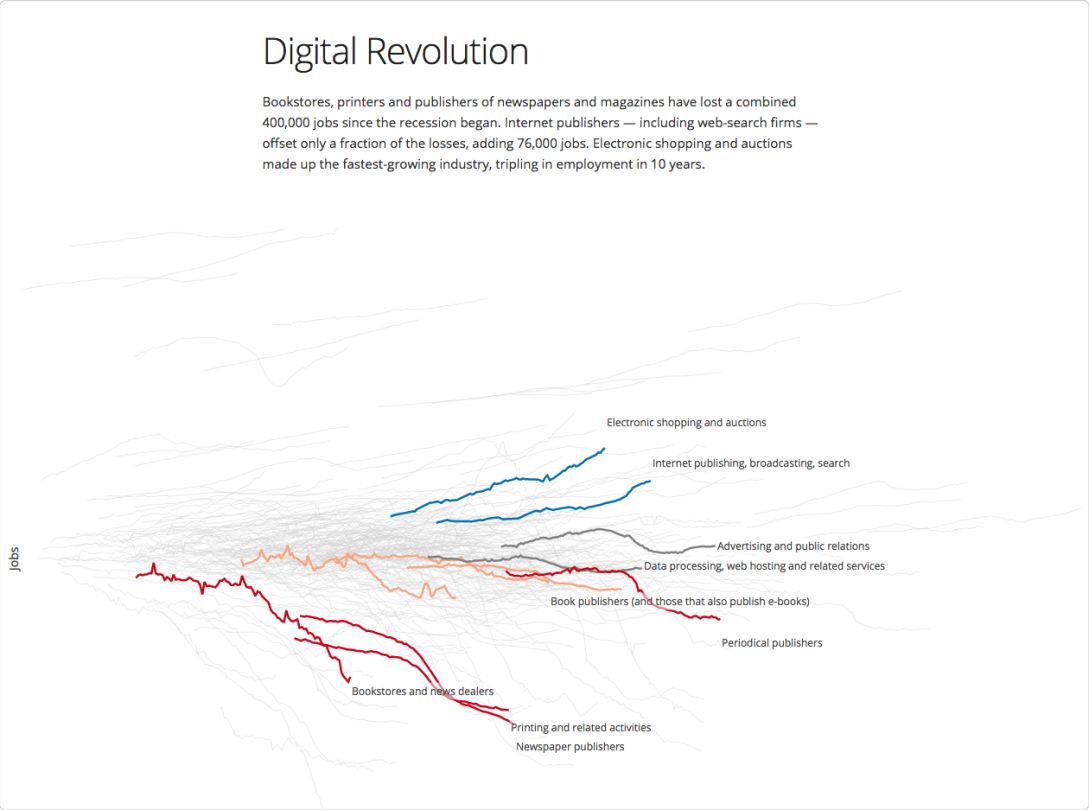


Vanguard Report | Source code

This app recreates the look and feel of a Vanguard report. It includes a Print to PDF button and the styles were optimized to look good on the web and in PDF form.

The charts in the report on the web version are interactive. You can hover over points to see their values and zoom into regions. Since this report was built on top of Dash, you could adapt this report to include even more interactive elements, like a dropdown or a search box.

With PDF styles, you can hide and show elements depending on whether the app is being viewed in the web browser or in print, using the same framework for both the rich interactive applications and the static PDF reports.



Recession in 255 Charts | Source code

485 lines of Python code, including text copy.

This Dash app was adapted from The New York Times' excellent report: How the Recession Reshaped the Economy in 255 Charts.

This Dash app displays its content linearly, like an interactive report. The report highlights several notable views of the data and then invites the user to highlight their own regions at the end. This method of highlighting views is a great strategy for walking your readers through your complex analysis.

The height of the charts was specified in viewport units (`vh`), scaling the size of the chart to the height of the screen. It looks great on monitors big and small.

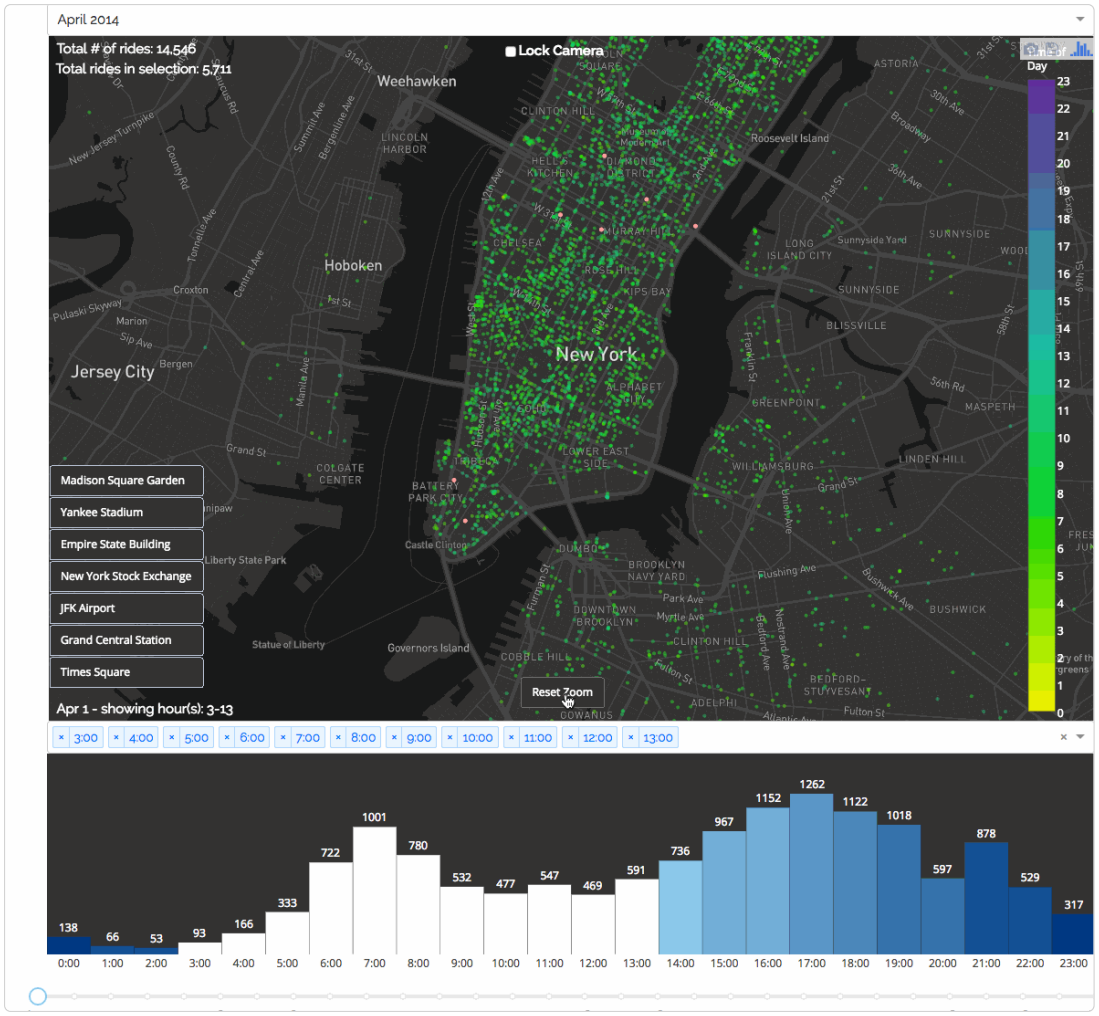
The text in the application is centered and its width is restricted to improve the reading experience. The graphs are full bleed: they extend past the narrow column of text to the edges of the page.

Energy and Transportation

Natural Gas Well Production | Source code

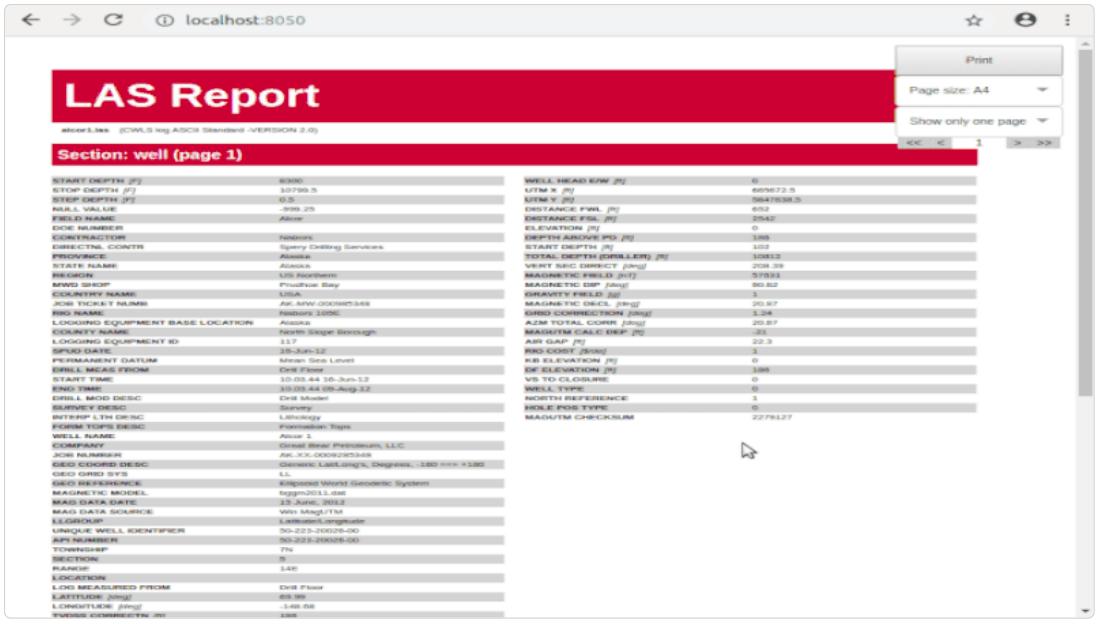
This Dash app displays well data from New York State. As you hover over values in the map, a time series is displayed showing production values over time. As you change the years in the range slider, the aggregate time series is updated with the sum of all production over time. The histogram chart is also selectable, serving as an alternative control for selecting a range of time.

This application is also mobile-friendly. Dash apps are built and published in the Web, so the full power of CSS is available. The Dash core team maintains a core style guide here that includes a responsive 12 column grid.



NYC Uber Rides | Source code

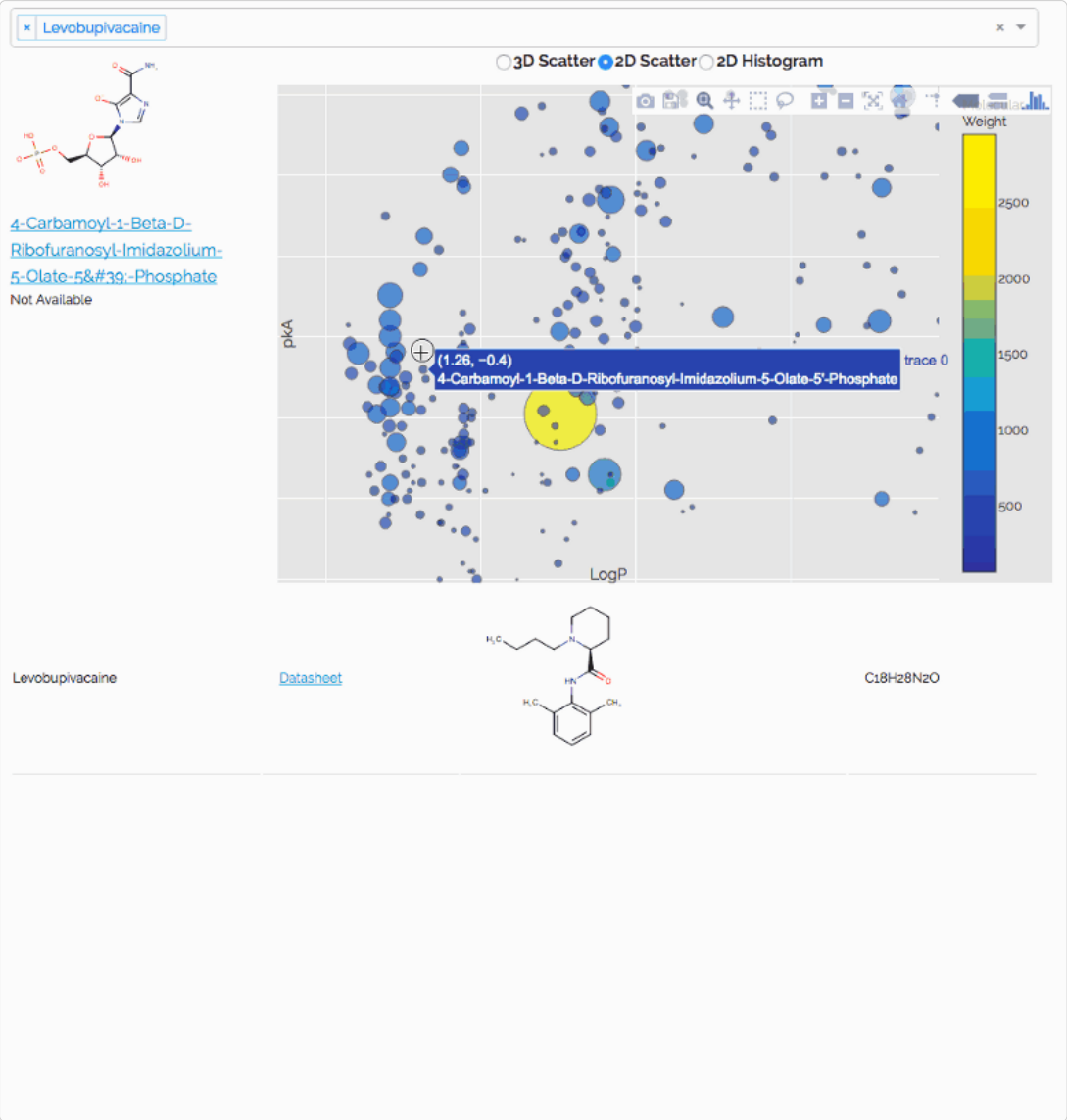
This app displays all of the Uber rides in New York City in 2014. The original datafile is almost 500MB large and all of the filtering is done in memory with Pandas. Buttons on the chart itself highlight different regions in the city.



LAStoDash | Source code

This dash app takes a Log ASCII Standard (LAS) file, and generates a web report application, making it easy to share. The report can be printed.

Life Sciences

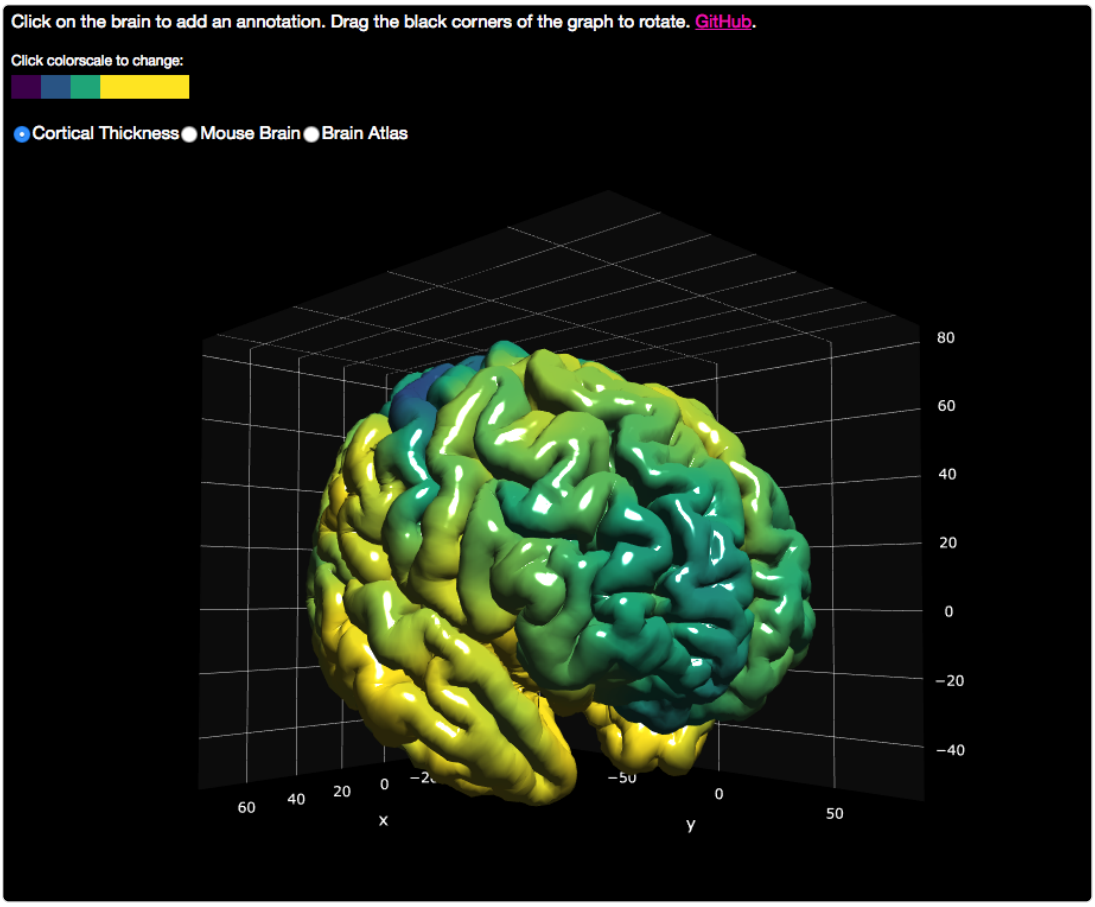


Drug Precursors | Source code

This app displays a description of the drug as you hover over points in the graph.

Selecting drugs in the dropdown highlights their position in the chart and appends their symbol in the table below.

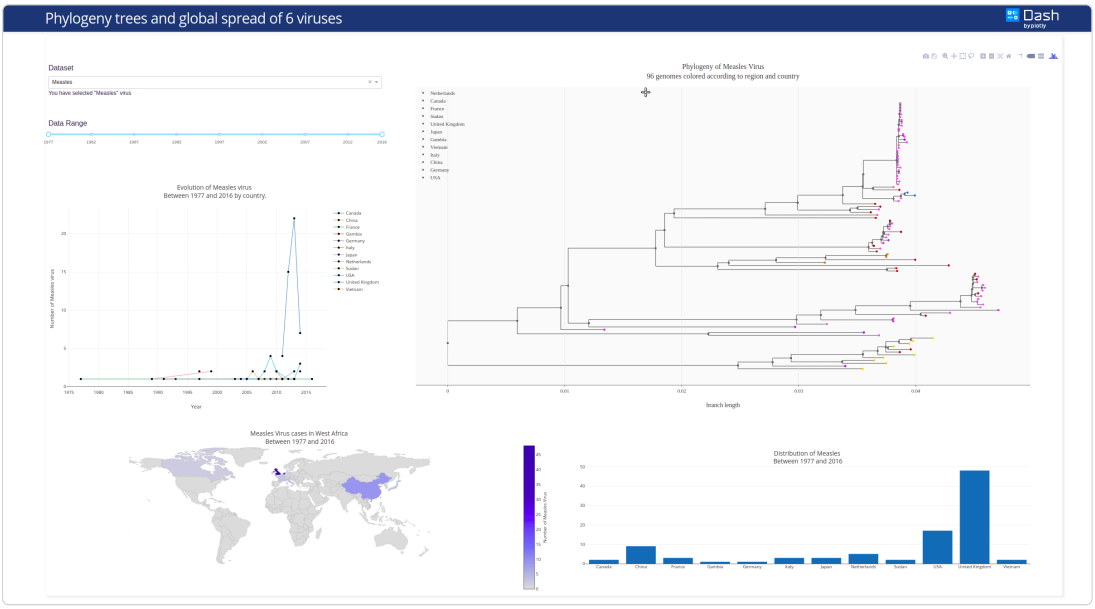
Built in a few hundred lines of Python code.



MRI Reconstruction | Source code

🐭 Explore human and mice brains in 3-D.

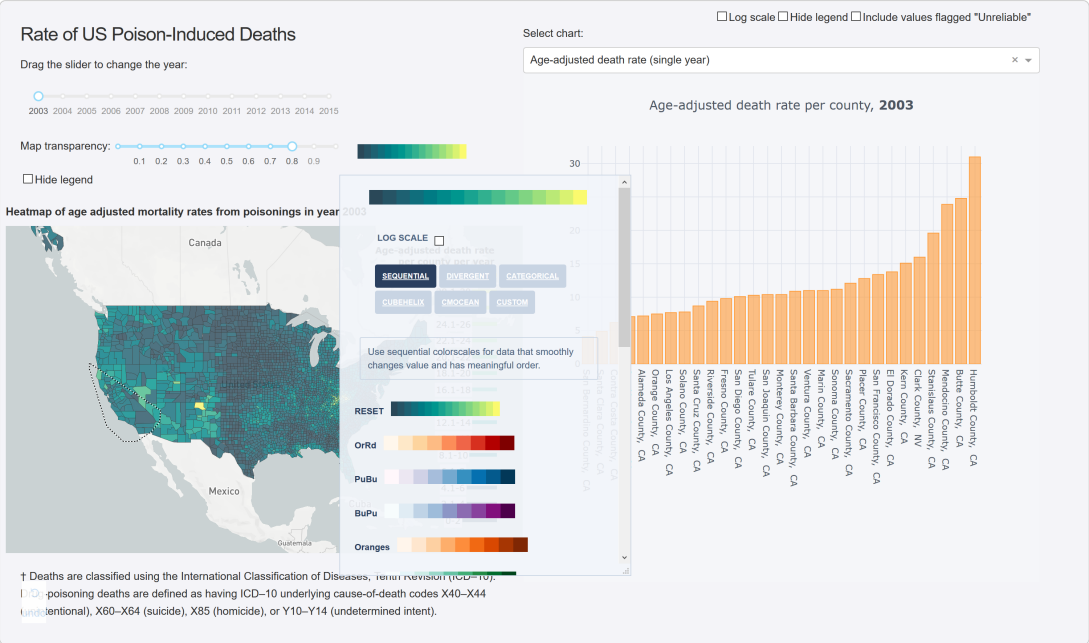
Add interactive labels to points on the brain surface and change the surface colorscale.



[Phylogeny trees and global spread of six viruses](#) | [Source code](#)

Interactively explore the propagation of six viruses, by time and/or by location. In the online app, you can select a virus to display its evolution as a phylogeny tree, along with a map and time series of the virus's global spread.

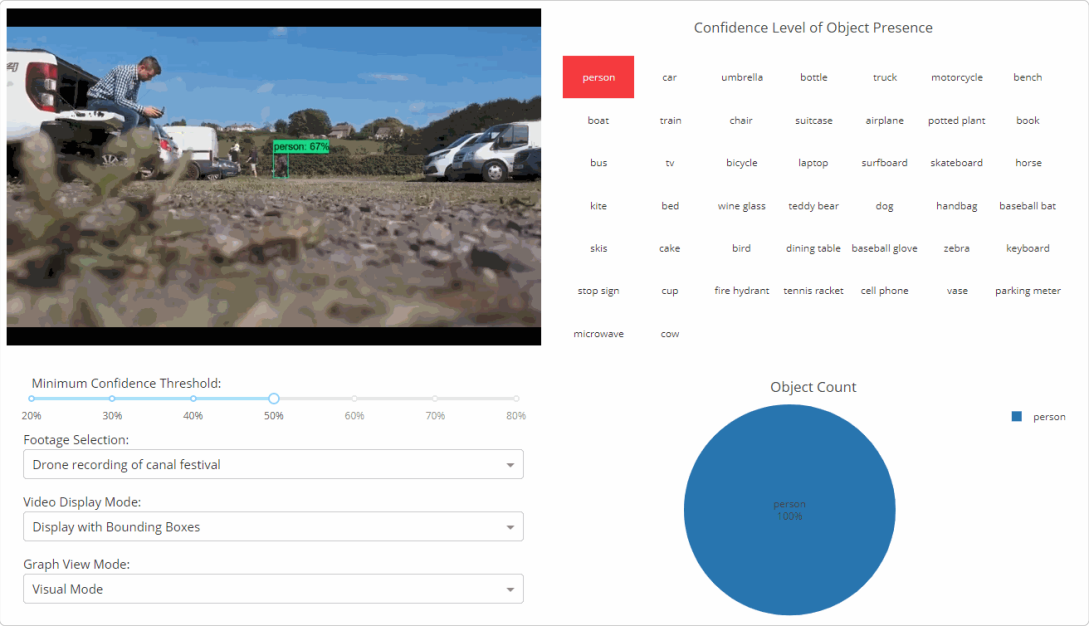
Government & Public Health



[US Opioid Epidemic](#) | [Source code](#)

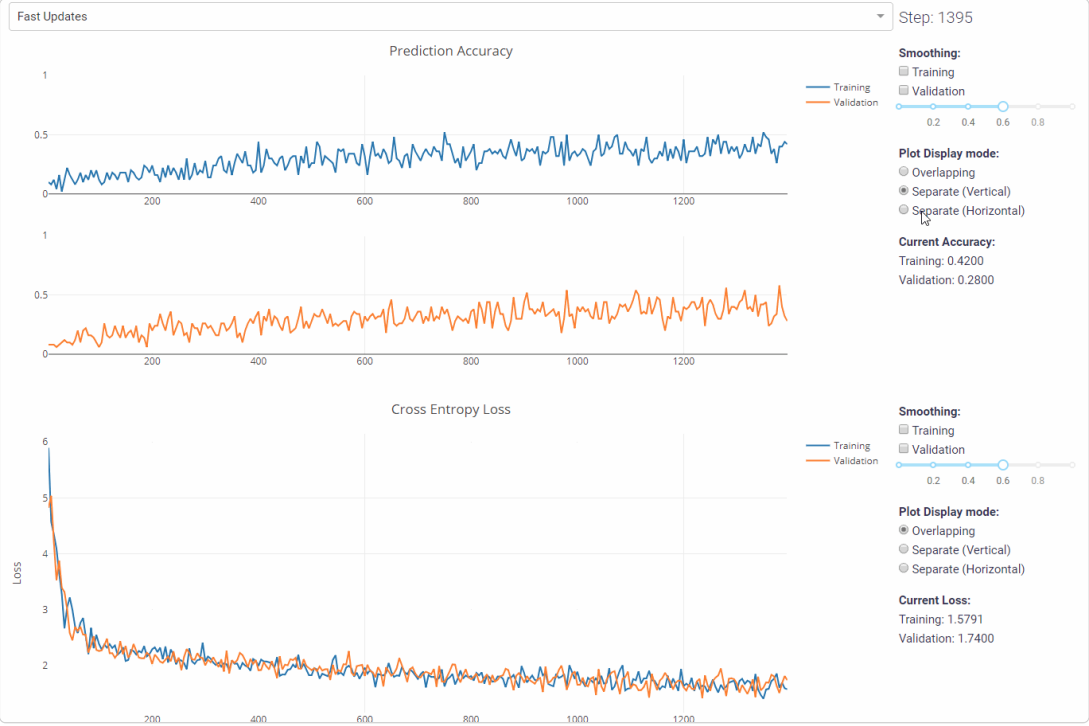
Interactively explore the effect of the opioid epidemic in North America.

Machine Learning & Computer Vision



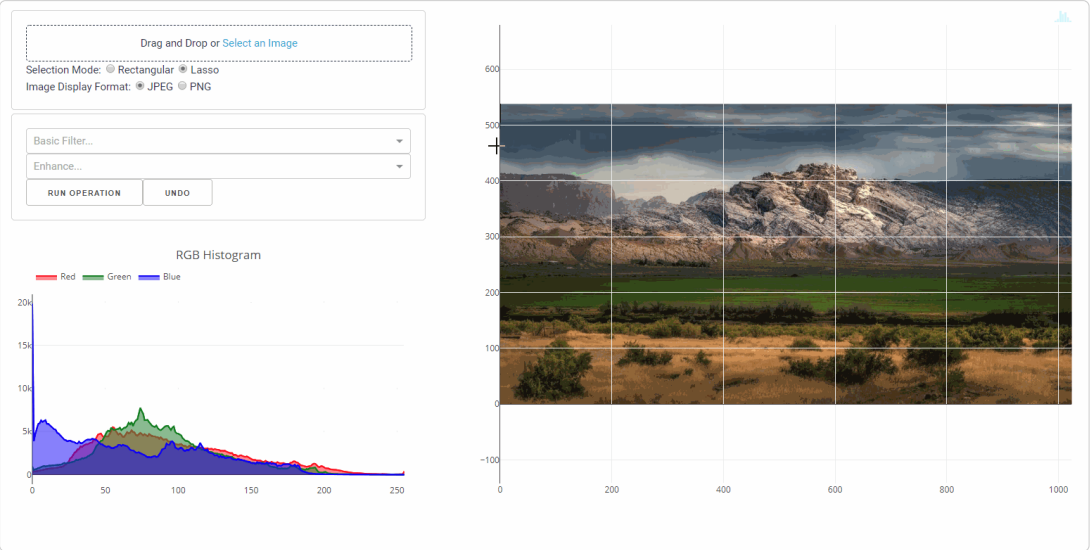
Object Detection | Source code

This object-detection app provides useful visualizations about what's happening inside a complex video in real time. The data is generated using MobileNet v1 in Tensorflow, trained on the COCO dataset. The video is displayed using the community-maintained video component.



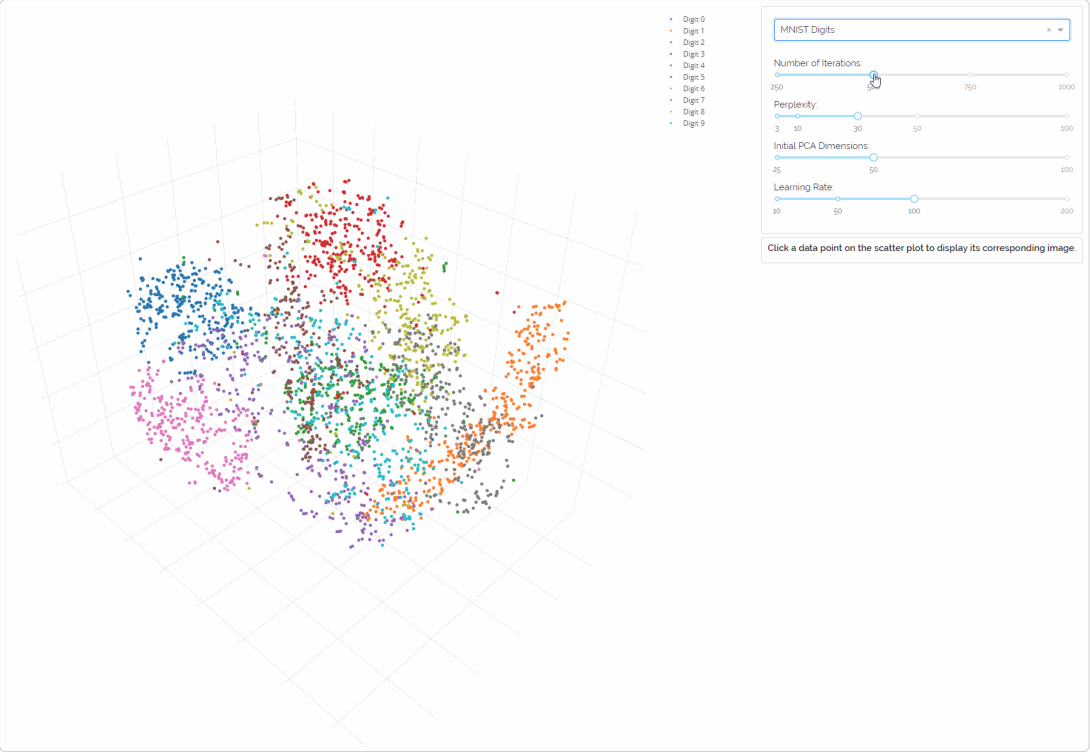
Visualize Model Training | Source code

Tracking accuracy and loss is an essential part of the training process for deep learning models. This real-time visualization app monitors core metrics of your Tensorflow graphs during the training so that you can quickly detect anomalies within your model.



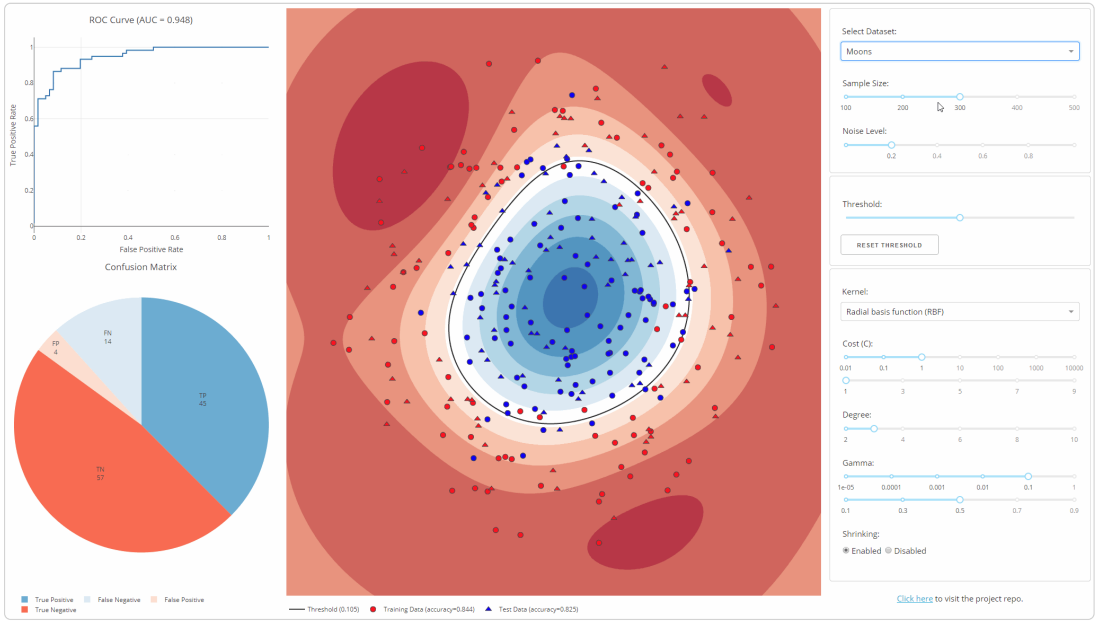
[Image Processing](#) | [Source code](#)

This app wraps Pillow, a powerful image processing library in Python, and abstracts all the operations through an easy-to-use GUI. All the computation is done on the back-end through Dash, and image transfer is optimized through session-based Redis caching and S3 storage.



Interactive t-SNE | Source code

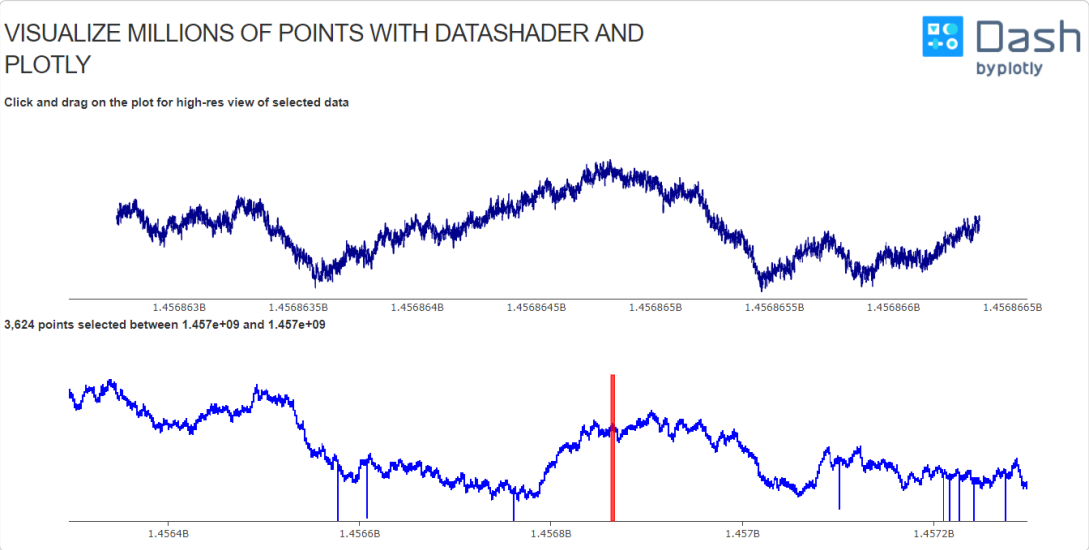
t-SNE is a visualization algorithm that projects your high-dimensional data into a 2D or 3D space so that you can explore the spatial distribution of your data. The t-SNE Explorer lets you interactively explore iconic image datasets such as MNIST, and state-of-the-art word embeddings such as GloVe, with all the computation done ahead of time. Data point previews and graphs help you better understand the dataset.



Explore SVMs | Source code

This app lets you explore support vector clustering (a type of support vector machine) with UI input parameters. Toy datasets and useful ML metrics plots are included. It is fully written in Dash + scikit-learn.

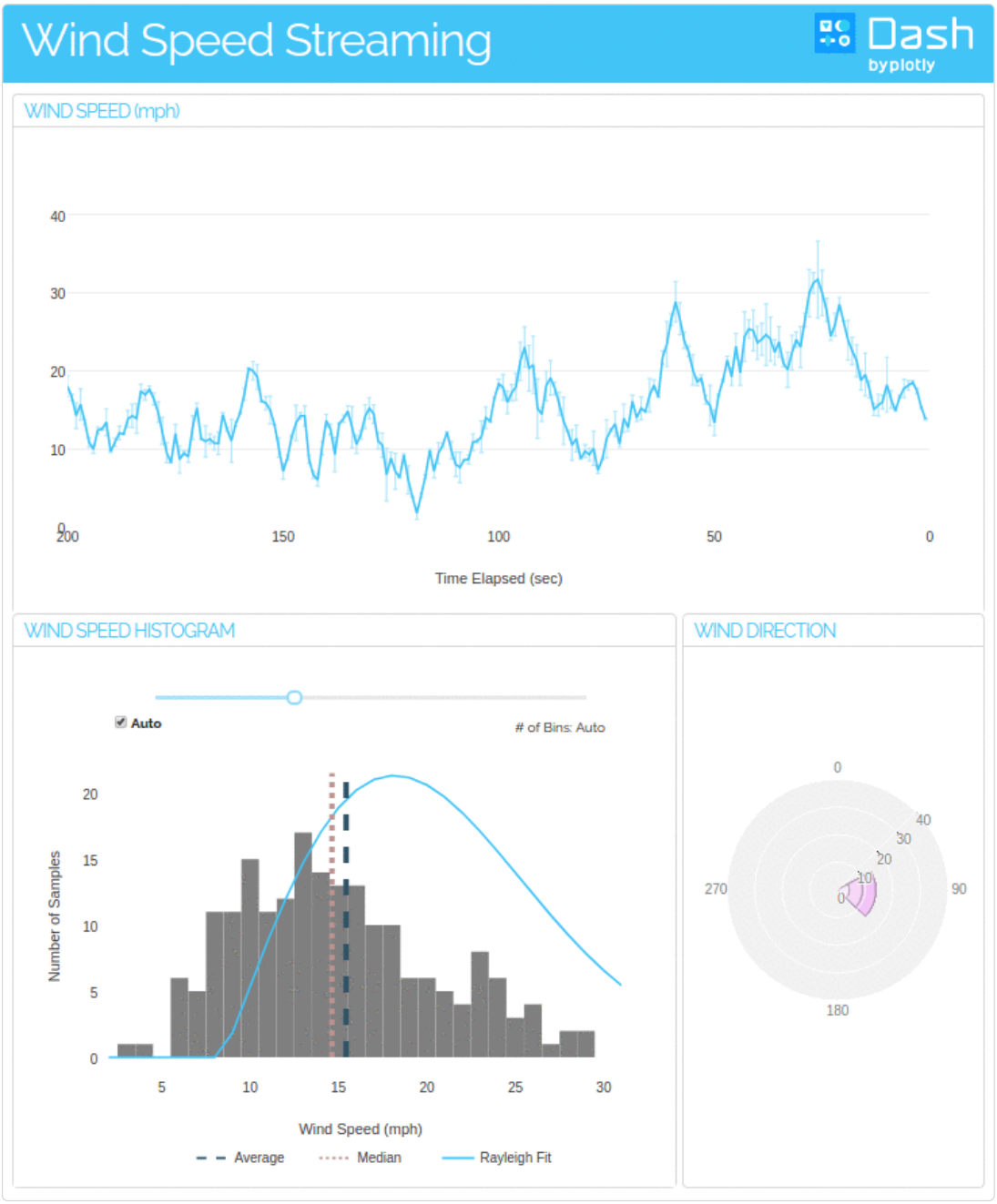
Big Data



Dash Datashader | Source code

Visualize hundreds of millions of points interactively with Dash and Datashader.

Live Updates



Wind Speed Measurement | Source code

This app continually queries a SQL database and displays live charts of wind speed and wind direction. In Dash, the dcc.Interval component can be used to update any element on a recurring interval.

Forex Trader Demo | Source code

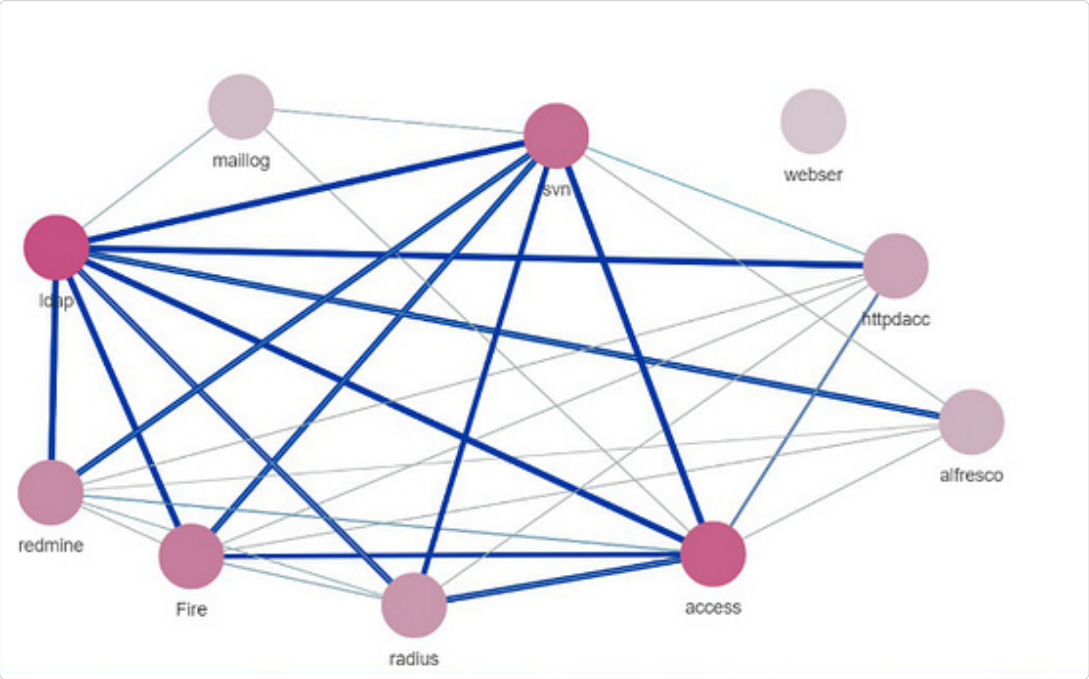
This app continually queries csv files and updates Ask and Bid prices for major currency pairs as well as Stock Charts. You can also virtually buy and sell stocks and see the profit updates.

Component Libraries

Dash DataTable | Source code

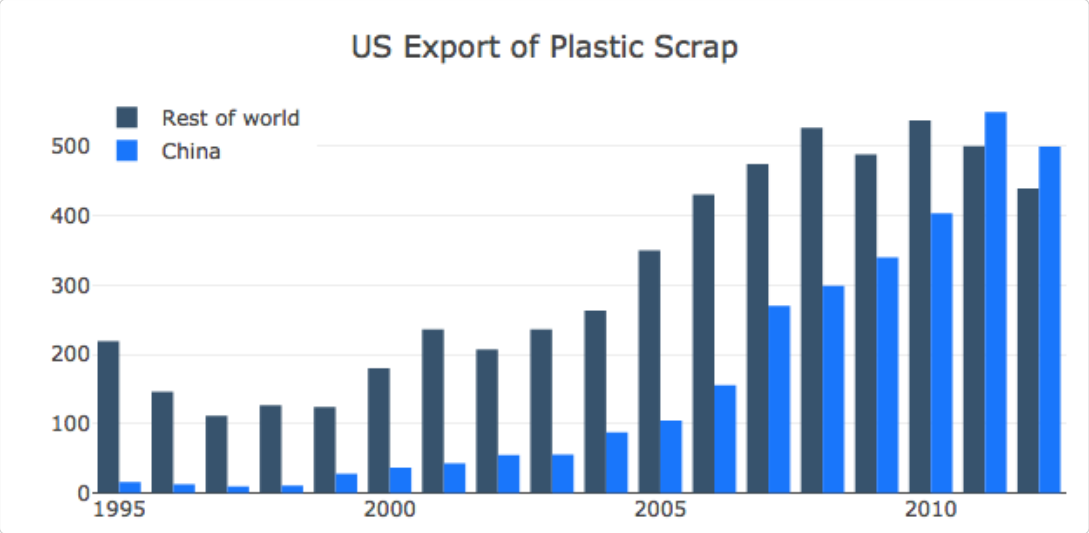
Dash provides an interactive `DataTable` as part of the `data-table` project. This table includes built-in filtering, row-selection, editing, and sorting.

This example was written in ~100 lines of code.



Dash Community Components

Dash has a plugin system for integrating your own React.js components. The Dash community has built many of their component libraries, like Video Components and Large File Upload. View more community-maintained components and other projects in the Dash Community Forum's Show and Tell Thread.

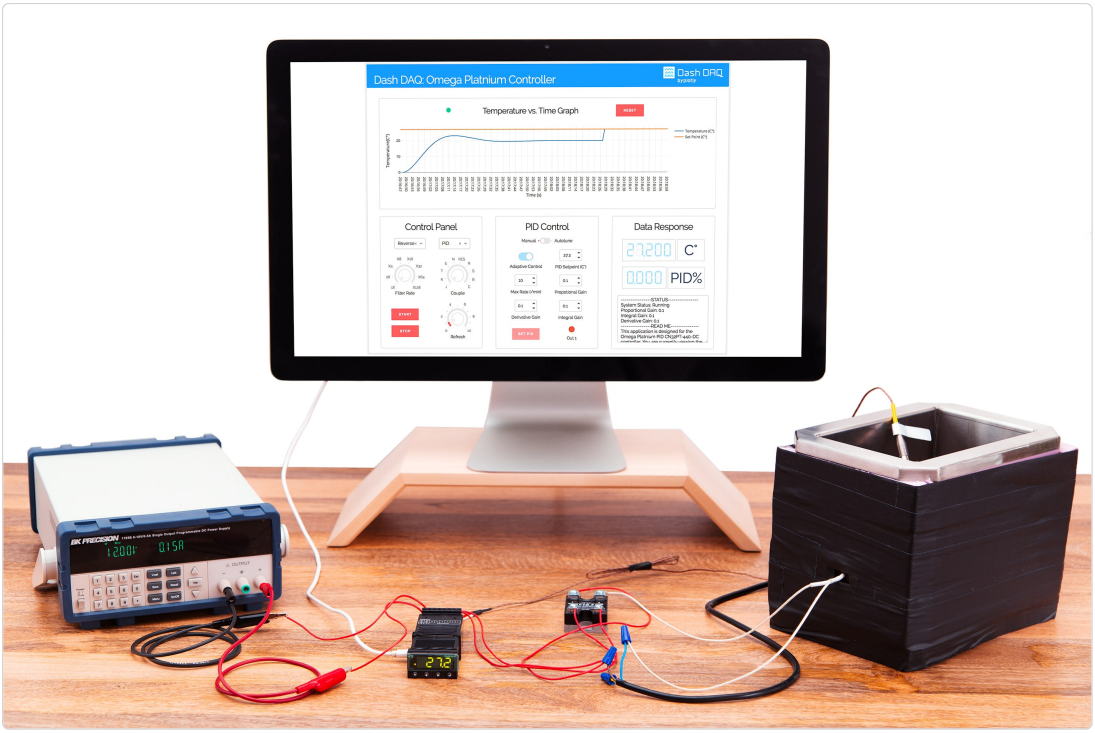


Dash Core Components | Source code

Dash comes with a set of rich components like sliders, dropdowns, graphs, and more. View the official Dash documentation to learn more.

Data Acquisition (DAQ)

Dash DAQ is a Dash component library for building custom data acquisition interfaces with Dash in Python. Learn more about Dash DAQ.

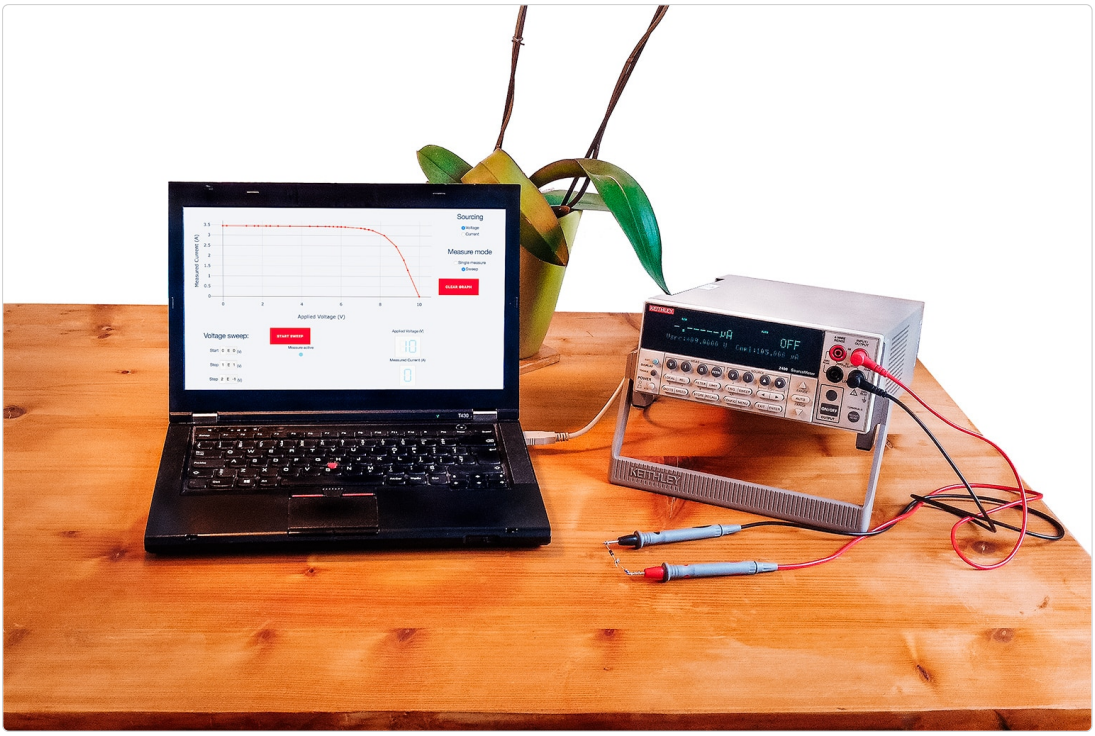


Omega CN32PT-440-DC PID Controller | Try the app

Let's heat things up with Dash DAQ! With this application, we use Python to monitor and manage a PID controller connected to a water heater 🔥

Wireless Arduino Robot | Try the app

We love our robots here at Plotly! This Dash DAQ app wirelessly controls Sparki, an Arduino-based robot 🤖

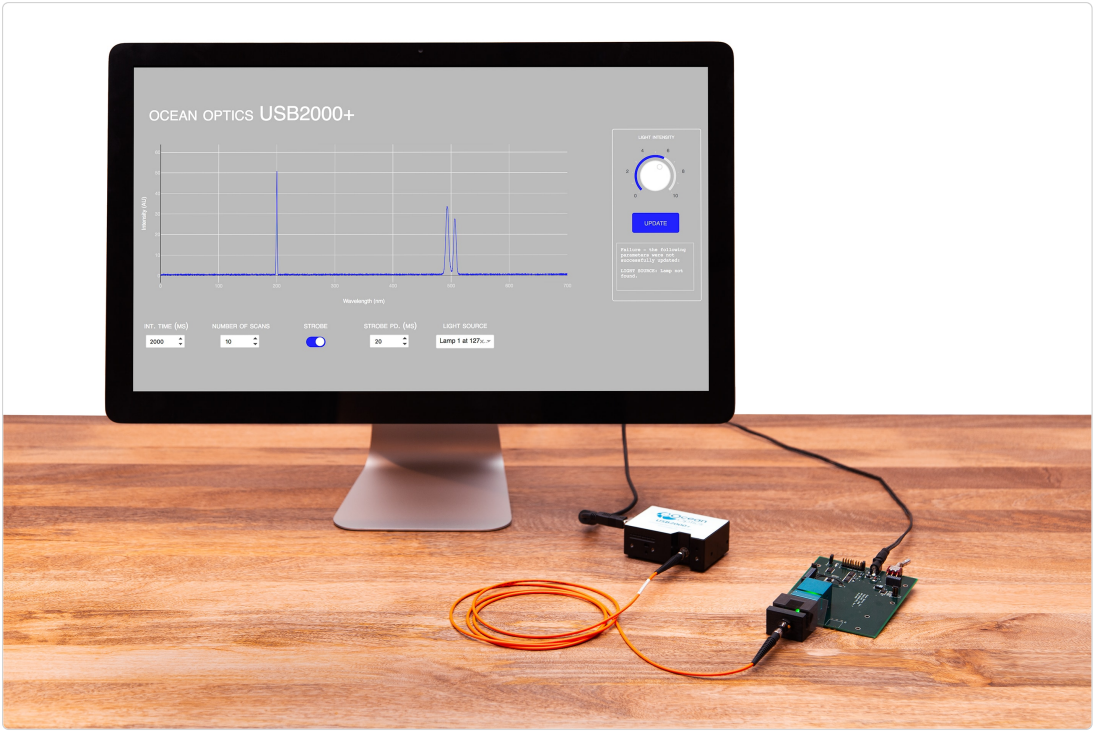


I-V curve tracer with a Keithley 2400 SourceMeter | Try the app

With this Dash DAQ application written in Python, you can create UI components to interface with a Keithley 2400 SourceMeter.

Control a Robotic Arm | Try the app

Dash DAQ's GUI components let you interface with all the robot's motors and LED, even from a mobile device… just as if it were a real remote control! 🤖



Ocean Optics Spectrometer | Try the app

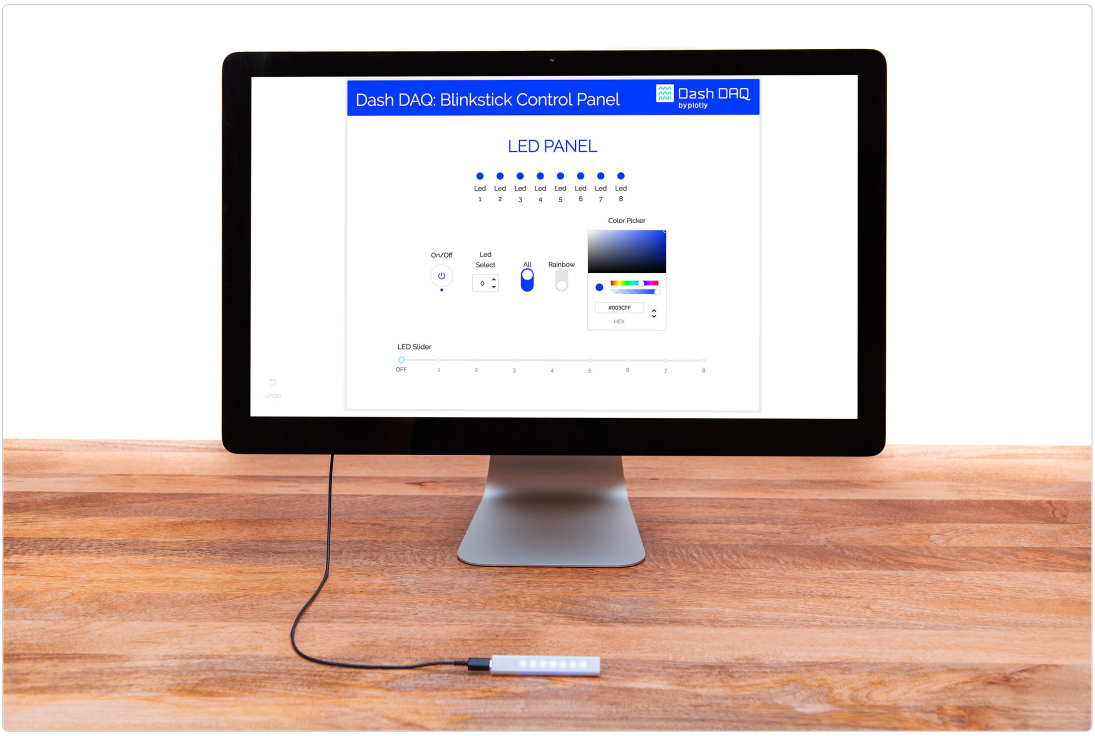
We wrote a Dash DAQ application in Python to control and read an Ocean Optics spectrometer with interactive UI components.

Kurt J. Lesker Pressure Gauge Controller | Try the app

A Dash DAQ application, written in Python, gives you clean, modern UI components to facilitate the readout of a Kurt J. Lesker pressure gauge controller.

Read Accelerometer Data | Try the app

Running tests with an accelerometer? Dash DAQ gives you the components you need to write rich, flexible GUIs for interfacing with your instruments in Python.



Control an LED Strip | Try the app

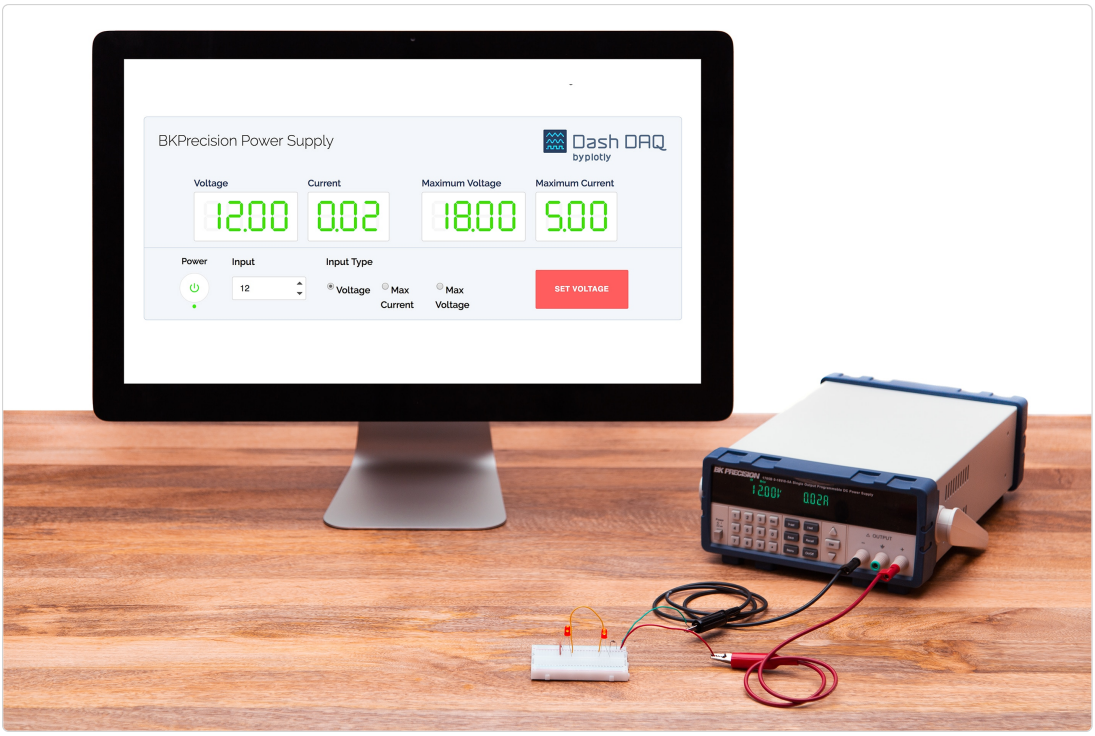
Team Plotly is getting colorful with Dash DAQ! This application controls the colored LED lights in a BlinkStick. We even wrote a Rainbow mode🌈

[Control a Stepper Motor](#) | [Try the app](#)

From 3D printers, to mirror mounts, to machine tools – stepper motors are ubiquitous. Using Dash DAQ, we created a GUI to control a Silverpak 17C Lin Engineering stepper motor.

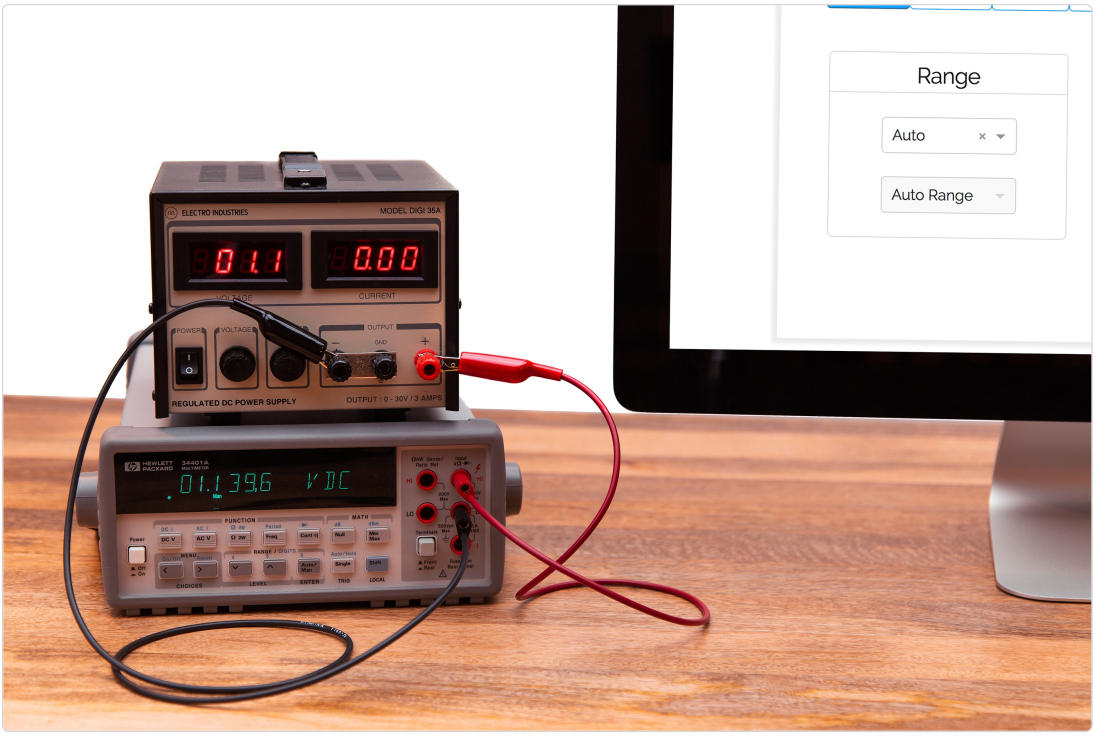
Tektronix Oscilloscope Data Logging | Try the app

Whether testing your power supply or monitoring a heartbeat, if you have an oscilloscope, Dash DAQ will help you control and read your instrument with user-friendly GUIs.



B&K Precision Power Supply | Try the app

This Dash DAQ app controls a B&K Precision power supply using a clean and functional UI, written in just over 300 lines of Python code.

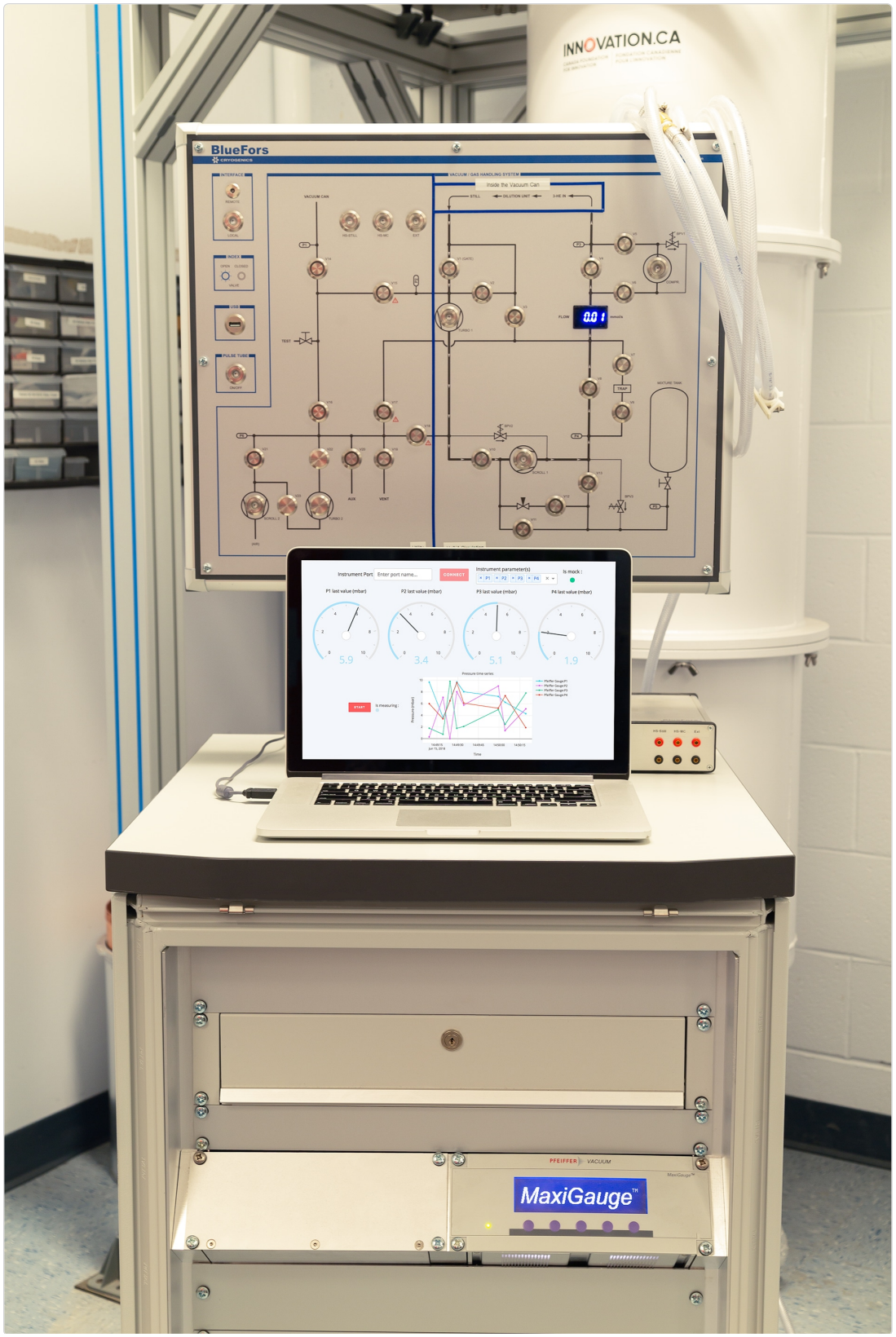


Agilent 34401A Multimeter | Try the app

Here's how we used Dash DAQ's interactive UI components to control the HP Agilent 34401A Multimeter.

Tektronix Function Generator | Try the app

A function generator generates a variety of electrical waveforms. This Dash DAQ application facilitates the control of a Tektronix AFG3021 function generator.


INNOVATION.CA
BlueFors
CRYOGENICS
VACUUM / GAS HANDLING SYSTEM
Inside the Vacuum Can
Instrument Port:
Enter port name...
CONNECT
Instrument parameter(s)
Is mock :
P1 last value (mbar)
P2 last value (mbar)
P3 last value (mbar)
P4 last value (mbar)
5.9
3.4
5.1
1.9
Pressure time series
START
Is measuring :
PFEIFFER VACUUM
MaxiGauge™

Pfeiffer Vacuum Gauge | Try the app

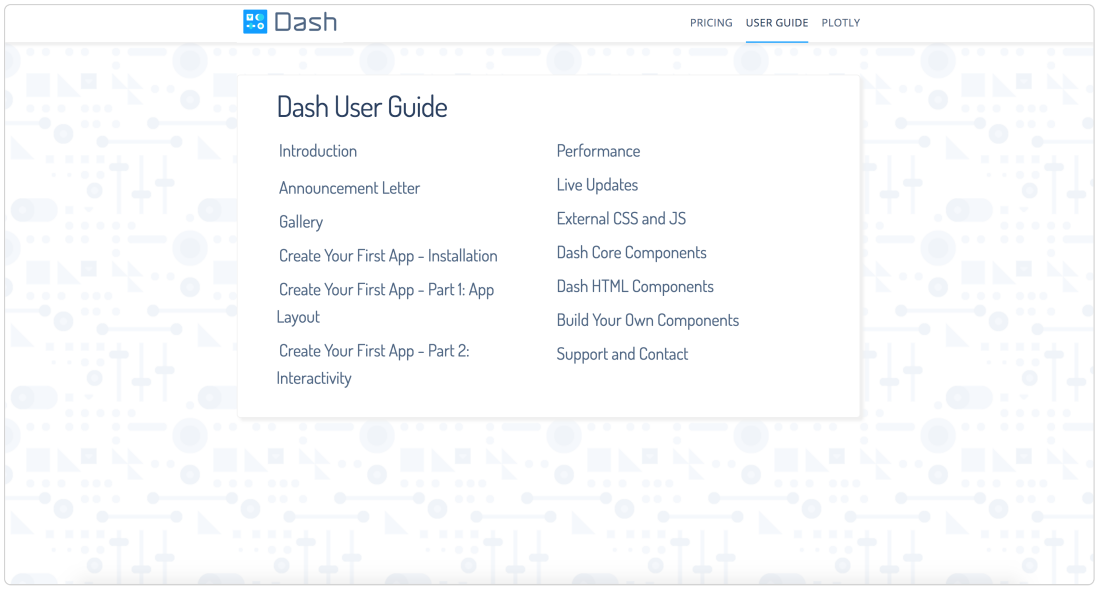
In just over 300 lines of code, this app helps you control and read a Pfeiffer vacuum gauge controller.

Analyze Frequency Responses using a Tektronix Function Generator and Oscilloscope | Try the app

With this Dash DAQ app, you can create a user-friendly GUI for analyzing the frequency responses of circuits.

Dash Documentation

These Dash docs that you're looking at? They themselves are a Dash app!



View Dash User Guide Source Code

Dash Tutorial

Dash Installation

In your terminal, install several dash libraries. These libraries are under active development, so install and upgrade frequently. Python 2 and 3 are supported.

```
pip install dash==0.35.1  # The core dash backend
pip install dash-html-components==0.13.2  # HTML components
pip install dash-core-components==0.42.1  # Supercharged components
pip install dash-table==3.1.11  # Interactive DataTable component (new!)
```

Ready? Now, let's make your first Dash app.

> A quick note on checking your versions and on upgrading. These docs are run using the versions listed above and these versions should be the latest versions available. To check which version that you have installed, you can run e.g.
>
> ```
> >>> import dash_core_components
> >>> print(dash_core_components.__version__)
> ```
>
> To see the latest changes of any package, check the GitHub repo's CHANGELOG.md file:
>
> - dash changelog
> - dash-core-components changelog
> - dash-html-components changelog
> - dash-table changelog
>
> Finally, note that the plotly package and the dash-renderer package are important package dependencies that are installed automatically with dash-core-components and dash respectively. These docs are using dash-renderer==0.15.0 and plotly==3.4.2 and their changelogs are located here:
>
> - dash-renderer changelog
> - plotly changelog
>
> All of these packages adhere to semver.

Dash Layout

> This is the 2nd chapter of the Dash Tutorial . The previous chapter covered installation and the next chapter covers Dash callbacks.

This tutorial will walk you through a fundamental aspect of Dash apps, the app `layout`, through 6 self-contained apps.

Dash apps are composed of two parts. The first part is the "`layout`" of the app and it describes what the application looks like. The second part describes the interactivity of the application and will be covered in the next chapter .

Dash provides Python classes for all of the visual components of the application. We maintain a set of components in the `dash_core_components` and the `dash_html_components` library but you can also build your own with JavaScript and React.js.

▼ To get started, create a file named `app.py` with the following code:

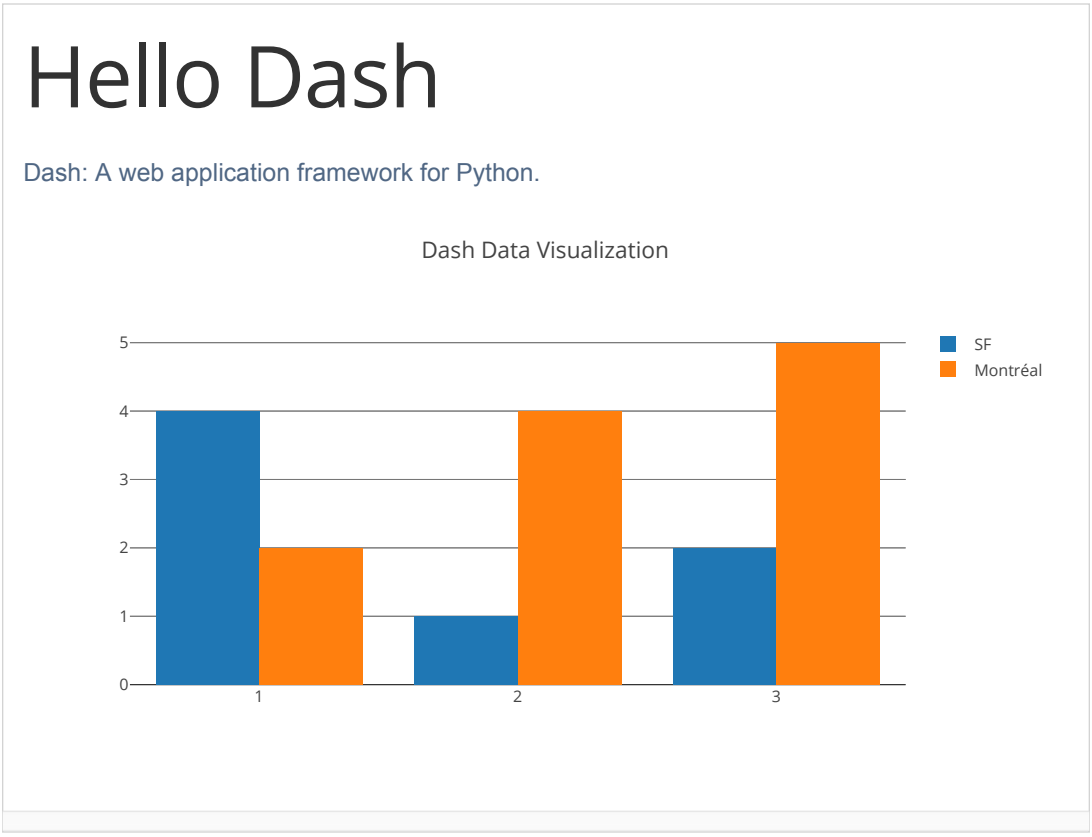
```
# -*- coding: utf-8 -*-
import dash
import dash_core_components as dcc
import dash_html_components as html

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div(children=[
    html.H1(children='Hello Dash'),

    html.Div(children='''
        Dash: A web application framework for Python.
    '''),

    dcc.Graph(
        id='example-graph',
        figure={
            'data': [
                {'x': [1, 2, 3], 'y': [4, 1, 2], 'type': 'bar', 'name':
                {'x': [1, 2, 3], 'y': [2, 4, 5], 'type': 'bar', 'name':
            ],
```

```
            'layout': {
                'title': 'Dash Data Visualization'
            }
        }
    )
])

if __name__ == '__main__':
    app.run_server(debug=True)
```

Run the app with

```
$ python app.py
...Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
```

and visit http:127.0.0.1:8050/ in your web browser. You should see an app that looks like this.



Note:

1. The `layout` is composed of a tree of "components" like `html.Div` and `dcc.Graph`.

2. The `dash_html_components` library has a component for every HTML tag. The `html.H1(children='Hello Dash')` component generates a `<h1>Hello Dash</h1>` HTML element in your application.

3. Not all components are pure HTML. The `dash_core_components` describe higher-level components that are interactive and are generated with JavaScript, HTML, and CSS through the React.js library.

4. Each component is described entirely through keyword attributes. Dash is declarative: you will primarily describe your application through these attributes.

5. The `children` property is special. By convention, it's always the first attribute which means that you can omit it: `html.H1(children='Hello Dash')` is the same as `html.H1('Hello Dash')`. Also, it can contain a string, a number, a single component, or a list of components.

6. The fonts in your application will look a little bit different than what is displayed here. This application is using a custom CSS stylesheet to modify the default styles of the elements. You can learn more in the css tutorial, but for now you can initialize your app with

```
external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

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

to get the same look and feel of these examples.

Making your first change

New in dash 0.30.0 and dash-renderer 0.15.0

Dash includes "hot-reloading", this features is activated by default when you run your app with `app.run_server(debug=True)`. This means that Dash will automatically refresh your browser when you make a change in your code.

Give it a try: change the title "Hello Dash" in your application or change the `x` or the `y` data. Your app should auto-refresh with your change.

> Don't like hot-reloading? You can turn this off with `app.run_server(dev_tools_hot_reload=False)`. Learn more in Dash Dev Tools documentation Questions? See the community forum hot reloading discussion .

More about HTML

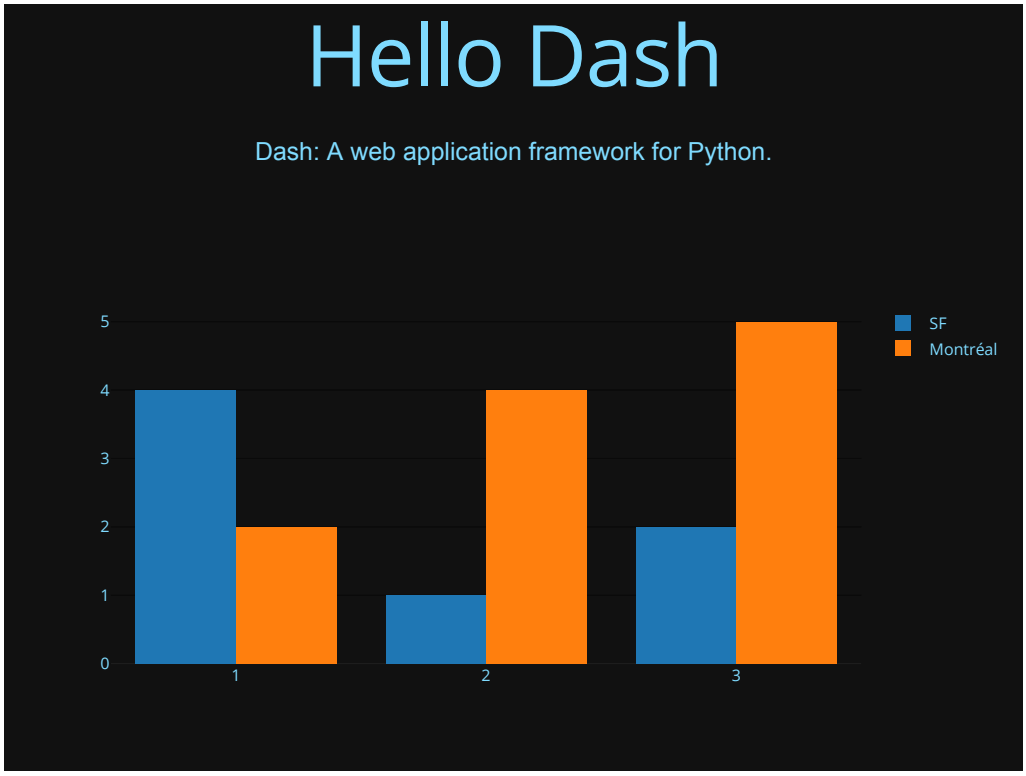
The `dash_html_components` library contains a component class for every HTML tag as well as keyword arguments for all of the HTML arguments.

▼ Let's customize the text in our app by modifying the inline styles of the components:

```
# -*- coding: utf-8 -*-
import dash
import dash_core_components as dcc
import dash_html_components as html
```

```
external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

colors = {
    'background': '#111111',
    'text': '#7FDBFF'
}

app.layout = html.Div(style={'backgroundColor': colors['background']}, c
    html.H1(
        children='Hello Dash',
        style={
            'textAlign': 'center',
            'color': colors['text']
        }
    ),

    html.Div(children='Dash: A web application framework for Python.', s
        'textAlign': 'center',
        'color': colors['text']
    }),

    dcc.Graph(
        id='example-graph-2',
        figure={
            'data': [
                {'x': [1, 2, 3], 'y': [4, 1, 2], 'type': 'bar', 'name':
                {'x': [1, 2, 3], 'y': [2, 4, 5], 'type': 'bar', 'name':
            ],
            'layout': {
                'plot_bgcolor': colors['background'],
                'paper_bgcolor': colors['background'],
                'font': {
                    'color': colors['text']
                }
            }
        }
    )
])
```

```
if __name__ == '__main__':
    app.run_server(debug=True)
```



In this example, we modified the inline styles of the `html.Div` and `html.H1` components with the `style` property.

`html.H1('Hello Dash', style={'textAlign': 'center', 'color': '#7FDBFF'})` is rendered in the Dash application as `<h1 style="text-align: center; color: #7FDBFF">Hello Dash</h1>`.

There are a few important differences between the `dash_html_components` and the HTML attributes:

1. The `style` property in HTML is a semicolon-separated string. In Dash, you can just supply a dictionary.

2. The keys in the `style` dictionary are camelCased. So, instead of `text-align`, it's `textAlign`.

3. The HTML `class` attribute is `className` in Dash.

4. The children of the HTML tag is specified through the `children` keyword argument. By convention, this is always the first argument and so it is often omitted.

Besides that, all of the available HTML attributes and tags are available to you within your Python context.

Reusable Components

By writing our markup in Python, we can create complex reusable components like tables without switching contexts or languages.

▼ Here's a quick example that generates a `Table` from a Pandas dataframe.

```
import dash
import dash_core_components as dcc
import dash_html_components as html
import pandas as pd

df = pd.read_csv(
    'https://gist.githubusercontent.com/chriddyp/'
    'c78bf172206ce24f77d6363a2d754b59/raw/'
    'c353e8ef842413cae56ae3920b8fd78468aa4cb2/'
    'usa-agricultural-exports-2011.csv')


def generate_table(dataframe, max_rows=10):
    return html.Table(
        # Header
        [html.Tr([html.Th(col) for col in dataframe.columns])] +

        # Body
        [html.Tr([
            html.Td(dataframe.iloc[i][col]) for col in dataframe.columns
        ]) for i in range(min(len(dataframe), max_rows))]
    )


external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div(children=[
    html.H4(children='US Agriculture Exports (2011)'),
    generate_table(df)
```

```
])

if __name__ == '__main__':
    app.run_server(debug=True)
```

US Agriculture Exports (2011)

Unnamed: 0	state	total exports	beef	pork	poultry	dairy	fruits fresh	fruits proc	total fruits
0	Alabama	1390.63	34.4	10.6	481	4.06	8	17.1	25.11
1	Alaska	13.31	0.2	0.1	0	0.19	0	0	0
2	Arizona	1463.17	71.3	17.9	0	105.48	19.3	41	60.27
3	Arkansas	3586.02	53.2	29.4	562.9	3.53	2.2	4.7	6.88
4	California	16472.88	228.7	11.1	225.4	929.95	2791.8	5944.6	8736.4
5	Colorado	1851.33	261.4	66	14	71.94	5.7	12.2	17.99
6	Connecticut	259.62	1.1	0.1	6.9	9.49	4.2	8.9	13.1
7	Delaware	282.19	0.4	0.6	114.7	2.3	0.5	1	1.53
8	Florida	3764.09	42.6	0.9	56.9	66.31	438.2	933.1	1371.36
9	Georgia	2860.84	31	18.9	630.4	38.38	74.6	158.9	233.51

More about Visualization

The `dash_core_components` library includes a component called `Graph`.

`Graph` renders interactive data visualizations using the open source plotly.js JavaScript graphing library. Plotly.js supports over 35 chart types and renders charts in both vector-quality SVG and high-performance WebGL.

The `figure` argument in the `dash_core_components.Graph` component is the same `figure` argument that is used by `plotly.py`, Plotly's open source Python graphing library. Check out the plotly.py documentation and gallery to learn more.

▼ Here's an example that creates a scatter plot from a Pandas dataframe.

```
import dash
import dash_core_components as dcc
import dash_html_components as html
import pandas as pd
import plotly.graph_objs as go

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

df = pd.read_csv(
    'https://gist.githubusercontent.com/chriddyp/' +
    '5d1ea79569ed194d432e56108a04d188/raw/' +
    'a9f9e8076b837d541398e999dcbac2b2826a81f8/'+
    'gdp-life-exp-2007.csv')


app.layout = html.Div([
    dcc.Graph(
        id='life-exp-vs-gdp',
        figure={
            'data': [
                go.Scatter(
                    x=df[df['continent'] == i]['gdp per capita'],
                    y=df[df['continent'] == i]['life expectancy'],
                    text=df[df['continent'] == i]['country'],
                    mode='markers',
                    opacity=0.7,
                    marker={
                        'size': 15,
                        'line': {'width': 0.5, 'color': 'white'}
                    },
                    name=i
                ) for i in df.continent.unique()
            ],
            'layout': go.Layout(
                xaxis={'type': 'log', 'title': 'GDP Per Capita'},
                yaxis={'title': 'Life Expectancy'},
                margin={'l': 40, 'b': 40, 't': 10, 'r': 10},
                legend={'x': 0, 'y': 1},
                hovermode='closest'
```

```
            )
        }
    )
])

if __name__ == '__main__':
    app.run_server(debug=True)
```

These graphs are interactive and responsive. Hover over points to see their values, click on legend items to toggle traces, click and drag to zoom, hold down shift, and click and drag to pan.

Markdown

While Dash exposes HTML through the `dash_html_components` library, it can be tedious to write your copy in HTML. For writing blocks of text, you can use the `Markdown` component in the `dash_core_components` library.

```
import dash
import dash_core_components as dcc
import dash_html_components as html

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

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

```
markdown_text = '''
### Dash and Markdown

Dash apps can be written in Markdown.
Dash uses the [CommonMark](http://commonmark.org/)
specification of Markdown.
Check out their [60 Second Markdown Tutorial](http://commonmark.org/help
if this is your first introduction to Markdown!
'''

app.layout = html.Div([
    dcc.Markdown(children=markdown_text)
])

if __name__ == '__main__':
    app.run_server(debug=True)
```

Dash and Markdown

Dash apps can be written in Markdown. Dash uses the CommonMark specification of Markdown. Check out their 60 Second Markdown Tutorial if this is your first introduction to Markdown!

Core Components

The `dash_core_components` includes a set of higher-level components like dropdowns, graphs, markdown blocks, and more.

Like all Dash components, they are described entirely declaratively. Every option that is configurable is available as a keyword argument of the component.

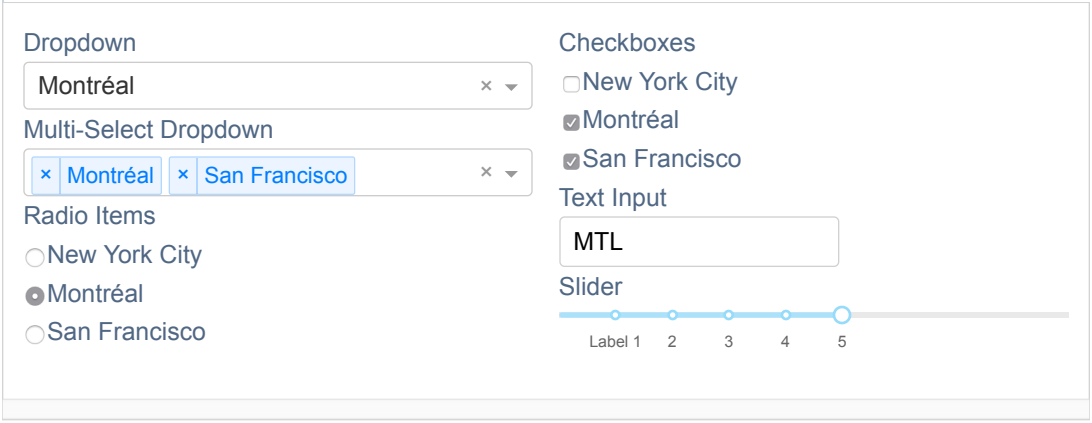
We'll see many of these components throughout the tutorial. You can view all of the available components in the Dash Core Components Gallery

▼ Here are a few of the available components:

```
# -*- coding: utf-8 -*-
import dash
import dash_core_components as dcc
import dash_html_components as html
```

```
external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div([
    html.Label('Dropdown'),
    dcc.Dropdown(
        options=[
            {'label': 'New York City', 'value': 'NYC'},
            {'label': u'Montréal', 'value': 'MTL'},
            {'label': 'San Francisco', 'value': 'SF'}
        ],
        value='MTL'
    ),

    html.Label('Multi-Select Dropdown'),
    dcc.Dropdown(
        options=[
            {'label': 'New York City', 'value': 'NYC'},
            {'label': u'Montréal', 'value': 'MTL'},
            {'label': 'San Francisco', 'value': 'SF'}
        ],
        value=['MTL', 'SF'],
        multi=True
    ),

    html.Label('Radio Items'),
    dcc.RadioItems(
        options=[
            {'label': 'New York City', 'value': 'NYC'},
            {'label': u'Montréal', 'value': 'MTL'},
            {'label': 'San Francisco', 'value': 'SF'}
        ],
        value='MTL'
    ),

    html.Label('Checkboxes'),
    dcc.Checklist(
        options=[
            {'label': 'New York City', 'value': 'NYC'},
            {'label': u'Montréal', 'value': 'MTL'},
```

```
            {'label': 'San Francisco', 'value': 'SF'}
        ],
        values=['MTL', 'SF']
    ),

    html.Label('Text Input'),
    dcc.Input(value='MTL', type='text'),

    html.Label('Slider'),
    dcc.Slider(
        min=0,
        max=9,
        marks={i: 'Label {}'.format(i) if i == 1 else str(i) for i in ra
        value=5,
    ),
], style={'columnCount': 2})

if __name__ == '__main__':
    app.run_server(debug=True)
```



Calling `help`

Dash components are declarative: every configurable aspect of these components is set during instantiation as a keyword argument. Call `help` in your Python console on any of the components to learn more about a component and its available arguments.

```
>>> help(dcc.Dropdown)
class Dropdown(dash.development.base_component.Component)
|  A Dropdown component.
|  Dropdown is an interactive dropdown element for selecting one or more
|  items.
```

```
| The values and labels of the dropdown items are specified in the `option
| property and the selected item(s) are specified with the `value` propert
|
| Use a dropdown when you have many options (more than 5) or when you are
| constrained for space. Otherwise, you can use RadioItems or a Checklist,
| which have the benefit of showing the users all of the items at once.
|
| Keyword arguments:
| - id (string; optional)
| - className (string; optional)
| - disabled (boolean; optional): If true, the option is disabled
| - multi (boolean; optional): If true, the user can select multiple value
| - options (list; optional)
| - placeholder (string; optional): The grey, default text shown when no o
| - value (string | list; optional): The value of the input. If `multi` is
| then value is just a string that corresponds to the values
| provided in the `options` property. If `multi` is true, then
| multiple values can be selected at once, and `value` is an
| array of items with values corresponding to those in the
| `options` prop.```
```

Summary

The `layout` of a Dash app describes what the app looks like. The `layout` is a hierarchical tree of components. The `dash_html_components` library provides classes for all of the HTML tags and the keyword arguments describe the HTML attributes like `style`, `className`, and `id`. The `dash_core_components` library generates higher-level components like controls and graphs.

For reference, see:

- `dash_core_components` gallery
- `dash_html_components` gallery

The next part of the Dash tutorial covers how to make these apps interactive.

Dash Tutorial Part 3: Basic Callbacks

Basic Dash Callbacks

> This is the 3rd chapter of the Dash Tutorial. The previous chapter covered the Dash app `layout` and the next chapter covers an additional concept of callbacks known as `state`. Just getting started? Make sure to install the necessary dependencies.

In the previous chapter on the `app.layout` we learned that the `app.layout` describes what the app looks like and is a hierarchical tree of components. The `dash_html_components` library provides classes for all of the HTML tags, and the keyword arguments describe the HTML attributes like `style`, `className`, and `id`. The `dash_core_components` library generates higher-level components like controls and graphs.

This chapter describes how to make your Dash apps interactive.

Let's get started with a simple example.

Dash App Layout

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

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div([
    dcc.Input(id='my-id', value='initial value', type='text'),
    html.Div(id='my-div')
])


@app.callback(
    Output(component_id='my-div', component_property='children'),
    [Input(component_id='my-id', component_property='value')]
)
def update_output_div(input_value):
    return 'You\'ve entered "{}"'.format(input_value)
```

```
if __name__ == '__main__':
    app.run_server(debug=True)
```

initial value

You've entered "initial value"

Try typing in the text box. The children of the output component updates right away. Let's break down what's happening here:

1. The "inputs" and "outputs" of our application interface are described declaratively through the `app.callback` decorator.

2. In Dash, the inputs and outputs of our application are simply the properties of a particular component. In this example, our input is the "`value`" property of the component that has the ID "`my-id`". Our output is the "`children`" property of the component with the ID "`my-div`".

3. 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.

4. The `component_id` and `component_property` keywords are optional (there are only two arguments for each of those objects). I have included them here for clarity but I will omit them from here on out for brevity and readability.

5. Don't confuse the `dash.dependencies.Input` object from the `dash_core_components.Input` object. The former is just used in these callbacks and the latter is an actual component.

6. Notice how we don't set a value for the `children` property of the `my-div` component in the `layout`. When the Dash app starts, it automatically calls all of the callbacks with the initial values of the input components in order to populate the initial state of the output components. In this example, if you specified something like `html.Div(id='my-div', children='Hello world')`, it would get overwritten when the app starts.

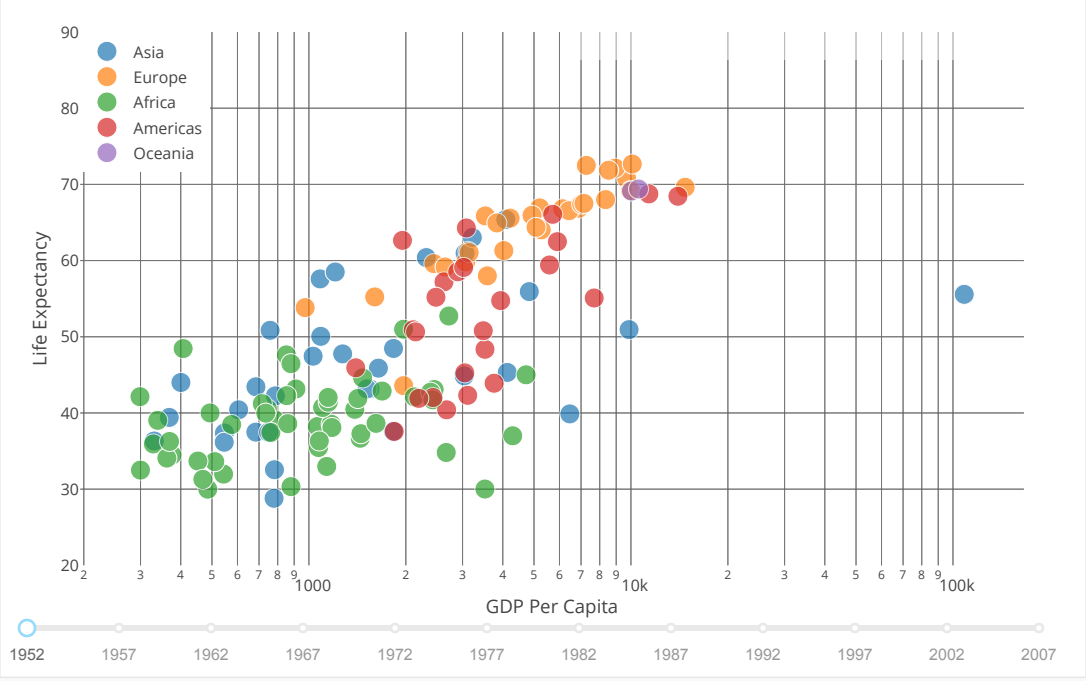
It's sort of like programming with Microsoft Excel: whenever an input cell changes, all of the cells that depend on that cell will get updated automatically. This is called "Reactive Programming".

Remember how every component was described entirely through its set of keyword arguments? Those properties are important now. With Dash interactivity, we can dynamically update any of those properties through a callback function. Frequently we'll update the `children` of a component to display new text or the `figure` of a `dcc.Graph` component to display new data, but we could also update the `style` of a component or even the available `options` of a `dcc.Dropdown` component!

Let's take a look at another example where a `dcc.Slider` updates a `dcc.Graph`.

```
import dash
import dash_core_components as dcc
import dash_html_components as html
import pandas as pd
import plotly.graph_objs as go

df = pd.read_csv(
    'https://raw.githubusercontent.com/plotly/'
    'datasets/master/gapminderDataFiveYear.csv')

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div([
    dcc.Graph(id='graph-with-slider'),
    dcc.Slider(
        id='year-slider',
        min=df['year'].min(),
        max=df['year'].max(),
        value=df['year'].min(),
        marks={str(year): str(year) for year in df['year'].unique()}
    )
])


@app.callback(
    dash.dependencies.Output('graph-with-slider', 'figure'),
    [dash.dependencies.Input('year-slider', 'value')])
def update_figure(selected_year):
    filtered_df = df[df.year == selected_year]
    traces = []
    for i in filtered_df.continent.unique():
        df_by_continent = filtered_df[filtered_df['continent'] == i]
        traces.append(go.Scatter(
            x=df_by_continent['gdpPercap'],
            y=df_by_continent['lifeExp'],
            text=df_by_continent['country'],
            mode='markers',
            opacity=0.7,
            marker={
```

```
                'size': 15,
                'line': {'width': 0.5, 'color': 'white'}
            },
            name=i
        ))

    return {
        'data': traces,
        'layout': go.Layout(
            xaxis={'type': 'log', 'title': 'GDP Per Capita'},
            yaxis={'title': 'Life Expectancy', 'range': [20, 90]},
            margin={'l': 40, 'b': 40, 't': 10, 'r': 10},
            legend={'x': 0, 'y': 1},
            hovermode='closest'
        )
    }


if __name__ == '__main__':
    app.run_server(debug=True)
```



In this example, the `"value"` property of the `Slider` is the input of the app and the output of the app is the `"figure"` property of the `Graph`. Whenever the `value` of the `Slider` changes,

Dash calls the callback function `update_figure` with the new value. The function filters the dataframe with this new value, constructs a `figure` object, and returns it to the Dash application.

There are a few nice patterns in this example:

1. We're using the `Pandas` library for importing and filtering datasets in memory.

2. We load our dataframe at the start of the app: `df = pd.read_csv('...')`. This dataframe `df` is in the global state of the app and can be read inside the callback functions.

3. Loading data into memory can be expensive. By loading querying data at the start of the app instead of inside the callback functions, we ensure that this operation is only done when the app server starts. When a user visits the app or interacts with the app, that data (the `df`) is already in memory. If possible, expensive initialization (like downloading or querying data) should be done in the global scope of the app instead of within the callback functions.

4. The callback does not modify the original data, it just creates copies of the dataframe by filtered through pandas filters. This is important: your callbacks should never mutate variables outside of their scope. If your callbacks modify global state, then one user's session might affect the next user's session and when the app is deployed on multiple processes or threads, those modifications will not be shared across sessions.

Multiple inputs

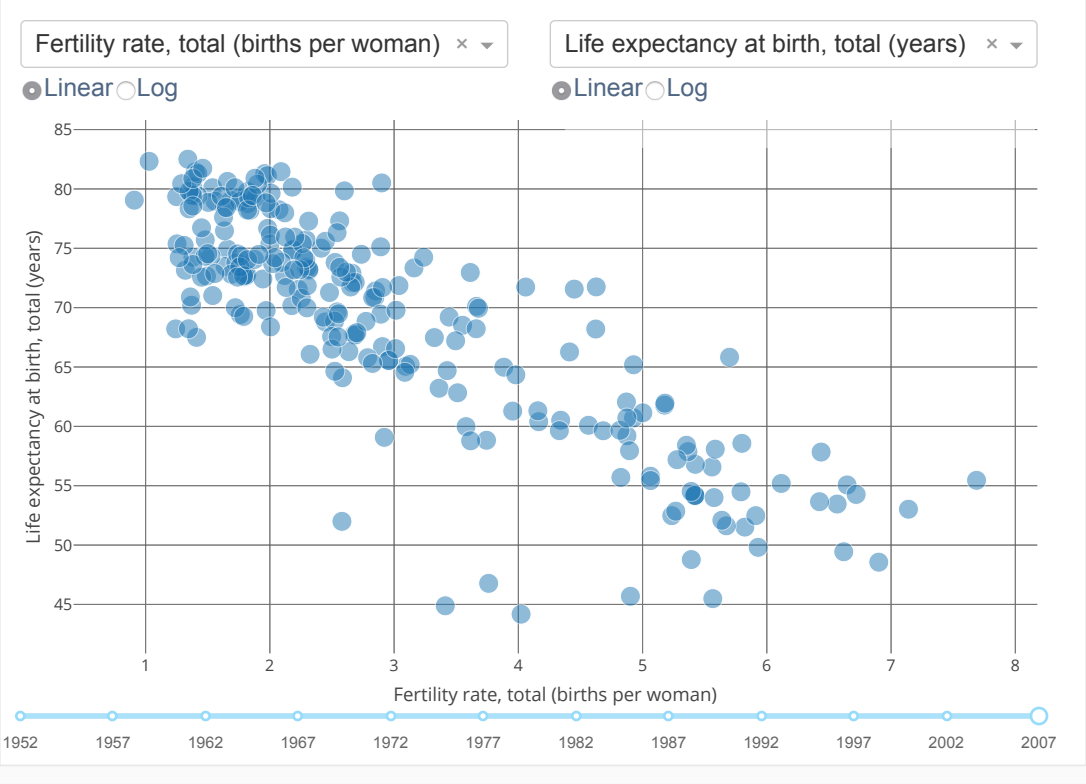
In Dash, any "`Output`" can have multiple "`Input`" components. Here's a simple example that binds five Inputs (the `value` property of 2 `Dropdown` components, 2 `RadioItems` components, and 1 `Slider` component) to 1 Output component (the `figure` property of the `Graph` component). Notice how the `app.callback` lists all five `dash.dependencies.Input` inside a list in the second argument.

```
import dash
import dash_core_components as dcc
import dash_html_components as html
import pandas as pd
import plotly.graph_objs as go

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

df = pd.read_csv(
    'https://gist.githubusercontent.com/chriddyp/'
    'cb5392c35661370d95f300086accea51/raw/'
    '8e0768211f6b747c0db42a9ce9a0937dafcbd8b2/'
    'indicators.csv')
```

```
available_indicators = df['Indicator Name'].unique()

app.layout = html.Div([
    html.Div([

        html.Div([
            dcc.Dropdown(
                id='xaxis-column',
                options=[{'label': i, 'value': i} for i in available_ind
                value='Fertility rate, total (births per woman)'
            ),
            dcc.RadioItems(
                id='xaxis-type',
                options=[{'label': i, 'value': i} for i in ['Linear', 'L
                value='Linear',
                labelStyle={'display': 'inline-block'}
            )
        ],
        style={'width': '48%', 'display': 'inline-block'}),

        html.Div([
            dcc.Dropdown(
                id='yaxis-column',
                options=[{'label': i, 'value': i} for i in available_ind
                value='Life expectancy at birth, total (years)'
            ),
            dcc.RadioItems(
                id='yaxis-type',
                options=[{'label': i, 'value': i} for i in ['Linear', 'L
                value='Linear',
                labelStyle={'display': 'inline-block'}
            )
        ],style={'width': '48%', 'float': 'right', 'display': 'inline-bl
    ]),

    dcc.Graph(id='indicator-graphic'),

    dcc.Slider(
        id='year--slider',
        min=df['Year'].min(),
        max=df['Year'].max(),
```

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

@app.callback(
    dash.dependencies.Output('indicator-graphic', 'figure'),
    [dash.dependencies.Input('xaxis-column', 'value'),
     dash.dependencies.Input('yaxis-column', 'value'),
     dash.dependencies.Input('xaxis-type', 'value'),
     dash.dependencies.Input('yaxis-type', 'value'),
     dash.dependencies.Input('year--slider', 'value')])
def update_graph(xaxis_column_name, yaxis_column_name,
                 xaxis_type, yaxis_type,
                 year_value):
    dff = df[df['Year'] == year_value]

    return {
        'data': [go.Scatter(
            x=dff[dff['Indicator Name'] == xaxis_column_name]['Value'],
            y=dff[dff['Indicator Name'] == yaxis_column_name]['Value'],
            text=dff[dff['Indicator Name'] == yaxis_column_name]['Countr
            mode='markers',
            marker={
                'size': 15,
                'opacity': 0.5,
                'line': {'width': 0.5, 'color': 'white'}
            }
        )],
        'layout': go.Layout(
            xaxis={
                'title': xaxis_column_name,
                'type': 'linear' if xaxis_type == 'Linear' else 'log'
            },
            yaxis={
                'title': yaxis_column_name,
                'type': 'linear' if yaxis_type == 'Linear' else 'log'
            },
            margin={'l': 40, 'b': 40, 't': 10, 'r': 0},
            hovermode='closest'
        )
    }
```

```
if __name__ == '__main__':
    app.run_server(debug=True)
```



In this example, the `update_graph` function gets called whenever the `value` property of the `Dropdown`, `Slider`, or `RadioItems` components change.

The input arguments of the `update_graph` function are the new or current value of each of the `Input` properties, in the order that they were specified.

Even though only a single `Input` changes at a time (a user can only change the value of a single Dropdown in a given moment), Dash collects the current state of all of the specified `Input` properties and passes them into your function for you. Your callback functions are always guaranteed to be passed the representative state of the app.

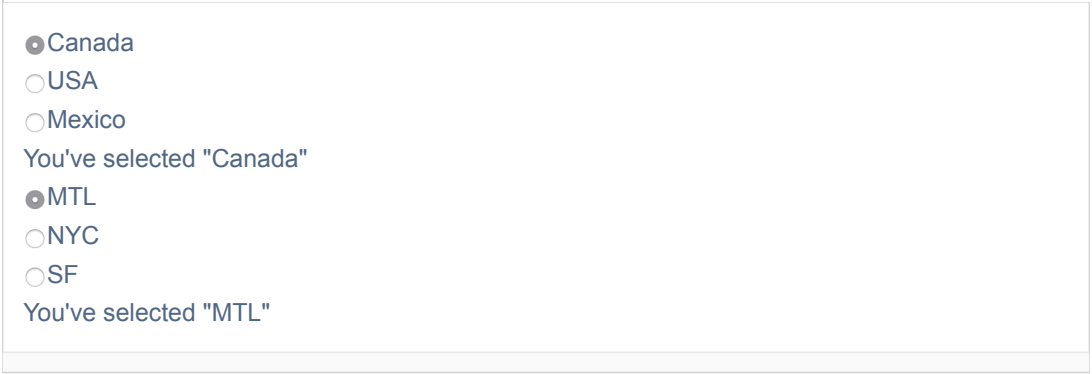
Let's extend our example to include multiple outputs.

Multiple Outputs

Each Dash callback function can only update a single Output property. To update multiple Outputs, just write multiple functions.

```
import dash
import dash_core_components as dcc
import dash_html_components as html

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div([
    dcc.RadioItems(
        id='dropdown-a',
        options=[{'label': i, 'value': i} for i in ['Canada', 'USA', 'Me
        value='Canada'
    ),
    html.Div(id='output-a'),

    dcc.RadioItems(
        id='dropdown-b',
        options=[{'label': i, 'value': i} for i in ['MTL', 'NYC', 'SF']]
        value='MTL'
    ),
    html.Div(id='output-b')

])


@app.callback(
    dash.dependencies.Output('output-a', 'children'),
    [dash.dependencies.Input('dropdown-a', 'value')])
def callback_a(dropdown_value):
    return 'You\'ve selected "{}"'.format(dropdown_value)


@app.callback(
    dash.dependencies.Output('output-b', 'children'),
    [dash.dependencies.Input('dropdown-b', 'value')])
def callback_b(dropdown_value):
    return 'You\'ve selected "{}"'.format(dropdown_value)
```

```
if __name__ == '__main__':
    app.run_server(debug=True)
```

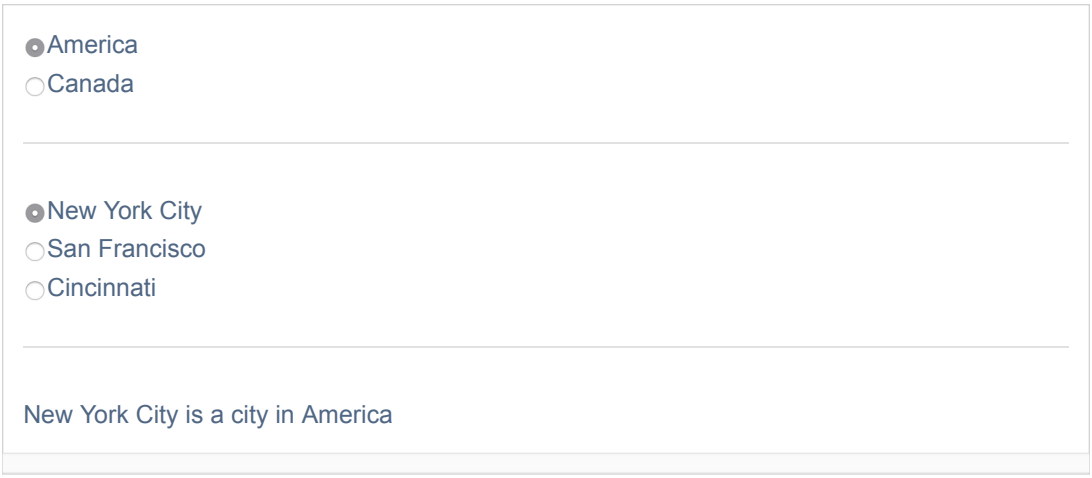


You can also chain outputs and inputs together: the output of one callback function could be the input of another callback function.

This pattern can be used to create dynamic UIs where one input component updates the available options of the next input component. Here's a simple example.

```
# -*- coding: utf-8 -*-
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

all_options = {
    'America': ['New York City', 'San Francisco', 'Cincinnati'],
    'Canada': [u'Montréal', 'Toronto', 'Ottawa']
}
app.layout = html.Div([
    dcc.RadioItems(
        id='countries-dropdown',
        options=[{'label': k, 'value': k} for k in all_options.keys()],
        value='America'
    ),

    html.Hr(),

    dcc.RadioItems(id='cities-dropdown'),
```

```
    html.Hr(),

    html.Div(id='display-selected-values')
])


@app.callback(
    dash.dependencies.Output('cities-dropdown', 'options'),
    [dash.dependencies.Input('countries-dropdown', 'value')])
def set_cities_options(selected_country):
    return [{'label': i, 'value': i} for i in all_options[selected_count


@app.callback(
    dash.dependencies.Output('cities-dropdown', 'value'),
    [dash.dependencies.Input('cities-dropdown', 'options')])
def set_cities_value(available_options):
    return available_options[0]['value']


@app.callback(
    dash.dependencies.Output('display-selected-values', 'children'),
    [dash.dependencies.Input('countries-dropdown', 'value'),
     dash.dependencies.Input('cities-dropdown', 'value')])
def set_display_children(selected_country, selected_city):
    return u'{} is a city in {}'.format(
        selected_city, selected_country,
    )


if __name__ == '__main__':
    app.run_server(debug=True)
```



The first callback updates the available options in the second `RadioItems` component based off of the selected value in the first `RadioItems` component.

The second callback sets an initial value when the `options` property changes: it sets it to the first value in that `options` array.

The final callback displays the selected `value` of each component. If you change the `value` of the countries `RadioItems` component, Dash will wait until the `value` of the cities component is updated before calling the final callback. This prevents your callbacks from being called with inconsistent state like with `"USA"` and `"Montréal"`.

Summary

We've covered the fundamentals of callbacks in Dash. Dash apps are built off of a set of simple but powerful principles: declarative UIs that are customizable through reactive and functional Python callbacks. Every element attribute of the declarative components can be updated through a callback and a subset of the attributes, like the `value` properties of the `dcc.Dropdown`, are editable by the user in the interface.

The next part of the Dash tutorial covers an additional concept of Dash callbacks: `State`

Dash Tutorial Part 4: State

Dash State

> This is the 4th chapter of the Dash Tutorial. The previous chapter covered Dash Callbacks and the next chapter covers interactive graphing and crossfiltering. Just getting started? Make sure to install the necessary dependencies.

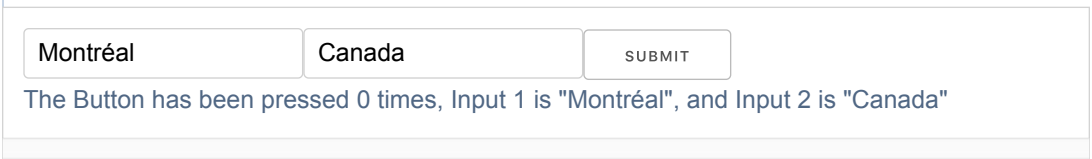
In the previous chapter on basic Dash callbacks, our callbacks looked something like:

```
# -*- coding: utf-8 -*-
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div([
    dcc.Input(id='input-1', type='text', value='Montréal'),
    dcc.Input(id='input-2', type='text', value='Canada'),
    html.Div(id='output')
])


@app.callback(Output('output', 'children'),
              [Input('input-1', 'value'),
               Input('input-2', 'value')])
def update_output(input1, input2):
    return u'Input 1 is "{}" and Input 2 is "{}"'.format(input1, input2)


if __name__ == '__main__':
    app.run_server(debug=True)
```

In this example, the callback function is fired whenever any of the attributes described by the `dash.dependencies.Input` change. Try it for yourself by entering data in the inputs above.

`dash.dependencies.State` allows you to pass along extra values without firing the callbacks. Here's the same example as above but with the `dcc.Input` as `dash.dependencies.State` and a button as `dash.dependencies.Input`.

```
# -*- coding: utf-8 -*-
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output, State

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div([
    dcc.Input(id='input-1-state', type='text', value='Montréal'),
    dcc.Input(id='input-2-state', type='text', value='Canada'),
    html.Button(id='submit-button', n_clicks=0, children='Submit'),
    html.Div(id='output-state')
])


@app.callback(Output('output-state', 'children'),
              [Input('submit-button', 'n_clicks')],
              [State('input-1-state', 'value'),
               State('input-2-state', 'value')])
def update_output(n_clicks, input1, input2):
    return u'''
        The Button has been pressed {} times,
        Input 1 is "{}",
        and Input 2 is "{}"
    '''.format(n_clicks, input1, input2)


if __name__ == '__main__':
    app.run_server(debug=True)
```



In this example, changing text in the `dcc.Input` boxes won't fire the callback but clicking on the button will. The current values of the `dcc.Input` values are still passed into the callback even though they don't trigger the callback function itself.

Note that we're triggering the callback by listening to the `n_clicks` property of the `html.Button` component. `n_clicks` is a property that gets incremented every time the component has been clicked on. It is available in every component in the `dash_html_components` library.

The next chapter of the user guide explains how to use callback principles with the `dash_core_components.Graph` component to make applications that respond to interactions with graphs on the page.

Dash Tutorial Part 5. Interactive Graphing

Interactive Visualizations

> This is the 5th chapter of the Dash Tutorial. The previous chapter covered callbacks with `State` and the next chapter describes how to share data between callbacks. Just getting started? Make sure to install the necessary dependencies.

The `dash_core_components` library includes a component called `Graph`.

`Graph` renders interactive data visualizations using the open source plotly.js JavaScript graphing library. Plotly.js supports over 35 chart types and renders charts in both vector-quality SVG and high-performance WebGL.

The `figure` argument in the `dash_core_components.Graph` component is the same `figure` argument that is used by `plotly.py`, Plotly's open source Python graphing library. Check out the plotly.py documentation and gallery to learn more.

Dash components are described declaratively by a set of attributes. All of these attributes can be updated by callback functions, but only a subset of these attributes are updated through user interaction, such as
when you click on an option in a `dcc.Dropdown` component and the `value` property of that component changes.

The `dcc.Graph` component has four attributes that can change through user-interaction: `hoverData`, `clickData`, `selectedData`, `relayoutData`. These properties update when you hover over points, click on points, or select regions of points in a graph.

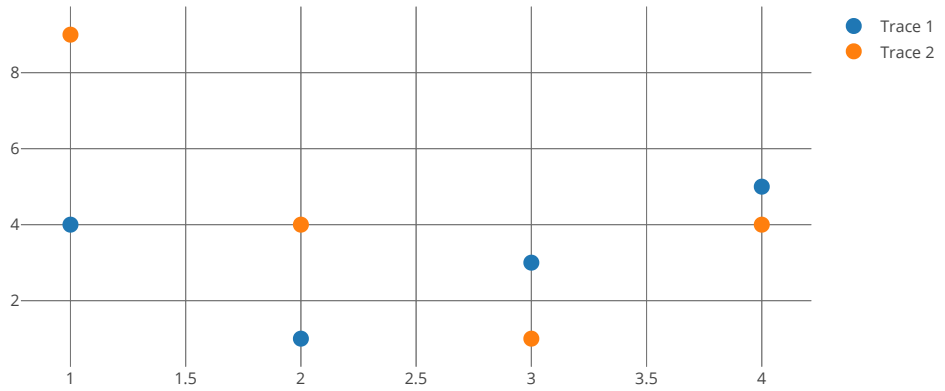
▼ Here's an simple example that prints these attributes in the screen.

```
import json
from textwrap import dedent as d

import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

styles = {
    'pre': {
        'border': 'thin lightgrey solid',
        'overflowX': 'scroll'
    }
```

```
}

app.layout = html.Div([
    dcc.Graph(
        id='basic-interactions',
        figure={
            'data': [
                {
                    'x': [1, 2, 3, 4],
                    'y': [4, 1, 3, 5],
                    'text': ['a', 'b', 'c', 'd'],
                    'customdata': ['c.a', 'c.b', 'c.c', 'c.d'],
                    'name': 'Trace 1',
                    'mode': 'markers',
                    'marker': {'size': 12}
                },
                {
                    'x': [1, 2, 3, 4],
                    'y': [9, 4, 1, 4],
                    'text': ['w', 'x', 'y', 'z'],
                    'customdata': ['c.w', 'c.x', 'c.y', 'c.z'],
                    'name': 'Trace 2',
                    'mode': 'markers',
                    'marker': {'size': 12}
                }
            ],
            'layout': {
                'clickmode': 'event+select'
            }
        }
    ),

    html.Div(className='row', children=[
        html.Div([
            dcc.Markdown(d("""
                **Hover Data**

                Mouse over values in the graph.
            """)),
            html.Pre(id='hover-data', style=styles['pre'])
        ], className='three columns'),
```

```
        html.Div([
            dcc.Markdown(d("""
                **Click Data**

                Click on points in the graph.
            """)),
            html.Pre(id='click-data', style=styles['pre']),
        ], className='three columns'),

        html.Div([
            dcc.Markdown(d("""
                **Selection Data**

                Choose the lasso or rectangle tool in the graph's menu
                bar and then select points in the graph.

                Note that if `layout.clickmode = 'event+select'`, select
                accumulates (or un-accumulates) selected data if you hol
                button while clicking.
            """)),
            html.Pre(id='selected-data', style=styles['pre']),
        ], className='three columns'),

        html.Div([
            dcc.Markdown(d("""
                **Zoom and Relayout Data**

                Click and drag on the graph to zoom or click on the zoom
                buttons in the graph's menu bar.
                Clicking on legend items will also fire
                this event.
            """)),
            html.Pre(id='relayout-data', style=styles['pre']),
        ], className='three columns')
    ])
])


@app.callback(
    Output('hover-data', 'children'),
    [Input('basic-interactions', 'hoverData')])
def display_hover_data(hoverData):
```

```
    return json.dumps(hoverData, indent=2)


@app.callback(
    Output('click-data', 'children'),
    [Input('basic-interactions', 'clickData')])
def display_click_data(clickData):
    return json.dumps(clickData, indent=2)


@app.callback(
    Output('selected-data', 'children'),
    [Input('basic-interactions', 'selectedData')])
def display_selected_data(selectedData):
    return json.dumps(selectedData, indent=2)


@app.callback(
    Output('relayout-data', 'children'),
    [Input('basic-interactions', 'relayoutData')])
def display_selected_data(relayoutData):
    return json.dumps(relayoutData, indent=2)


if __name__ == '__main__':
    app.run_server(debug=True)
```



Hover Data

Mouse over values in the graph.

```
null
```

Click Data

Click on points in the graph.

```
null
```

Selection Data

Choose the lasso or rectangle tool in the graph's menu bar and then select points in the graph.

Note that if `layout.clickmode = 'event+select'`, selection data also accumulates (or un-accumulates) selected data if you hold down the shift button while clicking.

```
null
```

Zoom and Relayout Data

Click and drag on the graph to zoom or click on the zoom buttons in the graph's menu bar. Clicking on legend items will also fire this event.

```
{
  "autosize": true
}
```

Update Graphs on Hover

▼ Let's update our world indicators example from the previous chapter by updating time series when we hover over points in our scatter plot.

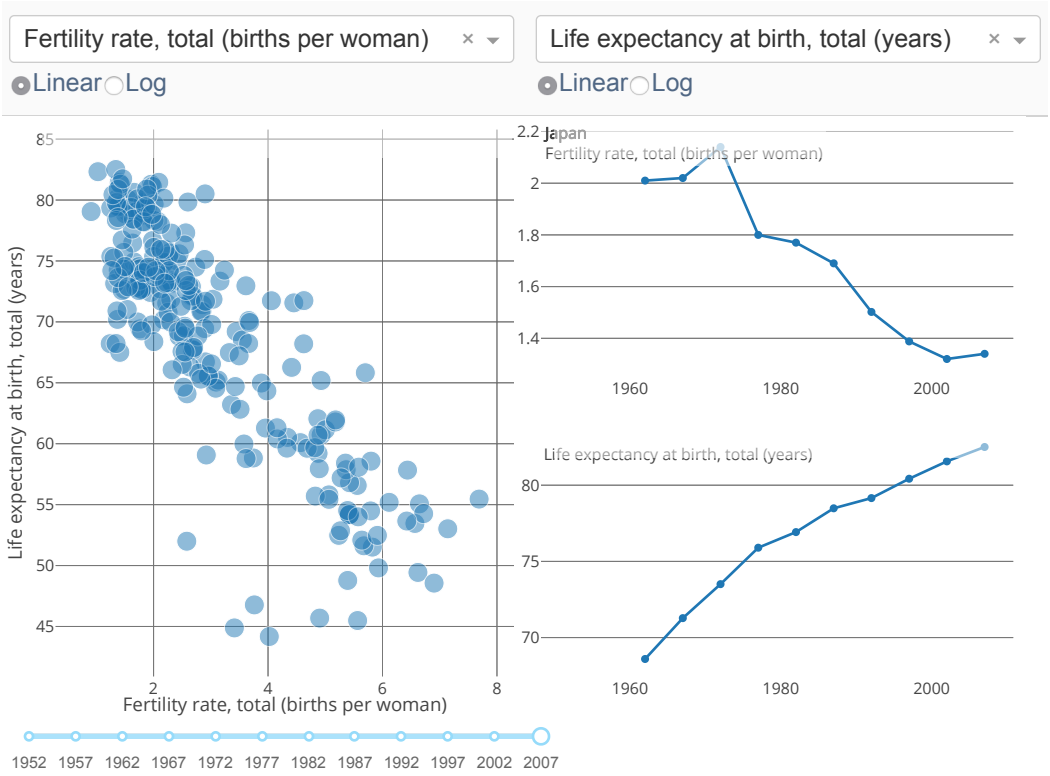
```
import dash
import dash_core_components as dcc
import dash_html_components as html
import pandas as pd
import plotly.graph_objs as go

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

df = pd.read_csv(
    'https://gist.githubusercontent.com/chriddyp/'
    'cb5392c35661370d95f300086accea51/raw/'
    '8e0768211f6b747c0db42a9ce9a0937dafcbd8b2/'
    'indicators.csv')

available_indicators = df['Indicator Name'].unique()

app.layout = html.Div([
    html.Div([

        html.Div([
            dcc.Dropdown(
                id='crossfilter-xaxis-column',
                options=[{'label': i, 'value': i} for i in available_ind
                value='Fertility rate, total (births per woman)'
            ),
            dcc.RadioItems(
                id='crossfilter-xaxis-type',
                options=[{'label': i, 'value': i} for i in ['Linear', 'L
                value='Linear',
                labelStyle={'display': 'inline-block'}
            )
        ],
        style={'width': '49%', 'display': 'inline-block'}),

        html.Div([
```

```
            dcc.Dropdown(
                id='crossfilter-yaxis-column',
                options=[{'label': i, 'value': i} for i in available_ind
                value='Life expectancy at birth, total (years)'
            ),
            dcc.RadioItems(
                id='crossfilter-yaxis-type',
                options=[{'label': i, 'value': i} for i in ['Linear', 'L
                value='Linear',
                labelStyle={'display': 'inline-block'}
            )
        ], style={'width': '49%', 'float': 'right', 'display': 'inline-b
    ], style={
        'borderBottom': 'thin lightgrey solid',
        'backgroundColor': 'rgb(250, 250, 250)',
        'padding': '10px 5px'
    }),

    html.Div([
        dcc.Graph(
            id='crossfilter-indicator-scatter',
            hoverData={'points': [{'customdata': 'Japan'}]}
        )
    ], style={'width': '49%', 'display': 'inline-block', 'padding': '0 2
    html.Div([
        dcc.Graph(id='x-time-series'),
        dcc.Graph(id='y-time-series'),
    ], style={'display': 'inline-block', 'width': '49%'}),

    html.Div(dcc.Slider(
        id='crossfilter-year--slider',
        min=df['Year'].min(),
        max=df['Year'].max(),
        value=df['Year'].max(),
        marks={str(year): str(year) for year in df['Year'].unique()}
    ), style={'width': '49%', 'padding': '0px 20px 20px 20px'})
])


@app.callback(
    dash.dependencies.Output('crossfilter-indicator-scatter', 'figure'),
    [dash.dependencies.Input('crossfilter-xaxis-column', 'value'),
```

```
     dash.dependencies.Input('crossfilter-yaxis-column', 'value'),
     dash.dependencies.Input('crossfilter-xaxis-type', 'value'),
     dash.dependencies.Input('crossfilter-yaxis-type', 'value'),
     dash.dependencies.Input('crossfilter-year--slider', 'value')])
def update_graph(xaxis_column_name, yaxis_column_name,
                 xaxis_type, yaxis_type,
                 year_value):
    dff = df[df['Year'] == year_value]

    return {
        'data': [go.Scatter(
            x=dff[dff['Indicator Name'] == xaxis_column_name]['Value'],
            y=dff[dff['Indicator Name'] == yaxis_column_name]['Value'],
            text=dff[dff['Indicator Name'] == yaxis_column_name]['Countr
            customdata=dff[dff['Indicator Name'] == yaxis_column_name]['
            mode='markers',
            marker={
                'size': 15,
                'opacity': 0.5,
                'line': {'width': 0.5, 'color': 'white'}
            }
        )],
        'layout': go.Layout(
            xaxis={
                'title': xaxis_column_name,
                'type': 'linear' if xaxis_type == 'Linear' else 'log'
            },
            yaxis={
                'title': yaxis_column_name,
                'type': 'linear' if yaxis_type == 'Linear' else 'log'
            },
            margin={'l': 40, 'b': 30, 't': 10, 'r': 0},
            height=450,
            hovermode='closest'
        )
    }


def create_time_series(dff, axis_type, title):
    return {
        'data': [go.Scatter(
            x=dff['Year'],
```

```
            y=dff['Value'],
            mode='lines+markers'
        )],
        'layout': {
            'height': 225,
            'margin': {'l': 20, 'b': 30, 'r': 10, 't': 10},
            'annotations': [{
                'x': 0, 'y': 0.85, 'xanchor': 'left', 'yanchor': 'bottom
                'xref': 'paper', 'yref': 'paper', 'showarrow': False,
                'align': 'left', 'bgcolor': 'rgba(255, 255, 255, 0.5)',
                'text': title
            }],
            'yaxis': {'type': 'linear' if axis_type == 'Linear' else 'lo
            'xaxis': {'showgrid': False}
        }
    }


@app.callback(
    dash.dependencies.Output('x-time-series', 'figure'),
    [dash.dependencies.Input('crossfilter-indicator-scatter', 'hoverData
     dash.dependencies.Input('crossfilter-xaxis-column', 'value'),
     dash.dependencies.Input('crossfilter-xaxis-type', 'value')])
def update_y_timeseries(hoverData, xaxis_column_name, axis_type):
    country_name = hoverData['points'][0]['customdata']
    dff = df[df['Country Name'] == country_name]
    dff = dff[dff['Indicator Name'] == xaxis_column_name]
    title = '<b>{}</b><br>{}'.format(country_name, xaxis_column_name)
    return create_time_series(dff, axis_type, title)


@app.callback(
    dash.dependencies.Output('y-time-series', 'figure'),
    [dash.dependencies.Input('crossfilter-indicator-scatter', 'hoverData
     dash.dependencies.Input('crossfilter-yaxis-column', 'value'),
     dash.dependencies.Input('crossfilter-yaxis-type', 'value')])
def update_x_timeseries(hoverData, yaxis_column_name, axis_type):
    dff = df[df['Country Name'] == hoverData['points'][0]['customdata']]
    dff = dff[dff['Indicator Name'] == yaxis_column_name]
    return create_time_series(dff, axis_type, yaxis_column_name)
```

```
if __name__ == '__main__':
    app.run_server(debug=True)
```



Try mousing over the points in the scatter plot on the left. Notice how the line graphs on the right update based off of the point that you are hovering over.

Generic Crossfilter Recipe

▼ Here's a slightly more generic example for crossfiltering across a six-column data set. Each scatter plot's selection filters the underlying dataset.

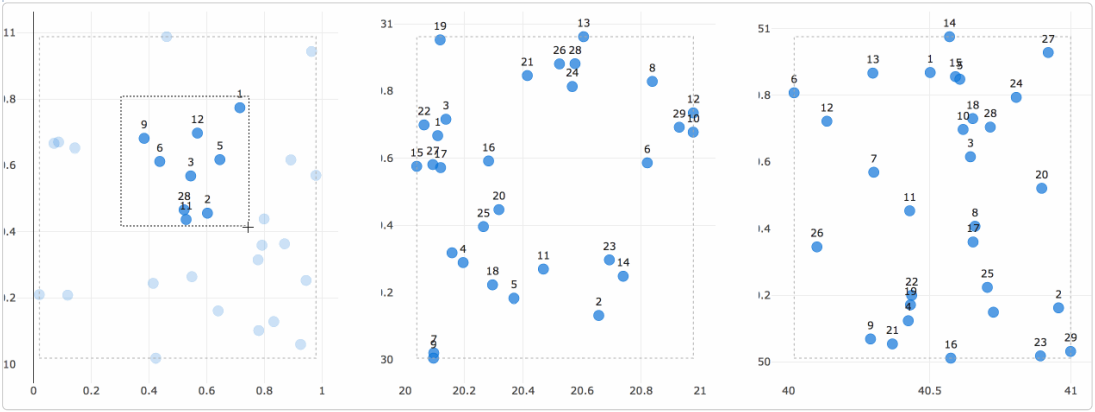
```
import dash
import dash_core_components as dcc
import dash_html_components as html
import numpy as np
import pandas as pd
from dash.dependencies import Input, Output

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']
```

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

np.random.seed(0)
df = pd.DataFrame({
    'Column {}'.format(i): np.random.rand(30) + i*10
    for i in range(6)})

app.layout = html.Div([
    html.Div(
        dcc.Graph(
            id='g1',
            config={'displayModeBar': False}
        ), className='four columns'
    ),
    html.Div(
        dcc.Graph(
            id='g2',
            config={'displayModeBar': False}
        ), className='four columns'),
    html.Div(
        dcc.Graph(
            id='g3',
            config={'displayModeBar': False}
        ), className='four columns')
], className='row')


def highlight(x, y):
    def callback(*selectedDatas):
        selectedpoints = df.index
        for i, selected_data in enumerate(selectedDatas):
            if selected_data is not None:
                selected_index = [
                    p['customdata'] for p in selected_data['points']
                ]
                if len(selected_index) > 0:
                    selectedpoints = np.intersect1d(
                        selectedpoints, selected_index)


        # set which points are selected with the `selectedpoints` proper
```

```
# and style those points with the `selected` and `unselected`
# attribute. see
# https://medium.com/@plotlygraphs/notes-from-the-latest-plotly-
# for an explanation

figure = {
    'data': [
        {
            'x': df[x],
            'y': df[y],
            'text': df.index,
            'textposition': 'top',
            'selectedpoints': selectedpoints,
            'customdata': df.index,
            'type': 'scatter',
            'mode': 'markers+text',
            'marker': {
                'color': 'rgba(0, 116, 217, 0.7)',
                'size': 12,
                'line': {
                    'color': 'rgb(0, 116, 217)',
                    'width': 0.5
                }
            },
            'textfont': {
                'color': 'rgba(30, 30, 30, 1)'
            },
            'unselected': {
                'marker': {
                    'opacity': 0.3,
                },
                'textfont': {
                    # make text transparent when not selected
                    'color': 'rgba(0, 0, 0, 0)'
                }
            }
        },
    ],
    'layout': {
        'clickmode': 'event+select',
        'margin': {'l': 15, 'r': 0, 'b': 15, 't': 5},
        'dragmode': 'select',
```

```
                'hovermode': 'closest',
                'showlegend': False
            }
        }

        # Display a rectangle to highlight the previously selected regio
        shape = {
            'type': 'rect',
            'line': {
                'width': 1,
                'dash': 'dot',
                'color': 'darkgrey'
            }
        }
        if selectedDatas[0] and selectedDatas[0]['range']:
            figure['layout']['shapes'] = [dict({
                'x0': selectedDatas[0]['range']['x'][0],
                'x1': selectedDatas[0]['range']['x'][1],
                'y0': selectedDatas[0]['range']['y'][0],
                'y1': selectedDatas[0]['range']['y'][1]
            }, **shape)]
        else:
            figure['layout']['shapes'] = [dict({
                'type': 'rect',
                'x0': np.min(df[x]),
                'x1': np.max(df[x]),
                'y0': np.min(df[y]),
                'y1': np.max(df[y])
            }, **shape)]

        return figure

    return callback



# app.callback is a decorator which means that it takes a function
# as its argument.
# highlight is a function "generator": it's a function that returns func
app.callback(
    Output('g1', 'figure'),
    [Input('g1', 'selectedData'),
```

```
     Input('g2', 'selectedData'),
     Input('g3', 'selectedData')]
)(highlight('Column 0', 'Column 1'))

app.callback(
    Output('g2', 'figure'),
    [Input('g2', 'selectedData'),
     Input('g1', 'selectedData'),
     Input('g3', 'selectedData')]
)(highlight('Column 2', 'Column 3'))

app.callback(
    Output('g3', 'figure'),
    [Input('g3', 'selectedData'),
     Input('g1', 'selectedData'),
     Input('g2', 'selectedData')]
)(highlight('Column 4', 'Column 5'))

if __name__ == '__main__':
    app.run_server(debug=True)
```



Try clicking and dragging in any of the plots to filter different regions. On every selection, the three graph callbacks are fired with the latest selected regions of each plot. A pandas dataframe is filtered based off of the selected points and the graphs are replotted with the selected points highlighted and the selected region drawn as a dashed rectangle.

> As an aside, if you find yourself filtering and visualizing highly-dimensional datasets, you should consider checking out the parallel coordinates chart type.

Current Limitations

There are a few limitations in graph interactions right now.

- It is not currently possible to customize the style of the hover interactions or the select box. This issue is being worked on in https://github.com/plotly/plotly.js/issues/1847 .

There's a lot that you can do with these interactive plotting features. If you need help exploring your use case, open up a thread in the Dash Community Forum .

The next chapter of the Dash User Guide explains how to share data between callbacks.

Dash Tutorial Part 6. Sharing Data Between Callbacks

Sharing State Between Callbacks

> This is the 6th chapter of the essential Dash Tutorial. The previous chapter covered how to use callbacks with the `dash_core_components.Graph` component. The rest of the Dash documentation covers other topics like multi-page apps and component libraries. Just getting started? Make sure to install the necessary dependencies. The next and final chapter covers frequently asked questions and gotchas.

One of the core Dash principles explained in the Getting Started Guide on Callbacks is that **Dash Callbacks must never modify variables outside of their scope**. It is not safe to modify any `global` variables. This chapter explains why and provides some alternative patterns for sharing state between callbacks.

Why Share State?

In some apps, you may have multiple callbacks that depend on expensive data processing tasks like making SQL queries, running simulations, or downloading data.

Rather than have each callback run the same expensive task, you can have one callback run the task and then share the results to the rest of the callbacks.

Why `global` variables will break your app

Dash is designed to work in multi-user environments where multiple people may view the application at the same time and will have **independent sessions**.

If your app uses modified `global` variables, then one user's session could set the variable to one value which would affect the next user's session.

Dash is also designed to be able to run with **multiple python workers** so that callbacks can be executed in parallel. This is commonly done with `gunicorn` using syntax like

```
$ gunicorn --workers 4 app:server
```

(`app` refers to a file named `app.py` and `server` refers to a variable in that file named `server`: `server = app.server`).

When Dash apps run across multiple workers, their memory is not shared. This means that if you modify a global variable in one callback, that modification will not be applied to the rest of the workers.

▼ Here is a sketch of an app with a callback that modifies data out of it's scope. This type of pattern *will not work reliably* for the reasons outlined above.

```
df = pd.DataFrame({
    'a': [1, 2, 3],
    'b': [4, 1, 4],
    'c': ['x', 'y', 'z'],
})

app.layout = html.Div([
    dcc.Dropdown(
        id='dropdown',
        options=[{'label': i, 'value': i} for i in df['c'].unique()],
        value='a'
    ),
    html.Div(id='output'),
])

@app.callback(Output('output', 'children'),
              [Input('dropdown', 'value')])
def update_output_1(value):
    # Here, `df` is an example of a variable that is
    # "outside the scope of this function".
    # *It is not safe to modify or reassign this variable
    #  inside this callback.*
    global df = df[df['c'] == value]  # do not do this, this is not safe
    return len(df)
```

▼ To fix this example, simply re-assign the filter to a new variable inside the callback, or follow one of the strategies outlined in the next part of this guide.

```
df = pd.DataFrame({
    'a': [1, 2, 3],
    'b': [4, 1, 4],
    'c': ['x', 'y', 'z'],
})

app.layout = html.Div([
    dcc.Dropdown(
        id='dropdown',
        options=[{'label': i, 'value': i} for i in df['c'].unique()],
        value='a'
    ),
    html.Div(id='output'),
```

```
])

@app.callback(Output('output', 'children'),
              [Input('dropdown', 'value')])
def update_output_1(value):
    # Safely reassign the filter to a new variable
    filtered_df = df[df['c'] == value]
    return len(filtered_df)
```

Sharing Data Between Callbacks

In order to share data safely across multiple python processes, we need to store the data somewhere that is accessible to each of the processes.

There are three main places to store this data:

1 - In the user's browser session

2 - On the disk (e.g. on a file or on a new database)

3 - In a shared memory space like with Redis

The following three examples illustrate these approaches.

Example 1 - Storing Data in the Browser with a Hidden Div

To save data in user's browser's session:

- Implemented by saving the data as part of Dash's front-end store through methods explained in https://community.plot.ly/t/sharing-a-dataframe-between-plots/6173
- Data has to be converted to a string like JSON for storage and transport
- Data that is cached in this way will only be available in the user's current session.
    - If you open up a new browser, the app's callbacks will always compute the data. The data is only cached and transported between callbacks within the session.
    - As such, unlike with caching, this method doesn't increase the memory footprint of the app.
    - There could be a cost in network transport. If you're sharing 10MB of data between callbacks, then that data will be transported over the network between each callback.
    - If the network cost is too high, then compute the aggregations upfront and transport those. Your app likely won't be displaying 10MB of data, it will just be displaying a subset or an aggregation of it.

▼ This example outlines how you can perform an expensive data processing step in one callback, serialize the output at JSON, and provide it as an input to the other callbacks. This example uses

standard Dash callbacks and stores the JSON-ified data inside a hidden div in the app.

```
global_df = pd.read_csv('...')
app.layout = html.Div([
    dcc.Graph(id='graph'),
    html.Table(id='table'),
    dcc.Dropdown(id='dropdown'),

    # Hidden div inside the app that stores the intermediate value
    html.Div(id='intermediate-value', style={'display': 'none'})
])

@app.callback(Output('intermediate-value', 'children'), [Input('dropdown
def clean_data(value):
     # some expensive clean data step
     cleaned_df = your_expensive_clean_or_compute_step(value)

     # more generally, this line would be
     # json.dumps(cleaned_df)
     return cleaned_df.to_json(date_format='iso', orient='split')

@app.callback(Output('graph', 'figure'), [Input('intermediate-value', 'c
def update_graph(jsonified_cleaned_data):

    # more generally, this line would be
    # json.loads(jsonified_cleaned_data)
    dff = pd.read_json(jsonified_cleaned_data, orient='split')

    figure = create_figure(dff)
    return figure

@app.callback(Output('table', 'children'), [Input('intermediate-value',
def update_table(jsonified_cleaned_data):
    dff = pd.read_json(jsonified_cleaned_data, orient='split')
    table = create_table(dff)
    return table
```

Example 2 - Computing Aggregations Upfront

Sending the computed data over the network can be expensive if the data is large. In some cases, serializing this data and JSON can also be expensive.

In many cases, your app will only display a subset or an aggregation of the computed or filtered data. In these cases, you could precompute your aggregations in your data processing callback and transport these aggregations to the remaining callbacks.

▼ Here's a simple example of how you might transport filtered or aggregated data to multiple callbacks.

```
@app.callback(
    Output('intermediate-value', 'children'),
    [Input('dropdown', 'value')])
def clean_data(value):
     # an expensive query step
     cleaned_df = your_expensive_clean_or_compute_step(value)

     # a few filter steps that compute the data
     # as it's needed in the future callbacks
     df_1 = cleaned_df[cleaned_df['fruit'] == 'apples']
     df_2 = cleaned_df[cleaned_df['fruit'] == 'oranges']
     df_3 = cleaned_df[cleaned_df['fruit'] == 'figs']

     datasets = {
         'df_1': df_1.to_json(orient='split', date_format='iso'),
         'df_2': df_2.to_json(orient='split', date_format='iso'),
         'df_3': df_3.to_json(orient='split', date_format='iso'),
     }

     return json.dumps(datasets)

@app.callback(
    Output('graph', 'figure'),
    [Input('intermediate-value', 'children')])
def update_graph_1(jsonified_cleaned_data):
    datasets = json.loads(jsonified_cleaned_data)
    dff = pd.read_json(datasets['df_1'], orient='split')
    figure = create_figure_1(dff)
    return figure

@app.callback(
    Output('graph', 'figure'),
```

```
    [Input('intermediate-value', 'children')])
def update_graph_2(jsonified_cleaned_data):
    datasets = json.loads(jsonified_cleaned_data)
    dff = pd.read_json(datasets['df_2'], orient='split')
    figure = create_figure_2(dff)
    return figure

@app.callback(
    Output('graph', 'figure'),
    [Input('intermediate-value', 'children')])
def update_graph_3(jsonified_cleaned_data):
    datasets = json.loads(jsonified_cleaned_data)
    dff = pd.read_json(datasets['df_3'], orient='split')
    figure = create_figure_3(dff)
    return figure
```

Example 3 - Caching and Signaling

This example:

- Uses Redis via Flask-Cache for storing “global variables”. This data is accessed through a function, the output of which is cached and keyed by its input arguments.
- Uses the hidden div solution to send a signal to the other callbacks when the expensive computation is complete.
- Note that instead of Redis, you could also save this to the file system. See https://flask-caching.readthedocs.io/en/latest/ for more details.
- This “signaling” is cool because it allows the expensive computation to only take up one process. Without this type of signaling, each callback could end up computing the expensive computation in parallel, locking four processes instead of one.

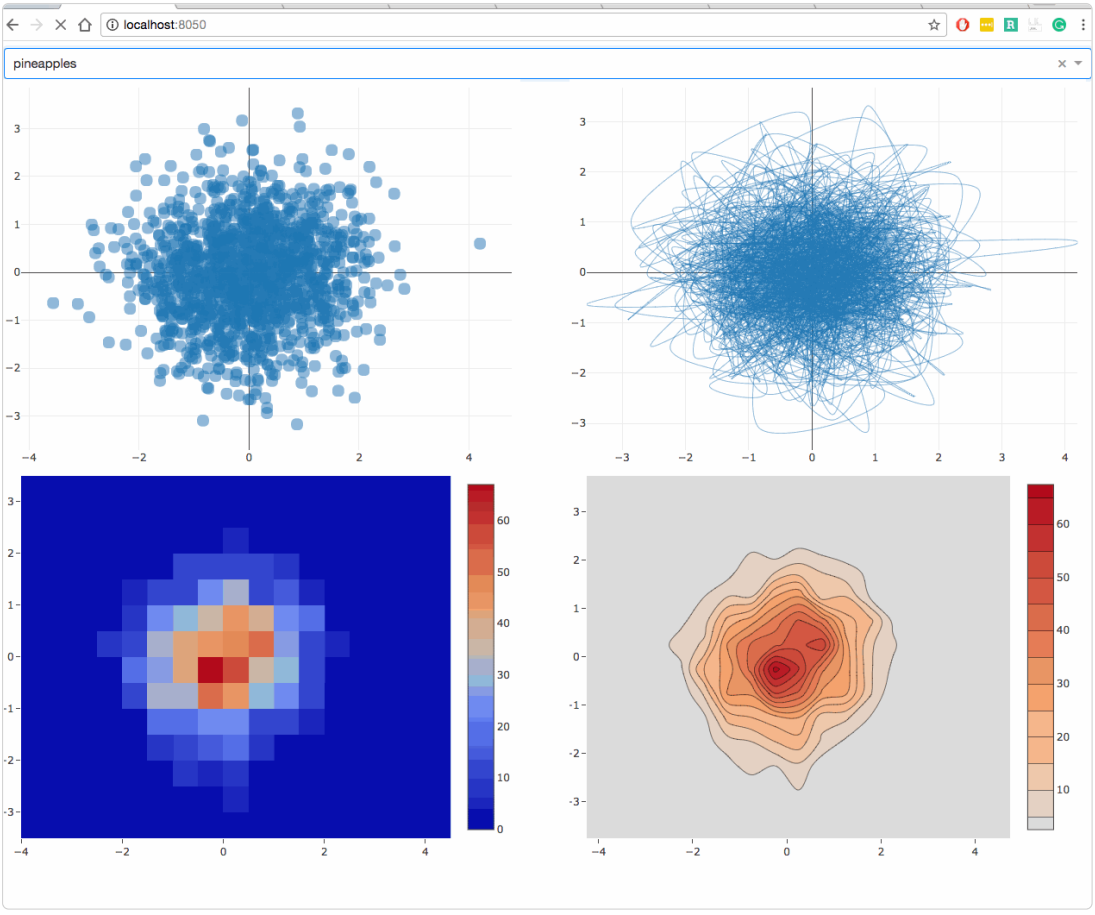
This approach is also advantageous in that future sessions can use the pre-computed value. This will work well for apps that have a small number of inputs.

Here’s what this example looks like. Some things to note:

- I’ve simulated an expensive process by using a time.sleep(5).
- When the app loads, it takes five seconds to render all four graphs.
- The initial computation only blocks one process.
- Once the computation is complete, the signal is sent and four callbacks are executed in parallel to render the graphs. Each of these callbacks retrieves the data from the “global store”: the Redis

or filesystem cache.

- I've set processes=6 in app.run_server so that multiple callbacks can be executed in parallel. In production, this is done with something like $ gunicorn --workers 6 --threads 2 app:server
- Selecting a value in the dropdown will take less than five seconds if it has already been selected in the past. This is because the value is being pulled from the cache.
- Similarly, reloading the page or opening the app in a new window is also fast because the initial state and the initial expensive computation has already been computed.



▼ Here's what this example looks like in code:

```
import os
import copy
import time
import datetime

import dash
import dash_core_components as dcc
```

```
import dash_html_components as html
import numpy as np
import pandas as pd
from dash.dependencies import Input, Output
from flask_caching import Cache


external_stylesheets = [
    # Dash CSS
    'https://codepen.io/chriddyp/pen/bWLwgP.css',
    # Loading screen CSS
    'https://codepen.io/chriddyp/pen/brPBPO.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
CACHE_CONFIG = {
    # try 'filesystem' if you don't want to setup redis
    'CACHE_TYPE': 'redis',
    'CACHE_REDIS_URL': os.environ.get('REDIS_URL', 'localhost:6379')
}
cache = Cache()
cache.init_app(app.server, config=CACHE_CONFIG)

N = 100

df = pd.DataFrame({
    'category': (
        (['apples'] * 5 * N) +
        (['oranges'] * 10 * N) +
        (['figs'] * 20 * N) +
        (['pineapples'] * 15 * N)
    )
})
df['x'] = np.random.randn(len(df['category']))
df['y'] = np.random.randn(len(df['category']))

app.layout = html.Div([
    dcc.Dropdown(
        id='dropdown',
        options=[{'label': i, 'value': i} for i in df['category'].unique
        value='apples'
    ),
    html.Div([
```

```
        html.Div(dcc.Graph(id='graph-1'), className="six columns"),
        html.Div(dcc.Graph(id='graph-2'), className="six columns"),
    ], className="row"),
    html.Div([
        html.Div(dcc.Graph(id='graph-3'), className="six columns"),
        html.Div(dcc.Graph(id='graph-4'), className="six columns"),
    ], className="row"),

    # hidden signal value
    html.Div(id='signal', style={'display': 'none'})
])


# perform expensive computations in this "global store"
# these computations are cached in a globally available
# redis memory store which is available across processes
# and for all time.
@cache.memoize()
def global_store(value):
    # simulate expensive query
    print('Computing value with {}'.format(value))
    time.sleep(5)
    return df[df['category'] == value]


def generate_figure(value, figure):
    fig = copy.deepcopy(figure)
    filtered_dataframe = global_store(value)
    fig['data'][0]['x'] = filtered_dataframe['x']
    fig['data'][0]['y'] = filtered_dataframe['y']
    fig['layout'] = {'margin': {'l': 20, 'r': 10, 'b': 20, 't': 10}}
    return fig


@app.callback(Output('signal', 'children'), [Input('dropdown', 'value')])
def compute_value(value):
    # compute value and send a signal when done
    global_store(value)
    return value


@app.callback(Output('graph-1', 'figure'), [Input('signal', 'children')])
```

```
def update_graph_1(value):
    # generate_figure gets data from `global_store`.
    # the data in `global_store` has already been computed
    # by the `compute_value` callback and the result is stored
    # in the global redis cached
    return generate_figure(value, {
        'data': [{
            'type': 'scatter',
            'mode': 'markers',
            'marker': {
                'opacity': 0.5,
                'size': 14,
                'line': {'border': 'thin darkgrey solid'}
            }
        }]
    })


@app.callback(Output('graph-2', 'figure'), [Input('signal', 'children')])
def update_graph_2(value):
    return generate_figure(value, {
        'data': [{
            'type': 'scatter',
            'mode': 'lines',
            'line': {'shape': 'spline', 'width': 0.5},
        }]
    })


@app.callback(Output('graph-3', 'figure'), [Input('signal', 'children')])
def update_graph_3(value):
    return generate_figure(value, {
        'data': [{
            'type': 'histogram2d',
        }]
    })


@app.callback(Output('graph-4', 'figure'), [Input('signal', 'children')])
def update_graph_4(value):
    return generate_figure(value, {
        'data': [{
```

```
            'type': 'histogram2dcontour',
        }]
    })


if __name__ == '__main__':
    app.run_server(debug=True, processes=6)
```

Example 4 - User-Based Session Data on the Server

The previous example cached computations on the filesystem and those computations were accessible for all users.

In some cases, you want to keep the data isolated to user sessions: one user's derived data shouldn't update the next user's derived data. One way to do this is to save the data in a hidden `Div`, as demonstrated in the first example.

Another way to do this is to save the data on the filesystem cache with a session ID and then reference the data using that session ID. Because data is saved on the server instead of transported over the network, this method is generally faster than the "hidden div" method.

This example was originally discussed in a Dash Community Forum thread.

This example:

- Caches data using the `flask_caching` filesystem cache. You can also save to an in-memory database like Redis.
- Serializes the data as JSON.
    - If you are using Pandas, consider serializing with Apache Arrow. Community thread
- Saves session data up to the number of expected concurrent users. This prevents the cache from being overfilled with data.
- Creates unique session IDs by embedding a hidden random string into the app's layout and serving a unique layout on every page load.

> Note: As with all examples that send data to the client, be aware that these sessions aren't necessarily secure or encrypted. These session IDs may be vulnerable to Session Fixation style attacks.

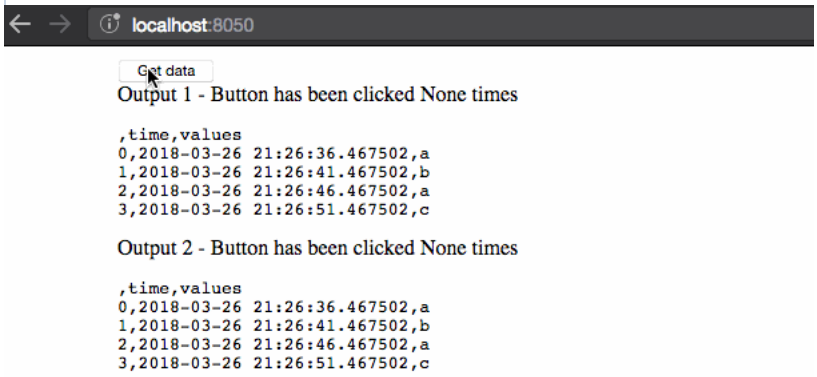
▼ Here's what this example looks like in code:

```
import dash
from dash.dependencies import Input, Output
import dash_core_components as dcc
```

```
import dash_html_components as html
import datetime
from flask_caching import Cache
import os
import pandas as pd
import time
import uuid

external_stylesheets = [
    # Dash CSS
    'https://codepen.io/chriddyp/pen/bWLwgP.css',
    # Loading screen CSS
    'https://codepen.io/chriddyp/pen/brPBPO.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
cache = Cache(app.server, config={
    'CACHE_TYPE': 'redis',
    # Note that filesystem cache doesn't work on systems with ephemeral
    # filesystems like Heroku.
    'CACHE_TYPE': 'filesystem',
    'CACHE_DIR': 'cache-directory',

    # should be equal to maximum number of users on the app at a single
    # higher numbers will store more data in the filesystem / redis cach
    'CACHE_THRESHOLD': 200
})


def get_dataframe(session_id):
    @cache.memoize()
    def query_and_serialize_data(session_id):
        # expensive or user/session-unique data processing step goes her

        # simulate a user/session-unique data processing step by generat
        # data that is dependent on time
        now = datetime.datetime.now()

        # simulate an expensive data processing task by sleeping
        time.sleep(5)

        df = pd.DataFrame({
            'time': [
```

```
                str(now - datetime.timedelta(seconds=15)),
                str(now - datetime.timedelta(seconds=10)),
                str(now - datetime.timedelta(seconds=5)),
                str(now)
            ],
            'values': ['a', 'b', 'a', 'c']
        })
        return df.to_json()

    return pd.read_json(query_and_serialize_data(session_id))


def serve_layout():
    session_id = str(uuid.uuid4())

    return html.Div([
        html.Div(session_id, id='session-id', style={'display': 'none'})
        html.Button('Get data', id='button'),
        html.Div(id='output-1'),
        html.Div(id='output-2')
    ])


app.layout = serve_layout


@app.callback(Output('output-1', 'children'),
              [Input('button', 'n_clicks'),
               Input('session-id', 'children')])
def display_value_1(value, session_id):
    df = get_dataframe(session_id)
    return html.Div([
        'Output 1 - Button has been clicked {} times'.format(value),
        html.Pre(df.to_csv())
    ])


@app.callback(Output('output-2', 'children'),
              [Input('button', 'n_clicks'),
               Input('session-id', 'children')])
def display_value_2(value, session_id):
    df = get_dataframe(session_id)
```

```
    return html.Div([
        'Output 2 - Button has been clicked {} times'.format(value),
        html.Pre(df.to_csv())
    ])


if __name__ == '__main__':
    app.run_server(debug=True)
```



There are three things to notice in this example:

- The timestamps of the dataframe don't update when we retrieve the data. This data is cached as part of the user's session.
- Retrieving the data initially takes five seconds but successive queries are instant, as the data has been cached.
- The second session displays different data than the first session: the data that is shared between callbacks is isolated to individual user sessions.

Questions? Discuss these examples on the Dash Community Forum.

FAQs and Gotchas

> This is the 7th and final chapter of the essential Dash Tutorial. The previous chapter described how to share data between callbacks. The rest of the Dash documentation covers other topics like multi-page apps and component libraries.

Frequently Asked Questions

Q: How can I customize the appearance of my Dash app?

A: Dash apps are rendered in the browser as modern standards compliant web apps. This means that you can use CSS to style your Dash app as you would standard HTML.

All `dash-html-components` support inline CSS styling through a `style` attribute. An external CSS stylesheet can also be used to style `dash-html-components` and `dash-core-components` by targeting the ID or class names of your components. Both `dash-html-components` and `dash-core-components` accept the attribute `className`, which corresponds to the HTML element attribute `class`.

The Dash HTML Components section in the Dash User Guide explains how to supply `dash-html-components` with both inline styles and CSS class names that you can target with CSS style sheets. The Adding CSS & JS and Overriding the Page-Load Template section in the Dash Guide explains how you can link your own style sheets to Dash apps.

Q: How can I add JavaScript to my Dash app?

A: You can add your own scripts to your Dash app, just like you would add a JavaScript file to an HTML document. See the Adding CSS & JS and Overriding the Page-Load Template section in the Dash Guide.

Q: Can I make a Dash app with multiple pages?

A: Yes! Dash has support for multi-page apps. See the Multi-Page Apps and URL Support section in the Dash User Guide.

Q: How I can I organise my Dash app into multiple files?

A: A strategy for doing this can be found in the Multi-Page Apps and URL Support section in the Dash User Guide.

Q: How do I determine which `Input` has changed?

A: In addition to the `n_clicks` property (which tracks the number of times a component has been clicked), all `dash-html-components` have an `n_clicks_timestamp` property, which records the time that the component was last clicked. This provides a convenient way for detecting which `html.Button` was clicked in order to trigger the current callback. Here's an example of how this can be done:

```
import dash
import dash_html_components as html
from dash.dependencies import Input, Output

app = dash.Dash(__name__)

app.layout = html.Div([
    html.Button('Button 1', id='btn-1', n_clicks_timestamp='0'),
    html.Button('Button 2', id='btn-2', n_clicks_timestamp='0'),
    html.Button('Button 3', id='btn-3', n_clicks_timestamp='0'),
    html.Div(id='container')
])


@app.callback(Output('container', 'children'),
              [Input('btn-1', 'n_clicks_timestamp'),
               Input('btn-2', 'n_clicks_timestamp'),
               Input('btn-3', 'n_clicks_timestamp')])
def display(btn1, btn2, btn3):
    if int(btn1) > int(btn2) and int(btn1) > int(btn3):
        msg = 'Button 1 was most recently clicked'
    elif int(btn2) > int(btn1) and int(btn2) > int(btn3):
        msg = 'Button 2 was most recently clicked'
    elif int(btn3) > int(btn1) and int(btn3) > int(btn2):
        msg = 'Button 3 was most recently clicked'
    else:
        msg = 'None of the buttons have been clicked yet'
    return html.Div([
        html.Div('btn1: {}'.format(btn1)),
        html.Div('btn2: {}'.format(btn2)),
        html.Div('btn3: {}'.format(btn3)),
        html.Div(msg)
    ])
```

```
if __name__ == '__main__':
    app.run_server(debug=True)
```

BUTTON 1 BUTTON 2 BUTTON 3

btn1: 0

btn2: 0

btn3: 0

None of the buttons have been clicked yet

Note that `n_clicks` is the only property that has this timestamp property. We will add general support for "determining which input changed" in the future, you can track our progress in this GitHub Issue.

Q: Can I use Jinja2 templates with Dash?

A: Jinja2 templates are rendered on the server (often in a Flask app) before being sent to the client as HTML pages. Dash apps, on the other hand, are rendered on the client using React. This makes these fundamentally different approaches to displaying HTML in a browser, which means the two approaches can't be combined directly. You can however integrate a Dash app with an existing Flask app such that the Flask app handles some URL endpoints, while your Dash app lives at a specific URL endpoint.

Q: Can I use jQuery with Dash?

A: For the most part, you can't. Dash uses React to render your app on the client browser. React is fundamentally different to jQuery in that it makes use of a virtual DOM (Document Object Model) to manage page rendering. Since jQuery doesn't speak React's virtual DOM, you can't use any of jQuery's DOM manipulation facilities to change the page layout, which is frequently why one might want to use jQuery. You can however use parts of jQuery's functionality that do not touch the DOM, such as registering event listeners to cause a page redirect on a keystroke.

In general, if you are looking to add custom clientside behavior in your application, we recommend encapsulating that behavior in a custom Dash component.

Q: I have more questions! Where can I go to ask them?

A: The Dash Community forums is full of people discussing Dash topics, helping each other with questions, and sharing Dash creations. Jump on over and join the discussion.

Gotchas

There are some aspects of how Dash works that can be counter-intuitive. This can be especially true of how the callback system works. This section outlines some common Dash gotchas that you might encounter as you start building out more complex Dash apps. If you have read through the rest of the Dash Tutorial and are encountering unexpected behaviour, this is a good section to read through. If you still have residual questions, the Dash Community forums is a great place to ask them.

Callbacks require their `Inputs`, `States`, and `Output` to be present in the layout

By default, Dash applies validation to your callbacks, which performs checks such as validating the types of callback arguments and checking to see whether the specified `Input` and `Output` components actually have the specified properties. For full validation, all components within your callback must therefore appear in the initial layout of your app, and you will see an error if they do not.

However, in the case of more complex Dash apps that involve dynamic modification of the layout (such as multi-page apps), not every component appearing in your callbacks will be included in the initial layout. You can remove this restriction by disabling callback validation like this:

```
app.config.supress_callback_exceptions = True
```

Callbacks require *all* `Inputs`, `States`, and `Output` to be rendered on the page

If you have disabled callback validation in order to support dynamic layouts, then you won't be automatically alerted to the situation where a component within a callback is not found within a layout. In this situation, where a component registered with a callback is missing from the layout, the callback will fail to fire. For example, if you define a callback with only a subset of the specified `Inputs` present in the current page layout, the callback will simply not fire at all.

Callbacks can only target a single `Output` component/property pair

Currently, for a given callback, it can only have a single `Output`, which targets one component/property pair eg `'my-graph'`, `'figure'`. If you wanted, say, four `Graph` components to be updated based on a particular user input, you either need to create four

separate callbacks which each target an individual `Graph`, or have the callback return a `html.Div` container that holds the updated four Graphs.

There are plans to remove this limitation. You can track the status of this in this GitHub Issue.

A component/property pair can only be the `Output` of one callback

For a given component/property pair (eg `'my-graph'`, `'figure'`), it can only be registered as the `Output` of one callback. If you want to associate two logically separate sets of `Inputs` with the one output component/property pair, you'll have to bundle them up into a larger callback and detect which of the relevant `Inputs` triggered the callback inside the function. For `html.Button` elements, detecting which one triggered the callback ca be done using the `n_clicks_timestamp` property. For an example of this, see the question in the FAQ, How do I determine which `Input` has changed?.

All callbacks must be defined before the server starts

All your callbacks must be defined before your Dash app's server starts running, which is to say, before you call `app.run_server(debug=True)`. This means that while you can assemble changed layout fragments dynamically during the handling of a callback, you can't define dynamic callbacks in response to user input during the handling of a callback. If you have a dynamic interface, where a callback changes the layout to include a different set of input controls, then you must have already defined the callbacks required to service these new controls in advance.

For example, a common scenario is a `Dropdown` component that updates the current layout to replace a dashboard with another logically distinct dashboard that has a different set of controls (the number and type of which might which might depend on other user input) and different logic for generating the underlying data. A sensible organisation would be for each of these dashboards to have separate callbacks. In this scenario, each of these callbacks much then be defined before the app starts running.

Generally speaking, if a feature of your Dash app is that the number of `Inputs` or `States` is determined by a user's input, then you must pre-define up front every permutation of callback that a user can potentially trigger. For an example of how this can be done programmatically using the `callback` decorator, see this Dash Community forum post.

All Dash Core Components in a layout should be registered with a callback.

If a Dash Core Component is present in the layout but not registered with a callback (either as an `Input`, `State`, or `Output`) then any changes to its value by the user will be reset to the original

value when any callback updates the page.

This is a known issue and you can track its status in this GitHub Issue.

Component Libraries

Dash Core Components

Dash ships with supercharged components for interactive user interfaces. A core set of components, written and maintained by the Dash team, is available in the `dash-core-components` library.

The source is on GitHub at plotly/dash-core-components.

These docs are using version 0.42.1.

```
>>> import dash_core_components as dcc
>>> print(dcc.__version__)
0.42.1
```

Dropdown

```
import dash_core_components as dcc

dcc.Dropdown(
    options=[
        {'label': 'New York City', 'value': 'NYC'},
        {'label': 'Montréal', 'value': 'MTL'},
        {'label': 'San Francisco', 'value': 'SF'}
    ],
    value='MTL'
)
```

Montréal

```
import dash_core_components as dcc

dcc.Dropdown(
    options=[
        {'label': 'New York City', 'value': 'NYC'},
        {'label': 'Montréal', 'value': 'MTL'},
        {'label': 'San Francisco', 'value': 'SF'}
    ],
    multi=True,
```

```
    value="MTL"
)
```

More Dropdown Examples and Reference

Slider

```
import dash_core_components as dcc

dcc.Slider(
    min=-5,
    max=10,
    step=0.5,
    value=-3,
)
```



```
import dash_core_components as dcc

dcc.Slider(
    min=0,
    max=9,
    marks={i: 'Label {}'.format(i) for i in range(10)},
    value=5,
)
```


More Slider Examples and Reference

RangeSlider

```
import dash_core_components as dcc
```

```
dcc.RangeSlider(
    count=1,
    min=-5,
    max=10,
    step=0.5,
    value=[-3, 7]
)
```

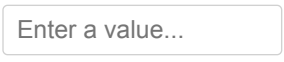


```
import dash_core_components as dcc

dcc.RangeSlider(
    marks={i: 'Label {}'.format(i) for i in range(-5, 7)},
    min=-5,
    max=6,
    value=[-3, 4]
)
```


More RangeSlider Examples and Reference

Input

```
import dash_core_components as dcc

dcc.Input(
    placeholder='Enter a value...',
    type='text',
    value=''
)
```



More Input Examples and Reference

Textarea

```
import dash_core_components as dcc

dcc.Textarea(
    placeholder='Enter a value...',
    value='This is a TextArea component',
    style={'width': '100%'}
)
```

This is a TextArea component

Textarea Reference

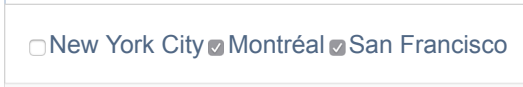
Checkboxes

```
import dash_core_components as dcc

dcc.Checklist(
    options=[
        {'label': 'New York City', 'value': 'NYC'},
        {'label': 'Montréal', 'value': 'MTL'},
        {'label': 'San Francisco', 'value': 'SF'}
    ],
    values=['MTL', 'SF']
)
```

- [ ] New York City
- [x] Montréal
- [x] San Francisco

```
import dash_core_components as dcc

dcc.Checklist(
    options=[
```

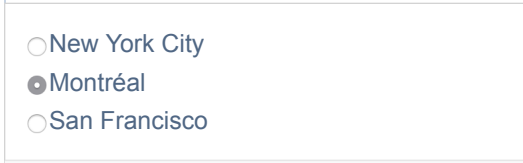
```
        {'label': 'New York City', 'value': 'NYC'},
        {'label': 'Montréal', 'value': 'MTL'},
        {'label': 'San Francisco', 'value': 'SF'}
    ],
    values=['MTL', 'SF'],
    labelStyle={'display': 'inline-block'}
)
```



Checklist Properties

Radio Items

```
import dash_core_components as dcc

dcc.RadioItems(
    options=[
        {'label': 'New York City', 'value': 'NYC'},
        {'label': 'Montréal', 'value': 'MTL'},
        {'label': 'San Francisco', 'value': 'SF'}
    ],
    value='MTL'
)
```




```
import dash_core_components as dcc

dcc.RadioItems(
    options=[
        {'label': 'New York City', 'value': 'NYC'},
        {'label': 'Montréal', 'value': 'MTL'},
        {'label': 'San Francisco', 'value': 'SF'}
    ],
    value='MTL',
```

```
    labelStyle={'display': 'inline-block'}
)
```

New York City Montréal San Francisco

RadioItems Reference

Button

```
import dash
import dash_html_components as html
import dash_core_components as dcc

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
app.layout = html.Div([
    html.Div(dcc.Input(id='input-box', type='text')),
    html.Button('Submit', id='button'),
    html.Div(id='output-container-button',
             children='Enter a value and press submit')
])


@app.callback(
    dash.dependencies.Output('output-container-button', 'children'),
    [dash.dependencies.Input('button', 'n_clicks')],
    [dash.dependencies.State('input-box', 'value')])
def update_output(n_clicks, value):
    return 'The input value was "{}" and the button has been clicked {}
        value,
        n_clicks
    )


if __name__ == '__main__':
    app.run_server(debug=True)
```

SUBMIT

The input value was "None" and the button has been clicked None times

More Button Examples and Reference

For more on `dash.dependencies.State`, see the tutorial on Dash State.

DatePickerSingle

```
import dash_core_components as dcc
from datetime import datetime as dt

dcc.DatePickerSingle(
    id='date-picker-single',
    date=dt(1997, 5, 10)
)
```

05/10/1997

More DatePickerSingle Examples and Reference

DatePickerRange

```
import dash_core_components as dcc
from datetime import datetime as dt

dcc.DatePickerRange(
    id='date-picker-range',
    start_date=dt(1997, 5, 3),
    end_date_placeholder_text='Select a date!'
)
```

05/03/1997 → Select a date!

More DatePickerRange Examples and Reference

Markdown

```
import dash_core_components as dcc

dcc.Markdown('''
#### Dash and Markdown

Dash supports [Markdown](http://commonmark.org/help).

Markdown is a simple way to write and format text.
It includes a syntax for things like **bold text** and *italics*,
[links](http://commonmark.org/help), inline `code` snippets, lists,
quotes, and more.
''')
```

Dash and Markdown

Dash supports Markdown.

Markdown is a simple way to write and format text. It includes a syntax for things like **bold text** and italics, links, inline `code` snippets, lists, quotes, and more.

More Markdown Examples and Reference

Interactive Tables

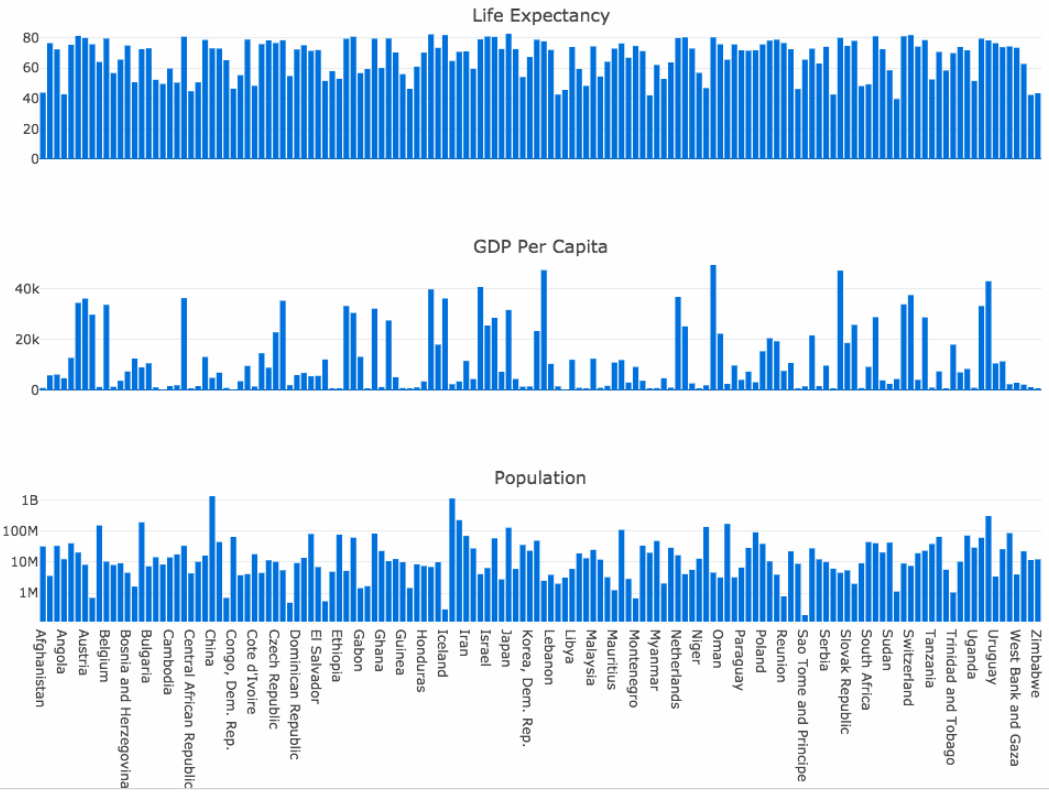
The `dash_html_components` library exposes all of the HTML tags. This includes the `Table`, `Tr`, and `Tbody` tags that can be used to create an HTML table. See Create Your First Dash App, Part 1 for an example.

Dash provides an interactive `DataTable` as part of the `data-table` project. This table includes built-in filtering, row-selection, editing, and sorting.

Dash DataTable

FILTER ROWS

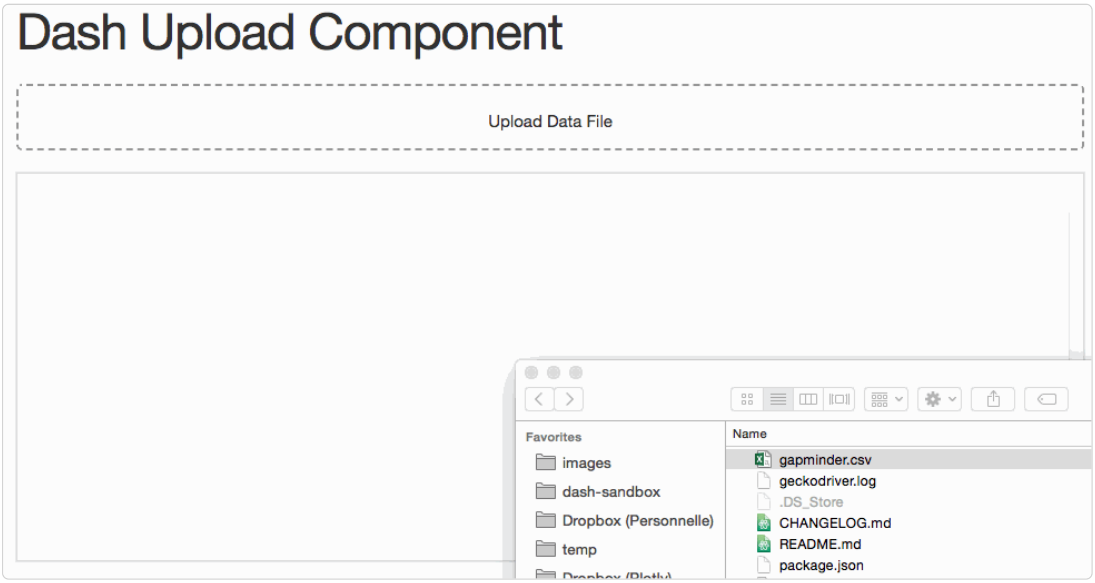
continent	country	gdpPercap	lifeExp	pop	year
Asia	Afghanistan	974.5803384	43.828	31889923	2007
Europe	Albania	5937.02952599999	76.423	3600523	2007
Africa	Algeria	6223.367465	72.301	33333216	2007
Africa	Angola	4797.231267	42.731	12420476	2007
Americas	Argentina	12779.3796400000	75.32	40301927	2007
Oceania	Australia	34435.3674399999	81.235	20434176	2007
Europe	Austria	36126.4927	79.829	8199783	2007
Asia	Bahrain	29796.0483399999	75.635	708573	2007
Asia	Bangladesh	1391.253792	64.062	150448339	2007



View the docs or View the source

Upload Component

The `dcc.Upload` component allows users to upload files into your app through drag-and-drop or the system's native file explorer.



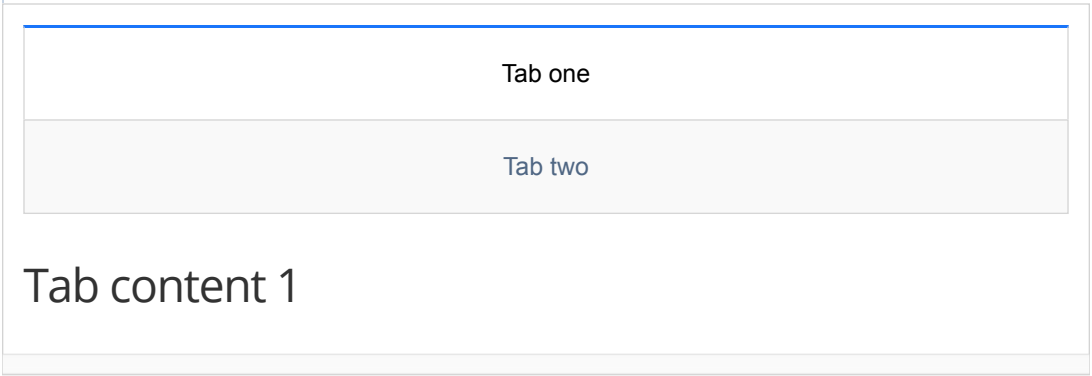
More Upload Examples and Reference

Tabs

The Tabs and Tab components can be used to create tabbed sections in your app.

```
import dash
import dash_html_components as html
import dash_core_components as dcc

from dash.dependencies import Input, Output

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

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')
```

```
])

@app.callback(Output('tabs-content', 'children'),
              [Input('tabs', 'value')])
def render_content(tab):
    if tab == 'tab-1':
        return html.Div([
            html.H3('Tab content 1')
        ])
    elif tab == 'tab-2':
        return html.Div([
            html.H3('Tab content 2')
        ])


if __name__ == '__main__':
    app.run_server(debug=True)
```



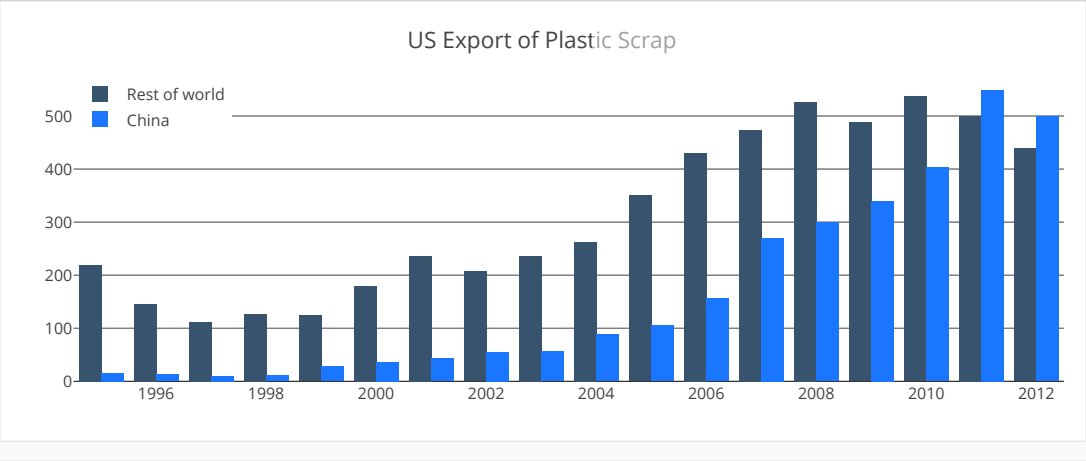
More Tabs Examples and Reference

Graphs

The `Graph` component shares the same syntax as the open-source `plotly.py` library. View the plotly.py docs to learn more.

```
import dash_core_components as dcc
import plotly.graph_objs as go

dcc.Graph(
    figure=go.Figure(
        data=[
```

```
            go.Bar(
                x=[1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003,
                   2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012]
                y=[219, 146, 112, 127, 124, 180, 236, 207, 236, 263,
                   350, 430, 474, 526, 488, 537, 500, 439],
                name='Rest of world',
                marker=go.bar.Marker(
                    color='rgb(55, 83, 109)'
                )
            ),
            go.Bar(
                x=[1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003,
                   2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012]
                y=[16, 13, 10, 11, 28, 37, 43, 55, 56, 88, 105, 156, 270
                   299, 340, 403, 549, 499],
                name='China',
                marker=go.bar.Marker(
                    color='rgb(26, 118, 255)'
                )
            )
        ],
        layout=go.Layout(
            title='US Export of Plastic Scrap',
            showlegend=True,
            legend=go.layout.Legend(
                x=0,
                y=1.0
            ),
            margin=go.layout.Margin(l=40, r=0, t=40, b=30)
        )
    ),
    style={'height': 300},
    id='my-graph'
)
```



View the plotly.py docs .

ConfirmDialog

The `dcc.ConfirmDialog` component send a dialog to the browser asking the user to confirm or cancel with a custom message.

```
import dash_core_components as dcc

confirm = dcc.ConfirmDialog(
    id='confirm',
    message='Danger danger! Are you sure you want to continue?'
)
```

More ConfirmDialog Examples and Reference

There is also a `dcc.ConfirmDialogProvider`, it will automatically wrap a child component to send a `dcc.ConfirmDialog` when clicked.

```
import dash_core_components as dcc
import dash_html_components as html
```

```
confirm = dcc.ConfirmDialogProvider(
    children=html.Button(
        'Click Me',
    ),
    id='danger-danger',
    message='Danger danger! Are you sure you want to continue?'
)
```

CLICK ME

More ConfirmDialogProvider Examples and Reference

Store

The store component can be used to keep data in the visitor's browser. The data is scoped to the user accessing the page.

Three types of storage (`storage_type` prop):

- `memory`: default, keep the data as long the page is not refreshed.
- `local`: keep the data until it is manually cleared.
- `session`: keep the data until the browser/tab closes.

For `local`/`session`, the data is serialized as json when stored.

```
import dash_core_components as dcc

store = dcc.Store(id='my-store', data={'my-data': 'data'})
```

The store must be used with callbacks

More Store Examples and Reference

Gauge

```
import dash_daq as daq

# Dash DAQ is a superpack of beautifully styled,
# premium Dash components for data acquisition apps.
# $840/developer/year. See dashdaq.io to purchase.

daq.Gauge(
  label="Gradient ranges",
  value=8,
  color={"gradient":true,"ranges":{"green":[0,6],"yellow":[6,8],"red":[8,10]}}
)
```

More Guage Examples and Reference

Logout Button

The logout button can be used to perform logout mechanism.

It's a simple form with a submit button, when the button is clicked, it will submit the form to the `logout_url` prop.

Please note that no authentication is performed in Dash by default and you have to implement the authentication yourself.

More Logout Button Examples and Reference

Dash HTML Components

Dash is a web application framework that provides pure Python abstraction around HTML, CSS, and JavaScript.

Instead of writing HTML or using an HTML templating engine, you compose your layout using Python structures with the `dash-html-components` library.

The source for this library is on GitHub: `plotly/dash-html-components`.

Here is an example of a simple HTML structure:

```
import dash_html_components as html

html.Div([
    html.H1('Hello Dash'),
    html.Div([
        html.P('Dash converts Python classes into HTML'),
        html.P('This conversion happens behind the scenes by Dash's Java
    ])
])
```

which gets converted (behind the scenes) into the following HTML in your web-app:

```
<div>
    <h1>Hello Dash</h1>
    <div>
        <p>Dash converts Python classes into HTML</p>
        <p>This conversion happens behind the scenes by Dash's JavaScrip
    </div>
</div>
```

If you're not comfortable with HTML, don't worry! You can get 95% of the way there with just a few elements and attributes. Dash's `core component library` also supports `Markdown`.

```
import dash_core_components as dcc

dcc.Markdown('''
#### Dash and Markdown

Dash supports [Markdown](http://commonmark.org/help).

Markdown is a simple way to write and format text.
```

```
It includes a syntax for things like **bold text** and *italics*,
[links](http://commonmark.org/help), inline `code` snippets, lists,
quotes, and more.
''')
```

Dash and Markdown

Dash supports [Markdown](#).

Markdown is a simple way to write and format text. It includes a syntax for things like **bold text** and italics, [links](#), inline `code` snippets, lists, quotes, and more.

If you're using HTML components, then you also have access to properties like `style`, `class`, and `id`. All of these attributes are available in the Python classes.

The HTML elements and Dash classes are mostly the same but there are a few key differences:

- The `style` property is a dictionary
- Properties in the `style` dictionary are camelCased
- The `class` key is renamed as `className`
- Style properties in pixel units can be supplied as just numbers without the `px` unit

Let's take a look at an example.

```
import dash_html_components as html

html.Div([
    html.Div('Example Div', style={'color': 'blue', 'fontSize': 14}),
    html.P('Example P', className='my-class', id='my-p-element')
], style={'marginBottom': 50, 'marginTop': 25})
```

That dash code will render this HTML markup:

```
<div style="margin-bottom: 50px; margin-top: 25px;">

    <div style="color: blue; font-size: 14px">
        Example Div
    </div>

    <p class="my-class", id="my-p-element">
        Example P
    </p>
```

```
</div>
```

Dash DataTable

Star

> New! Released on November 2, 2018
>
> Dash DataTable is an interactive table component designed for viewing, editing, and exploring large datasets.
>
> DataTable is rendered with standard, semantic HTML `<table/>` markup, which makes it accessible, responsive, and easy to style.
>
> This component was written from scratch in React.js specifically for the Dash community. Its API was designed to be ergonomic and its behavior is completely customizable through its properties.
>
> 7 months in the making, this is the most complex Dash component that Plotly has written, all from the ground-up using React and TypeScript. DataTable was designed with a featureset that allows that Dash users to create complex, spreadsheet driven applications with no compromises. We're excited to continue to work with users and companies that `invest in DataTable's future` .
>
> DataTable is in `Alpha` . This is more of a statement on the DataTable API rather than on its features. The table currently works beautifully and is already used in production at F500 companies. However, we expect to make a few more breaking changes to its API and behavior within the next couple of months. Once the community feels good about its API, we'll lock it down and we'll commit to reducing the frequency of breaking changes. Please subscribe to `dash-table#207` and the `CHANGELOG.md` to stay up-to-date with any breaking changes.
>
> So, check out DataTable and let us know what you think. Or even better, share your DataTable Dash apps on the `community forum` !
>
> -- chriddyp

Quickstart

```
pip install dash-table==3.1.11
```

```
import dash
import dash_table
import pandas as pd

df = pd.read_csv('https://raw.githubusercontent.com/plotly/datasets/mast

app = dash.Dash(__name__)

app.layout = dash_table.DataTable(
    id='table',
```

```
    columns=[{"name": i, "id": i} for i in df.columns],
    data=df.to_dict("rows"),
)

if __name__ == '__main__':
    app.run_server(debug=True)
```

State	Number of Solar Plants	Installed Capacity (MW)	Average MW
California	289	4395	
Arizona	48	1078	
Nevada	11	238	
New Mexico	33	261	
Colorado	20	118	
Texas	12	187	
North Carolina	148	669	
New York	13	53	

Dash DataTable User Guide

Part 1. Sizing

All about sizing the DataTable. Examples include:

- Setting the width and the height of the table
- Responsive table design
- Setting the widths of individual columns
- Handling long text
- Fixing rows and columns

Part 2. Styling

The style of the DataTable is highly customizable. This chapter includes examples for:

- Conditional formatting
- Displaying multiple rows of headers
- Highlighting rows, columns, and cells
- Styling the table as a list view
- Changing the colors (including a dark theme!)

The sizing API for the table has been particularly tricky for us to nail down, so be sure to read this chapter to understand the nuances, limitations, and the APIs that we're exploring.

Part 3. Sorting, Filtering, Selecting, and Paging

The DataTable is interactive. This chapter demonstrates the interactive features of the table and how to wire up these interations to Python callbacks. These actions include:

- Paging
- Selecting Rows
- Sorting Columns
- Filtering Data

Part 4. Sorting, Filtering, and Paging *with Python*

In Part 3, the paging, sorting, and filtering was done entirely clientside (in the browser). This means that you need to load all of the data into the table up-front. If your data is large, then this can be prohibitively slow.

In this chapter, you'll learn how to write your own filtering, sorting, and paging backends in Python with Dash. We'll do the data processing with Pandas but you could write your own routines with SQL or even generate the data on the fly!

Part 5. Editable Tables

The DataTable is editable. Like a spreadsheet, it can be used as an input for controlling models with a variable number of inputs.

This chapter includes recipes for:

- Determining which cell has changed
- Filtering out null values
- Adding or removing columns
- Adding or removing rows
- Ensuring that a minimum set of rows are visible
- Running Python computations on certain columns or cells

Part 6. Rendering Cells as Dropdowns

Cells can be rendered as editable Dropdowns. This is our first stake in bringing a full typing system to the table. Rendering cells as dropdowns introduces some complexity in the markup and so there are a few limitations that you should be aware of.

Part 7. Virtualization

Examples using DataTable virtualization.

Part 8. Filtering Syntax

An explanation and examples of filtering syntax for both frontend and backend filtering in the DataTable.

Part 9. Table Reference

The full list of Table properties and their settings.

Roadmap, Sponsorships, and Contact

Immediately, we're working on stability, virtualization, and a first-class data type system. Check out our roadmap project board to see what's coming next.

Many thanks to all of our customers who have sponsored the development of this table. Interested in steering the roadmap? Get in touch

Dash DAQ

Dash is a web application framework that provides pure Python abstraction around HTML, CSS, and JavaScript.

Dash DAQ comprises a robust set of controls that make it simpler to integrate data acquisition and controls into your Dash applications.

The source is on GitHub at `plotly/dash-daq`.

These docs are using version 0.0.6.

BooleanSwitch

A switch component that toggles between on and off.

```
import dash_daq as daq

daq.BooleanSwitch(
  id='my-daq-booleanswitch',
  on=True
)
```


More BooleanSwitch Examples and Reference

ColorPicker

A color picker.

```
import dash_daq as daq

daq.ColorPicker(
  id='my-daq-colorpicker',
  label="colorPicker",
  style={
    'float': 'center'
```

```
  }
)
```

More ColorPicker Examples and Reference

Gauge

A gauge component that points to some value between some range.

```
import dash_daq as daq

daq.Gauge(
  id='my-daq-gauge',
  min=0,
  max=10,
  value=6
)
```

More Gauge Examples and Reference

GraduatedBar

A graduated bar component that displays a value within some range as a percentage.

```
import dash_daq as daq

daq.GraduatedBar(
  id='my-daq-graduatedbar',
  min=0,
  max=100,
  value=42
)
```


More GraduatedBar Examples and Reference

Indicator

A boolean indicator LED.

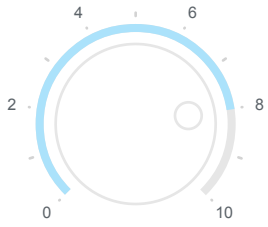
```
import dash_daq as daq

daq.Indicator(
  id='my-daq-indicator',
  value=True,
  color="#00cc96"
)
```

More Indicator Examples and Reference

Knob

A knob component that can be turned to a value between some range.

```
import dash_daq as daq

daq.Knob(
  id='my-daq-knob',
  min=0,
  max=10,
  value=8
)
```



More Knob Examples and Reference

LEDDisplay

A 7-segment LED display component.

```
import dash_daq as daq

daq.LEDDisplay(
  id='my-daq-leddisplay',
  value="3.14159"
)
```

More LEDDisplay Examples and Reference

NumericInput

A numeric input component that can be set to a value between some range.

```
import dash_daq as daq

daq.NumericInput(
  id='my-daq-numericinput',
  min=0,
  max=10,
  value=5
)
```

More NumericInput Examples and Reference

PowerButton

A power button component that can be turned on or off.

```
import dash_daq as daq

daq.PowerButton(
  id='my-daq-powerbutton',
  on=True
)
```


More PowerButton Examples and Reference

PrecisionInput

A numeric input component that converts an input value to the desired precision.

```
import dash_daq as daq

daq.PrecisionInput(
  id='my-daq-precisioninput',
  precision=4,
  value=299792458
)
```

2.998E8

More PrecisionInput Examples and Reference

StopButton

A stop button.

```
import dash_daq as daq

daq.StopButton(
  id='my-daq-stopbutton'
)
```

STOP

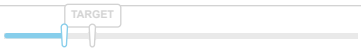
More StopButton Examples and Reference

Slider

A slider component with support for a target value.

```
import dash_daq as daq

daq.Slider(
```

```
  id='my-daq-slider',
  value=17,
  min=0,
  max=100,
  targets={"25": {"label": "TARGET"}}
)
```



More Slider Examples and Reference

Tank

A tank component that fills to a value between some range.

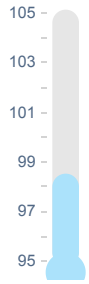
```
import dash_daq as daq

daq.Tank(
  id='my-daq-tank',
  min=0,
  max=10,
  value=5
)
```


More Tank Examples and Reference

Thermometer

A thermometer component that fills to a value between some range.

```
import dash_daq as daq

daq.Thermometer(
  id='my-daq-thermometer',
  min=95,
  max=105,
  value=98.6,
  style={
    'margin-bottom': '-5px'
  }
)
```



More Thermometer Examples and Reference

ToggleSwitch

A switch component that toggles between two values.

```
import dash_daq as daq

daq.ToggleSwitch(
  id='my-daq-toggleswitch'
)
```

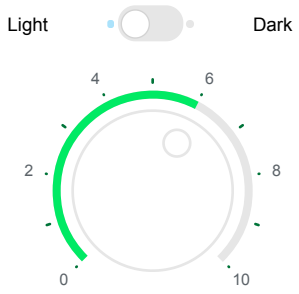

More ToggleSwitch Examples and Reference

DarkThemeProvider

A component placed at the root of the component tree to make all components match the dark theme.

```
import dash
from dash.dependencies import Input, Output
import dash_daq as daq
import dash_html_components as html


app = dash.Dash(__name__)

theme = {
    'dark': False,
    'detail': '#007439',
    'primary': '#00EA64',
    'secondary': '#6E6E6E'
}

app.layout = html.Div(id='dark-theme-provider-demo', children=[
    html.Br(),
    daq.ToggleSwitch(
        id='daq-light-dark-theme',
        label=['Light', 'Dark'],
        style={'width': '250px', 'margin': 'auto'},
        value=False
    ),
    html.Div(
        id='dark-theme-component-demo',
        children=[
            daq.DarkThemeProvider(theme=theme, children=
                                  daq.Knob(value=6))
        ],
        style={'display': 'block', 'margin-left': 'calc(50% - 110px)'}
    )
])


@app.callback(
    Output('dark-theme-component-demo', 'children'),
    [Input('daq-light-dark-theme', 'value')]
)
def turn_dark(dark_theme):
```

```
    if(dark_theme):
        theme.update(
            dark=True
        )
    else:
        theme.update(
            dark=False
        )
    return daq.DarkThemeProvider(theme=theme, children=
                                 daq.Knob(value=6))


@app.callback(
    Output('dark-theme-provider-demo', 'style'),
    [Input('daq-light-dark-theme', 'value')]
)
def change_bg(dark_theme):
    if(dark_theme):
        return {'background-color': '#303030', 'color': 'white'}
    else:
        return {'background-color': 'white', 'color': 'black'}


if __name__ == '__main__':
    app.run_server(debug=True)
```



More DarkThemeProvider Examples and Reference

Creating Your Own Components

React for Python Developers: a primer

Introduction

If you're a Dash developer, at some point or another you probably have thought about writing your own set of components for Dash. You might have even taken a peek at some of our source code, or taken the `dash-component-boilerplate` for a spin.

However, if you've never programmed in JavaScript and/or used React before, you might feel slightly confused. By the end of this guide, you should feel comfortable creating your own Dash component in React and JavaScript, even if you have never programmed in those languages before. Let us know how it went by commenting in this thread .

JavaScript

JavaScript is the language of the web - all modern browsers can run it, and most modern web pages use it to make their pages interactive. It is the de-facto standard of front end development, and has come a long way since its inception. Today, modern JavaScript has a rich set of features, designed to create a development experience perfectly suited for the web.

React

React is JavaScript library for building user interfaces, written and maintained by Facebook. It has been very popular over the last few years because it brings the power of reactive, declarative programming to the world of front end development.

React has made it easier to think about user interface code, and its programming model encourages code that's modular and reusable. It also has a huge, vibrant open-source community that has published all sorts of reusable UI components, from sliders to data tables, dropdowns to buttons.

It is important to realise that React is just JavaScript. React is not a language on its own, nor is it a domain-specific framework that takes years to master. It has a relatively small API, with just a few functions and paradigms to learn before you, too, can use it to write applications for the web. That being said, all unfamiliar technology will have a learning curve, but with practice and patience you will master it!

Dash uses React under the hood to render the user interface you see when you load a web page created with Dash. Because React allows you to write your user interface in encapsulated components that manage their own state, it is easy to split up parts of code for Dash too. At the end of this tutorial, you will see that Dash components and React components map one to one!

For now, the important thing to know is that Dash components are mostly simple wrappers around existing React components. This means the entire React ecosystem is potentially usable in a Dash application!

Installing everything you need

Let's start by setting up our JavaScript development environment. We will use Node.js, NPM, and our `dash-component-boilerplate` to write our first React application. Node.js is a JavaScript runtime, which allows you to run JavaScript code outside of the browser. Just like you would run `python my-code.py` to run Python code in a terminal, you'd run `node my-code.js` to run JavaScript code in a terminal.

Node comes in very handy when developing, even when you intend to run the code in the browser.

NPM

NPM is the "Node Package Manager" and it is used to install packages and run scripts. Besides being a package manager (like `pip` for Python), `npm` also allows you to run scripts and perform tasks, such as creating a project for you (`npm init`), starting up a project (`npm start`), or firing custom scripts (`npm run custom-script`). These scripts are defined in a `package.json` file, which every project that uses `npm` has.

The `package.json` file holds your `requirements` and `devRequirements`, which can be installed using `npm install`, the same way `pip` has a `requirements.txt` option you can use in `pip install -r requirements.txt`.

`package.json` also holds a `scripts` section where custom scripts can be defined. It is usually a good idea to check out a new project's `package.json` file to see which scripts the project uses.

If you go to the dash-component-boilerplate repo, you'll find instructions for setting up some React boilerplate code. This code will help you quickly set up a React development environment, complete with the necessary scripts for building our React component for Dash.

These scripts will use a variety of technologies (e.g. Babel, Webpack, and more) to compile our code into a web-ready package.

- To install Node.js, go to the Node.js website to download the latest version. We recommend installing the LTS version.
- Node.js will automatically install the Node Package Manager `npm` on your machine
- Verify that node is installed by running: `node -v`
- Verify that npm is installed by running: `npm -v`

Python

You will need python to generate your components classes to work with Dash.

Download python on the official website or through your os distribution package manager.

- https://www.python.org/
- `apt-get install python`/`yum install python`

Virtual environments

It is best to use virtual environments when working on projects, we recommend creating a fresh virtual environment for each project you have so they can have specific requirements and remain isolated from your main python environment.

In Python 2 you have to use `virtualenv`:

`pip install virtualenv` Then you create a venv with the command `virtualenv venv`, it will create a folder `venv` in the current directory with your new environment.

In Python 3 you can use the builtin module `venv`:

`python -m venv venv`

Cookiecutter boilerplate

Now that we have Node.js and Python up and running, we can generate a dash component project using the `cookiecutter dash-component-boilerplate` .

The boilerplate is built using `cookiecutter` , a project template renderer made with jinja2. This allows users to create a project with custom values formatted for the project.

Install cookiecutter:

`pip install cookiecutter`

Generate a new dash component project

- Run the cookiecutter: `cookiecutter https://github.com/plotly/dash-component-boilerplate.git`
- Answer the questions about the project:
  - `project_name`: A display name for the project, can contain spaces and uppercase letters, for example `Example Component`.
  - `project_shortname`: A variable derived from project_name without spaces and all lowercase letters.
  - `component_name`: Derived from project without spaces and `-`, it will be the default component class name and as such should be PascalCase for naming.
  - `author_name`/`author_email`: Your name/email to be included in `package.json` and `setup.py`.
  - `description`: A short description for the project.

- `license`: Choose a license from the list.
- `publish_on_npm`: Set to false if you don't want to publish on npm, your component will always be loaded locally.
- `install_dependencies`: Install the npm packages and `requirements.txt` and build the initial component so it's ready for a spin.

Project structure

```
- project_shortname           # Root of the project
  - project_shortname         # The python package, output folder for the bu
  - src                       # The javascript source directory for the comp
      - lib
          - components        # Where to put the react component classes.
      - demo
          - App.js            # A sample react demo, only use for quick test
          - index.js          # A reactDOM entry point for the demo App.
      - index.js              # The index for the components exported by the
  - tests                     #
      - requirements.txt      # python requirements for testing.
      - test_usage.py         # Runs `usage.py` as a pytest integration test
  - package.json              # npm package info and build commands.
  - setup.py                  # Python package info
  - requirements.txt          # Python requirements for building the compone
  - usage.py                  # Sample Python dash app to run the custom com
  - webpack.config.js         # The webpack configs used to generate the bun
  - webpack.serve.config.js   # webpack configs to run the demo.
  - MANIFEST.in               # Contains a list of files to include in the I
  - LICENSE                   # License info
```

Build the project

- `npm run build:js` generate the production bundle `project_shortname.min.js`
- `npm run build:js-dev` generate the development bundle `project_shortname.dev.js`, use with `app.run_server(debug=True)`
- `npm run build:py` generate the python classes files for the components.
- `npm run build:all` generate both bundles and the languages classes files.

Release the project

If you choose `publish_on_npm`, you will have to publish on npm first. Publishing your component on npm will rebuild the bundles, do not rebuild between that and publishing on pypi as the bundle will be different when serving locally and externally.

Publish on npm

`npm publish` If you have 2 factor enabled, you will need to enter the otp argument.

Publish on pypi

`python setup.py sdist` will build the python tarball package locally in the dist folder.

You can then upload the package using twine (`pip install twine`):

`twine upload dist/*`

Quick intro to React

Now, let's go ahead and see what the code for our new React application looks like. In your favorite code editor, open the `src/lib/components/ExampleComponent.react.js` file (supposing you named your project `example_component`). This is our first React component!

In React, user interfaces are made of "components," and by convention there will usually be one main component per file. This project imports the `ExampleComponent` in `src/demo/App.js`, the demo application.

The demo application `src/demo/App.js` is what you see after you run `npm run start`. To see how `src/demo/App.js` and `ExampleComponent.react.js` work, try adding `<h1>Hello, Dash!</h1>` inside the `render()` function of either of the files.

When you save the files, your browser should automatically refresh and you should see your change.

JSX

The `<h1>` and `<div>` tags you see look exactly like HTML tags, however, they are slightly different. These tags are what is called JSX - a syntax extension to JavaScript. JSX was developed by the React team to make easy, inline, HTML-like markup in JavaScript components.

There are a few key differences between JSX tags and HTML tags:

- The `class` keyword is renamed `className` (similarly as in Dash)
- In HTML, we write inline styles with strings like `<h1 style="color: hotpink; font-size: 12px">Hello Dash</h1>`. In JSX (as in Dash), we use objects with camelCased properties: `<h1 style={{"color": "hotpink", "fontSize": "12px"}}>Hello Dash<h1>`
- In JSX, we can embed variables in our markup. To embed a variable in the markup, wrap the variable in `{}`. For example:

```
render() {
    var myText = 'Hello Dash!';
    return (
          <h1>{myText}</h1>
    );
}
```

- In addition to the HTML tags like `<h1>` and `<div>`, we can also reference other React classes. For example, in our `src/demo/App.js`, we render the `ExampleComponent` component by referencing it as `<ExampleComponent>`.

A quick primer on the JavaScript language

Variable declaration

In JavaScript, we have to declare our variables with `let` or `const`. `const` is used when the variable shouldn't change, `let` is used elsewhere:

```
const color = 'blue';
let someText = 'Hello World';
let myText;
myText = 'Hello Dash';
```

Comments

Single line comments are prefixed with `//`. Multi-line comments are wrapped in `/* */`

```
/*
* This is a multi-line comment
* By convention, we use a `*` on each line, but it's
* not strictly necessary.
*/
const color = 'blue'; // This is a single line comment
```

Strings

Strings are defined the same way in JavaScript: single or double quotes:

```
const someString = 'Hello Dash';
const anotherString = "Hello Dash";
```

Instead of Python's `format`, JavaScript allows you to embed variables directly into strings by wrapping the variable in `${}` and wrapping the string in backticks:

```
const name = 'Dash';
const someString = `Hello ${name}`;
```

Dictionaries

In Python, we use dictionaries for key-value pairs. In JavaScript, we use "objects" and they are instantiated and accessed very similarly:

```
const myObject = {"color": "blue", "size": 20};
myObject['color']; // is blue
myObject.color;  // another way to access the color variable
```

In Python, the keys of a dictionary can be any type. But in JavaScript, the keys can only be strings. JavaScript allows you to omit the quotes around the strings and we frequently do:

```
const myObject = {color: "blue"}; // notice how there are no strings around
```

So if you want to set a dynamic key in an object, you have to wrap it in square brackets:

```
const styleProperty = "color";
const myObject = {[styleProperty]: "blue"};
myObject.color;
```

Lists

In JavaScript, lists are called "arrays" and they're instantiated and accessed the same way:

```
const myList = ["Hello", "Dash", "!"];
myList[0];  // Hello
myList[1];  // Dash
myList[myList.length - 1]; // -1 references aren't allowed in JavaScript
```

Semicolons

In JavaScript, the convention is to end each line in a semicolon. It's not strictly necessary anymore, but it's still the convention.

Print Statements, Errors, and the Console

In JavaScript, we use `console.log` to print statements into the "console":

```
console.log("Hello Dash");
```

Since JavaScript runs in the web browser, we won't see these statements in our terminal like we would in Python. Instead, we'll see these statements in the browser's "dev tools console".

To access the console:

1. Right click on the web page
2. Click "Inspect Element"
3. Click on the "Console" tab.

To see for yourself, add a `console.log` statement inside the `render` method of `ExampleComponent`, refresh the page, and then inspect your browser's console.

Like Python, error messages and exceptions will also appear inside this console.

If, For, While

`if`

```
if (color === 'red') {
  console.log("the color is red");
} else if (color === 'blue') {
  console.log("the color is blue");
} else {
  console.log("the color is something else");
}
```

`for`

```
for (let i = 0; i < 10; i++) {
  console.log(i);
}
```

`while`

```
let i = 0;
while (i < 10) {
  i += 1;
}
```

Functions

In JavaScript, you'll see functions defined in two ways:

The new style way:

```
const add = (a, b) => {
  // The inside of the function
  const c = a + b;
  return c;
}

console.log(add(4, 6)); // 10
```

The traditional way:

```
function (a, b) {
  // The inside of the function
  const c = a + b;
```

```
  return c;
}

console.log(add(4, 6)); // 10
```

Classes

> Heads up! Classes, among other features, are new language features in JavaScript. Technically, they're part of a new version of JavaScript called ES6. When we build our JavaScript code, a tool called Babel will convert these new language features into simpler JavaScript that older browsers like IE11 can understand.

JavaScript classes are very similar to Python classes. For example, this Python class:

```
class MyComponent(Component):
    def __init__(self, a):
        self.a = a;
        super().__init__(a)

    def render(this):
        return self.a;
```

would be written in JavaScript as:

```
class MyComponent extends Component {
    init() {
        super();
        this.a = a;
    }

    render() {
        return this.a;
    }
}
```

Importing and Exporting

In Python, we can import any variable from any file. In JavaScript, we have to explicitly specify which variables we want to make "importable" by "exporting" the variables.

If we only want to export a single variable, we'll write `export default`:

`some_file.js`

```
const text = 'hello world';
export default text;
```

`another_file.js`

```
import text from './some_file.js';
```

If we want to export multiple variables, we'll just write `export`:

`some_file.js`

```
const text = 'hello world';
const color = 'blue';
const size = '12px';

export text;
export color;
```

`another_file.js`

```
import {text, color} from './some_file.js';
/*
* note that we can't import size
* because we didn't export it
*/
```

The Standard Library and Ramda

Unlike Python, JavaScript's "standard library" is pretty small. At Plotly, we use the 3rd party library `ramda` for many of our common data manipulations.

Virtual DOM

If we look at the `App` component again, we see that it is `export`ed at the bottom. If you open up the `src/demo/index.js` file, you can see that it's imported there so that it can be used in a call to `ReactDOM.render()`.

ReactDOM.render() is what actually renders our React code into HTML on the web page. This ReactDOM.render() method is only called here, and only called once.

Classes

We see here in our `App` component that it is defined as a `class` which `extends` from the `Component` class of React. This provides some methods to us, for example the `render()` method we're using here. `render()` is the method that is called by the component that is rendering it. In our case, `render()` is called by the `ReactDOM.render()` call in `index.js`.

Notice how the `<App />` is called in the `ReactDOM.render()` method: our `App` component is used as a JSX tag!

Other Methods on React.Component

Other methods provided by React are mostly related to component state management. Lifecycle hooks like `shouldComponentUpdate` and `componentDidMount` allow you to better specify

when and how a component should update.

For these methods, please refer to the React documentation.

Our very own React component

Creating a boilerplate component

Now, let's create our very own component. Create a file named `TextInput.react.js` inside the `src/lib/components/` folder. In `TextInput.react.js` write:

```
import React, { Component } from 'react';

class TextInput extends Component {
  // here we'll define everything we need our TextInput component to have
}
```

Next, we'll write a `constructor` method on our component. A class constructor in Python is usually defined as `def __init__()` on a class, but in JavaScript we use the `constructor()` syntax.

In the constructor, we call the `super()` method on our component's props (more on props later), and set some `state`. It will look like this:

```
import React, { Component } from 'react';

class TextInput extends Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'default'
    }
  }
}
```

`props` are a component's properties. They are passed down from a component's parent, and are available as the `props` attribute. Calling `super()` on `props` in the constructor makes our props available in the component as `this.props`. The `this` keyword in JavaScript is Python's `self`.

We'll show you how to pass down `props` a bit later on.

Defining the `render` method

Next, let's define our `render()` method for our new component. In React, we are declaring UI components, and React calls the `render()` method when it wants to render those components.

A component's `render` method can return a basic string, for example `return "Hello, World!"`. When this component is used somewhere, it's `render()` method is called and "Hello,

World!" will be displayed on the page.

Likewise, you can return a React element (specified using JSX) and React will render that element.

Exporting and importing a component

We'll also go ahead and `export` our component as the `default`. This means whenever we're trying to `import` something from this file, and we don't specify a name, we'll get the `default` export.

```
import React, { Component } from 'react';

class TextInput extends Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'default'
    }
  }
  render() {
    return <input />
  }
}

export default TextInput;
```

Now let's `import` that component and use it in our `App` component!

Add the `import TextInput from './TextInput';` line to the top of `App.js`, and somewhere in the return of our `render()` method, use our newly created `<TextInput />` component.

Tada!

We've got a text input.

In order to use our component from Python, you also need to import and export the component in `src/lib/index.js` (see how it is done for ExampleComponent).

Updating state with the `setState()` method

However, this input doesn't really do much - it's not connected to anything, nor does it save what you type in. Let's change our `render()` method of `TextInput` to set the HTML `value` attribute on our `<input />` tag, so it looks like this: `<input value='dash' />`. Save it, and we should now see that the value of our <input> tag is set to 'dash'!

We can also change our value to be that which is defined in our `state` object, so `<input value={this.state.value} />`. The `{}` syntax in JSX means that we want to write inline

JavaScript in our JSX, so our `this.state.value` statement can be computed.

Great! Now our input says 'default'! Unfortunately, our input is still not very useful, because we can't change our input's value, try as we might.

It may seem odd to you that we can't type anything into the `<input/>` box. However, this is consistent with the React model: in our render method, we are telling React to render an input with a particular value set. React will faithfully render this input with that value no matter what, even if we try typing in it.

In order to have the input update when we type, we have to make sure that the `value` variable is updated with whatever we're typing in the input. To do that, we can listen for changes to the input and update our state accordingly:

```
import React, { Component } from 'react';

class TextInput extends Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'default'
    }
  }
  handleInputChange = (e) => {
    // get the value from the DOM node
    const newValue = e.target.value;
    // update the state!
    this.setState({
      value: newValue
    })
  }
  render() {
    return <input value={this.state.value} onChange={this.handleInputChange
  }
}

export default TextInput;
```

Here, we wrote a method which we set on our input's `onChange` attribute, which will fire every time we type into the input. This method has a parameter (named `e` for event) on which certain attributes are set: `target.value` is what we need. This is how the HTML DOM works - for more information check out these docs .

Next, we use a method called `setState()` that's provided by `React.Component`. This method will handle updates to our `state` object. This method is really special. It'll do two things:

1. It'll merge the object that you provide with whatever was currently in `this.state`.

2. Then, it'll rerender the component. That is, it'll tell React to call the components render method again with the new data set in `this.state`.

See how this now allows you to type in our input component? We can also display our state by writing our `render()` method something like:

```
render() {
  return <div>
    <input value={this.state.value} onChange={this.handleInputChange} />
    <p>{this.state.value}</p> // here we're displaying our state
  </div>
}
```

Notice that we're not allowed to return multiple elements from `render()`, but an element with children is totally fine.

Component props

We can also pass along properties to our components, via the aforementioned `props`. This works the same as assigning attributes on a component, as we'll demonstrate by adding a `label` prop to our `TextInput` component!

Let's edit our call to `<TextInput />` in `App.js` to say `<TextInput label='dash-input' />`. This means we now have a `prop` called `label` available on our `TextInput` component. In `TextInput`, we can reference this via `this.props`.

Let's extend our `render()` method further so it renders our `label` prop:

```
render() {
  return <div>
    <label>{this.props.label}</label>
    <input value={this.state.value} onChange={this.handleInputChange} />
    <p>{this.state.value}</p>
  </div>
}
```

Props always flow down, but you can set a method as a prop too, so that a child can call a method of a parent. For more information, please refer to the React docs.

These are just the basics of React, if you want to know more, the React docs are a great place to start!

Using your React components in Dash

We can use most, if not all, React components in Dash! Dash uses React under the hood, specifically in the `dash-renderer`. The `dash-renderer` is basically just a React app that

renders the layout defined in your Dash app as `app.layout`. It is also responsible for assigning the callbacks you write in Dash to the proper components, and keeping everything up-to-date.

Control the state in the parent

Let's modify our previous example to control `state` in the parent `App` component instead of in the `TextInput` component. Let's start by moving the `value` in `state` up to the parent component.

In `src/demo/App.js`, add state and pass the value into the component:

```
component App extends Component {
  constructor() {
    super(props)

    this.state = {
        value: 'dash'
    };
  }

    render() {
      return <TextInput label={'Dash'} value={this.state.value}/>
    }
}
```

In `src/lib/components/TextInput.react.js`, use the `value` prop instead of the state:

```
component TextInput extends Component {
  constructor() {
    super(props)
  }

    render() {
      return (
          <div>
            <label>{this.props.label}</label>
            <input value={this.props.value}/>
            <p>{this.props.value}</p>
          </div>
        )
    }
}
```

Now, as before, the `<input/>` won't actually update when you type into it. We need to update the component's `value` property as we type. To do this, we'll define a function in our parent

component that will update the parent's component state, and we'll pass that function down into our component. We'll call this function `setProps`:

```
component App extends Component {
  constructor() {
    super(props)

    this.state = {
        value: 'dash'
    };
  }

    setProps(newProps) {
      this.setState(newProps);
    }

    render() {
      return (
        <TextInput
          label={'Dash'}
          value={this.state.inputValue}
          setProps={this.setProps}
        />
    }
}
```

and in `TextInput`, we'll call this function when the `value` of our `input` changes. That is, when we type into the input:

```
component TextInput extends Component {
  constructor() {
    super(props)

  }

    handleInputChange = (e) => {
      const newValue = e.target.value;
      this.props.setProps({value: newValue});
    }

    render() {
      return (
          <div>
```

```
            <label>{this.props.label}</label>
            <input value={this.props.value} onChange={this.handleInputChang
            <p>{this.props.value}</p>
          </div>
        )
    }
}
```

To review, this is what happens when we type into our `<input>`:

1. The `handleInputChange` is called with whatever value we typed into the `<input/>`

2. `this.props.setProps` is called, which in turn calls the `setState` property of the `App` component.

3. `this.setState` in `App` is called. This updates the `this.state` of `App` and implicitly calls the `render` method of `App`.

4. When `App.render` is called, it calls `TextInput.render` with the new properties, rerendering the `<input/>` with the new `value`.

In Dash apps, the `dash-renderer` project is very similar to `App.js`. It contains all of the "state" of the application and it passes those properties into the individual components. When a component's properties change through user interaction (e.g. typing into an `<input/>` or hovering on a graph), the component needs to call `setProps` with the new values of the property. Dash's frontend (`dash-renderer`) will then rerender the component with the new property and make the necessary API calls to Dash's Python server callbacks.

Handling the case when `setProps` isn't defined

In Dash, `setProps` is only defined if the particular component is referenced in an `@app.callback`. If the component isn't referenced in a callback, then Dash's frontend will not pass in the `setProps` property and it will be undefined.

> As an aside, why does Dash do that? In some cases, it could be computationally expensive to determine the new properties. In these cases, Dash allows component authors to skip doing these computations if the Dash app author doesn't actually need the properties. That is, if the component isn't in any `@app.callback`, then it doesn't need to go through the "effort" to compute it's new properties and inform Dash.

In most cases, this is a non-issue. After all, why would you render an `Input` on the page if you didn't want to use it as an `@app.callback`? However, sometimes we still want to be able to interact with the component, even if it isn't connected to Dash's backend. In this case, we'll manage our state locally or through the parent. That is:

1. If `setProps` is defined, then the component will call this function when its properties change and Dash will faithfully rerender the component with the new properties that it passed up.

2. If `setProps` isn't defined, then the component isn't "connected" to Dash's backend through a callback and it will manage its state locally.

Here's an example with our `TextInput` component:

```
component TextInput extends Component {
  constructor() {
    super(props)
    this.state = props;
  }

    handleInputChange = (e) => {
      const newValue = e.target.value;
      this.props.setProps({value: newValue});
    }

    render() {
      let value;
      if (this.props.setProps) {
        value = this.props.value;
      } else {
        value = this.state.value;
      }
      return (
          <div>
            <label>{this.props.label}</label>
            <input value={value} onChange={this.handleInputChange}/>
            <p>{value}</p>
          </div>
        )
    }
}
```

Annotate your function with `propTypes`

The final step in authoring your Dash component is to describe which properties are available.

This is done through React's `propTypes`. At the end of your file, write:

```
TextInput.propTypes = {
  ...
}
```

(You should fill that in with the actual ones.)

You must include `propTypes` because they describe the input properties of the component, their types, and whether or not they are required. Dash's React-to-Python toolchain looks for these `propTypes` in order to automatically generate the Dash component Python classes.

A few notes:

- The comments above each property are translated directly into the Python component's docstrings. For example, compare the output of `>>> help(dcc.Dropdown)` with that component's propTypes.
- The `id` property is required in all Dash components.
- The list of available types are available here.
- In the future, we will use these `propTypes` to provide validation in Python. That is, if you specify that a property is a `PropTypes.string`, then Dash's Python code will throw an error if you supply something else. Track our progress in this issue: 264.

React as a peer dependency

React and reactDOM are included as peer dependencies, your editor may warn you that react is not installed. You can safely ignore those warnings as they are served by the `dash-renderer` and they don't need to be bundled with your components.

Build your component in Python

Now that you have your React component, you can build it and import it into your Dash program. View instructions on how to build the component in the boilerplate repo.

In this tutorial, we rebuilt the `ExampleComponent` that was provided in the boilerplate. So, the Python component code in `usage.py` should look familar - the properties and behaviour of `ExampleComponent` are exactly the same as our `TextInput`.

Feedback, questions? Let us know in on GitHub in our dedicated feedback issue.

Here are some helpful links:

- The React docs
- dash-core-components
- dash-component-boilerplate

Writing your own components

One of the really cool things about dash is that it is built on top of React.js, a JavaScript library for building web components.

The React community is huge. Thousands of components have been built and released with open source licenses. For example, here are just some of the slider components and table components that have been published by the React community, any of which could be adapted into a Dash Component.

Creating a Component

To create a Dash component, fork our sample component repository and follow the instructions in the README.md: https://github.com/plotly/dash-component-boilerplate

If you are just getting started with React.js as a Python programmer, please check out our essay "React for Python Devs".

How Are Components Converted From React.js to Python?

Dash provides a framework that converts React components (written in JavaScript) into Python classes that are compatible with the Dash ecosystem.

On a high level, this is how that works:

- Components in dash are serialized as JSON. To write a dash-compatible component, all of the props shared between the Python code and the React code must be serializable as JSON. Numbers, Strings, Booleans, or Arrays or Objects containing Numbers, Strings, Booleans. For example, JavaScript functions are not valid input arguments. In fact, if you try to add a function as a prop to your Dash component, you will see that the generated Python code for your component will not include that prop as part of your component's accepted list of props. (It's not going to be listed in the `Keyword arguments` enumeration or in the `self._prop_names` array of the generated Python file for your component).

- By annotating components with React docstrings (not required but helpful and encouraged), Dash extracts the information about the component's name, properties, and description through React Docgen. This is exported as a JSON file (metadata.json).

- At build time, Dash reads this JSON file and dynamically creates Python classes that subclass a core Dash component. These classes include argument validation, Python docstrings, types, and a basic set of methods. These classes are generated entirely automatically. A JavaScript developer does not need to write any Python in order to generate a component that can be used in the Dash ecosystem.

- You will find all of the auto-generated files from the build process in the folder named after your component. When you create your Python package, by default any non-Python files won't be included in the actual package. To include these files in the package, you must list them explicitly in `MANIFEST.in`. That is, `MANIFEST.in` needs to contain each JavaScript, JSON, and CSS file that you have included in your `my_dash_component/` folder. In the `dash-component-boilerplate` repository, you can see that all the javascript for your React component is included in the build.js file.
- The Dash app will crawl through the app's `layout` property and check which component packages are included in the layout and it will extract that component's necessary JavaScript or CSS bundles. Dash will serve these bundles to Dash's front-end. These JavaScript bundles are used to render the components.
- Dash's `layout` is serialized as JSON and served to Dash's front-end. This `layout` is recursively rendered with these JavaScript bundles and React.

Encapuslating D3.js Charts as Python Dash Components

D3.js is a flexible library for rendering and animating SVG in the web browser. It's approach toward rendering content in the DOM is quite different than React.js, the user interface library that Dash components use.

By popular demand, we've created a set of tutorials to help you make high quality Dash components with D3.js. Beyond D3, these tutorials should also help you better understand how to integrate 3rd-party libraries into your custom React and Dash components.

- Tutorial 1 - React + D3.js, Guiding Principles
- Tutorial 2 - Real World Example: Dash Sunburst
- Tutorial 3 - Real World Example: Dash Network

Questions or feedback? Let us know in our Dash + D3 Feedback Thread

> Note: Before you dig in too deep into D3.js, we recommend checking out `plotly.js`, our flagship open source charting library. Since 2012, we've worked hard to abstract away the complexity of data visualization through a standard, declarative interface for over 30 chart types. Under the hood, `plotly.js` uses D3.js and WebGL. `dash_core_components.Graph` is the official Dash interface to `plotly.js`.

Beyond the Basics

Performance

This chapter contains several recommendations for improving the performance of your dash apps.

The main performance limitation of dash apps is likely the callbacks in the application code itself. If you can speed up your callbacks, your app will feel snappier.

Memoization

Since Dash's callbacks are functional in nature (they don't contain any state), it's easy to add memoization caching. Memoization stores the results of a function after it is called and re-uses the result if the function is called with the same arguments.

To better understand how memoization works, let's start with a simple example.

```
import time
import functools32

@functools32.lru_cache(maxsize=32)
def slow_function(input):
    time.sleep(10)
    return 'Input was {}'.format(input)
```

Calling `slow_function('test')` the first time will take 10 seconds. Calling it a second time with the same argument will take almost no time since the previously computed result was saved in memory and reused.

Dash apps are frequently deployed across multiple processes or threads. In these cases, each process or thread contains its own memory, it doesn't share memory across instances. This means that if we were to use `lru_cache`, our cached results might not be shared across sessions.

Instead, we can use the Flask-Caching library which saves the results in a shared memory database like Redis or as a file on your filesystem. Flask-Caching also has other nice features like time-based expiry. Time-based expiry is helpful if you want to update your data (clear your cache) every hour or every day.

Here is an example of `Flask-Caching` with Redis:

```
import datetime
import os

import dash
import dash_core_components as dcc
```

```
import dash_html_components as html
from dash.dependencies import Input, Output
from flask_caching import Cache

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
cache = Cache(app.server, config={
    # try 'filesystem' if you don't want to setup redis
    'CACHE_TYPE': 'redis',
    'CACHE_REDIS_URL': os.environ.get('REDIS_URL', '')
})
app.config.suppress_callback_exceptions = True

timeout = 20
app.layout = html.Div([
    html.Div(id='flask-cache-memoized-children'),
    dcc.RadioItems(
        id='flask-cache-memoized-dropdown',
        options=[
            {'label': 'Option {}'.format(i), 'value': 'Option {}'.format
            for i in range(1, 4)
        ],
        value='Option 1'
    ),
    html.Div('Results are cached for {} seconds'.format(timeout))
])


@app.callback(
    Output('flask-cache-memoized-children', 'children'),
    [Input('flask-cache-memoized-dropdown', 'value')])
@cache.memoize(timeout=timeout)  # in seconds
def render(value):
    return 'Selected "{}" at "{}"'.format(
        value, datetime.datetime.now().strftime('%H:%M:%S')
    )


if __name__ == '__main__':
    app.run_server(debug=True)
```

Here is an example that **caches a dataset** instead of a callback. It uses the FileSystem cache, saving the cached results to the filesystem.

This approach works well if there is one dataset that is used to update several callbacks.

```
import datetime as dt
import os
import time

import dash
import dash_core_components as dcc
import dash_html_components as html
import numpy as np
import pandas as pd
from dash.dependencies import Input, Output
from flask_caching import Cache

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
cache = Cache(app.server, config={
    'CACHE_TYPE': 'filesystem',
    'CACHE_DIR': 'cache-directory'
})

TIMEOUT = 60

@cache.memoize(timeout=TIMEOUT)
def query_data():
    # This could be an expensive data querying step
    df =  pd.DataFrame(
        np.random.randint(0,100,size=(100, 4)),
        columns=list('ABCD')
    )
    now = dt.datetime.now()
    df['time'] = [now - dt.timedelta(seconds=5*i) for i in range(100)]
    return df.to_json(date_format='iso', orient='split')


def dataframe():
    return pd.read_json(query_data(), orient='split')
```

```
app.layout = html.Div([
    html.Div('Data was updated within the last {} seconds'.format(TIMEOU
    dcc.Dropdown(
        id='live-dropdown',
        value='A',
        options=[{'label': i, 'value': i} for i in dataframe().columns]
    ),
    dcc.Graph(id='live-graph')
])


@app.callback(Output('live-graph', 'figure'),
              [Input('live-dropdown', 'value')])
def update_live_graph(value):
    df = dataframe()
    now = dt.datetime.now()
    return {
        'data': [{
            'x': df['time'],
            'y': df[value],
            'line': {
                'width': 1,
                'color': '#0074D9',
                'shape': 'spline'
            }
        }],
        'layout': {
            # display the current position of now
            # this line will be between 0 and 60 seconds
            # away from the last datapoint
            'shapes': [{
                'type': 'line',
                'xref': 'x', 'x0': now, 'x1': now,
                'yref': 'paper', 'y0': 0, 'y1': 1,
                'line': {'color': 'darkgrey', 'width': 1}
            }],
            'annotations': [{
                'showarrow': False,
                'xref': 'x', 'x': now, 'xanchor': 'right',
                'yref': 'paper', 'y': 0.95, 'yanchor': 'top',
                'text': 'Current time ({}:{}:{})'.format(
```

```
                    now.hour, now.minute, now.second),
                'bgcolor': 'rgba(255, 255, 255, 0.8)'
            }],
            # aesthetic options
            'margin': {'l': 40, 'b': 40, 'r': 20, 't': 10},
            'xaxis': {'showgrid': False, 'zeroline': False},
            'yaxis': {'showgrid': False, 'zeroline': False}
        }
    }


if __name__ == '__main__':
    app.run_server(debug=True)
```

Graphs

`Plotly.js` is pretty fast out of the box.

Most plotly charts are rendered with SVG. This provides crisp rendering, publication-quality image export, and wide browser support. Unfortunately, rendering graphics in SVG can be slow for large datasets (like those with more than 15k points). To overcome this limitation, plotly.js has WebGL alternatives to some chart types. WebGL uses the GPU to render graphics.

The high performance, WebGL alternatives include:

- `scattergl`: A webgl implementation of the `scatter` chart type. Examples, reference
- `pointcloud`: A lightweight version of `scattergl` with limited customizability but even faster rendering. Reference
- `heatmapgl`: A webgl implementation of the `heatmap` chart type. Reference

Currently, dash redraws the entire graph on update using the `plotly.js` `newPlot` call. The performance of updating a chart could be improved considerably by introducing `restyle` calls into this logic. If you or your company would like to sponsor this work, get in touch.

Sponsoring Performance Enhancements

There are many other ways that we can improve the performance of dash apps, like caching front-end requests, pre-filling the cache, improving plotly.js's webgl capabilities, reducing JavaScript bundle sizes, and more.

Historically, many of these performance related features have been funded through company sponsorship. If you or your company would like to sponsor these types of enhancements, please get in touch, we'd love to help.

Live Updating Components

The `dash_core_components.Interval` component

Components in Dash usually update through user interaction: selecting a dropdown, dragging a slider, hovering over points.

If you're building an application for monitoring, you may want to update components in your application every few seconds or minutes.

The `dash_core_components.Interval` element allows you to update components on a predefined interval. The `n_intervals` property is an integer that is automatically incremented every time `interval` milliseconds pass. You can listen to this variable inside your app's `callback` to fire the callback on a predefined interval.

This example pulls data from live satellite feeds and updates the graph and the text every second.

```
import datetime

import dash
import dash_core_components as dcc
import dash_html_components as html
import plotly
from dash.dependencies import Input, Output

# pip install pyorbital
from pyorbital.orbital import Orbital
satellite = Orbital('TERRA')

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
app.layout = html.Div(
    html.Div([
        html.H4('TERRA Satellite Live Feed'),
        html.Div(id='live-update-text'),
        dcc.Graph(id='live-update-graph'),
        dcc.Interval(
            id='interval-component',
            interval=1*1000, # in milliseconds
```

```
            n_intervals=0
        )
    ])
)


@app.callback(Output('live-update-text', 'children'),
              [Input('interval-component', 'n_intervals')])
def update_metrics(n):
    lon, lat, alt = satellite.get_lonlatalt(datetime.datetime.now())
    style = {'padding': '5px', 'fontSize': '16px'}
    return [
        html.Span('Longitude: {0:.2f}'.format(lon), style=style),
        html.Span('Latitude: {0:.2f}'.format(lat), style=style),
        html.Span('Altitude: {0:0.2f}'.format(alt), style=style)
    ]


# Multiple components can update everytime interval gets fired.
@app.callback(Output('live-update-graph', 'figure'),
              [Input('interval-component', 'n_intervals')])
def update_graph_live(n):
    satellite = Orbital('TERRA')
    data = {
        'time': [],
        'Latitude': [],
        'Longitude': [],
        'Altitude': []
    }

    # Collect some data
    for i in range(180):
        time = datetime.datetime.now() - datetime.timedelta(seconds=i*20
        lon, lat, alt = satellite.get_lonlatalt(
            time
        )
        data['Longitude'].append(lon)
        data['Latitude'].append(lat)
        data['Altitude'].append(alt)
        data['time'].append(time)

    # Create the graph with subplots
```

```
    fig = plotly.tools.make_subplots(rows=2, cols=1, vertical_spacing=0.
    fig['layout']['margin'] = {
        'l': 30, 'r': 10, 'b': 30, 't': 10
    }
    fig['layout']['legend'] = {'x': 0, 'y': 1, 'xanchor': 'left'}

    fig.append_trace({
        'x': data['time'],
        'y': data['Altitude'],
        'name': 'Altitude',
        'mode': 'lines+markers',
        'type': 'scatter'
    }, 1, 1)
    fig.append_trace({
        'x': data['Longitude'],
        'y': data['Latitude'],
        'text': data['time'],
        'name': 'Longitude vs Latitude',
        'mode': 'lines+markers',
        'type': 'scatter'
    }, 2, 1)

    return fig


if __name__ == '__main__':
    app.run_server(debug=True)
```

Updates on Page Load

By default, Dash apps store the `app.layout` in memory. This ensures that the `layout` is only computed once, when the app starts.

If you set `app.layout` to a function, then you can serve a dynamic layout on every page load.

For example, if your `app.layout` looked like this:

```
import datetime

import dash
```

```
import dash_html_components as html

app.layout = html.H1('The time is: ' + str(datetime.datetime.now()))

if __name__ == '__main__':
    app.run_server(debug=True)
```

then your app would display the time when the app was started.

If you change this to a function, then a new `datetime` will get computed everytime you refresh the page. Give it a try:

```
import datetime

import dash
import dash_html_components as html

def serve_layout():
    return html.H1('The time is: ' + str(datetime.datetime.now()))

app.layout = serve_layout

if __name__ == '__main__':
    app.run_server(debug=True)
```

> Heads up! You need to write `app.layout = serve_layout` not `app.layout = serve_layout()`. That is, define `app.layout` to the actual function instance.

You can combine this with time-expiring caching and serve a unique `layout` every hour or every day and serve the computed `layout` from memory in between.

Adding CSS & JS and Overriding the Page-Load Template

Dash applications are rendered in the web browser with CSS and JavaScript. On page load, Dash serves a small HTML template that includes references to the CSS and JavaScript that are required to render the application. This chapter covers everything that you need to know about configuring this HTML file and about including external CSS and JavaScript in Dash applications.

Table of Contents

Adding Your Own CSS and JavaScript to Dash Apps

New in dash 0.22.0

Including custom CSS or JavaScript in your Dash apps is simple. Just create a folder named `assets` in the root of your app directory and include your CSS and JavaScript files in that folder. Dash will automatically serve all of the files that are included in this folder. By default the url to request the assets will be `/assets` but you can customize this with the `assets_url_path` argument to `dash.Dash`.

Important: For these examples, you need to include `__name__` in your Dash constructor.

That is, `app = dash.Dash(__name__)` instead of `app = dash.Dash()`. Here's why.

Example: Including Local CSS and JavaScript

We'll create several files: `app.py`, a folder named `assets`, and three files in that folder:

```
- app.py
- assets/
    |-- typography.css
```

```
    |-- header.css
    |-- custom-script.js
```

`app.py`

```
import dash
import dash_core_components as dcc
import dash_html_components as html

app = dash.Dash(__name__)

app.layout = html.Div([
    html.Div(
        className="app-header",
        children=[
            html.Div('Plotly Dash', className="app-header--title")
        ]
    ),
    html.Div(
        children=html.Div([
            html.H5('Overview'),
            html.Div('''
                This is an example of a simple Dash app with
                local, customized CSS.
            ''')
        ])
    )
])

if __name__ == '__main__':
    app.run_server(debug=True)
```

`typography.css`

```
body {
    font-family: sans-serif;
}
h1, h2, h3, h4, h5, h6 {
```

```
    color: hotpink
}
```

`header.css`

```
.app-header {
    height: 60px;
    line-height: 60px;
    border-bottom: thin lightgrey solid;
}

.app-header .app-header--title {
    font-size: 22px;
    padding-left: 5px;
}
```

`custom-script.js`

```
alert('If you see this alert, then your custom JavaScript script has run
```

When you run `app.py`, your app should look something like this: (Note that there may be some slight differences in appearance as the CSS from this Dash User Guide is applied to all of these embedded examples.)

Plotly Dash

Overview

This is an example of a simple Dash app with local, customized CSS.

There are a few things to keep in mind when including assets automatically:

1 - The following file types will automatically be included:

A - CSS files suffixed with `.css`

B - JavaScript files suffixed with `.js`

C - A single file named `favicon.ico` (the page tab's icon)

2 - Dash will include the files in alphanumerical order by filename. So, we recommend prefixing your filenames with numbers if you need to ensure their order (e.g. `10_typography.css`, `20_header.css`)

3 - You can ignore certain files in your `assets` folder with a regex filter using `app = dash.Dash(assets_ignore='.*ignored.*')`. This will prevent Dash from loading files which contain the above pattern.

4 - If you want to include CSS from a remote URL, then see the next section.

5 - Your custom CSS will be included after the Dash component CSS

6 - It is recommended to add `__name__` to the dash init to ensure the resources in the assets folder are loaded, eg: `app = dash.Dash(__name__, meta_tags=[...])`. When you run your application through some other command line (like the flask command or gunicorn/waitress), the `__main__` module wil no longer located where `app.py` is. By explicitly setting `__name__`, Dash will be able to locate the relative `assets` folder correctly.

Hot Reloading

By default, Dash includes "hot-reloading". This means that Dash will automatically refresh your browser when you make a change in your Python code and your CSS code.

Give it a try: Change the color in `typography.css` from `hotpink` to `orange` and see your application update.

> Don't like hot-reloading? You can turn this off with `app.run_server(dev_tools_hot_reload=False)`. Learn more in Dash Dev Tools documentation. Questions? See the community forum hot reloading discussion.

Load Assets from a Folder Hosted on a CDN

If you duplicate the file structure of your local assets folder to a folder hosted externally to your Dash app, you can use `assets_external_path='http://your-external-assets-folder-url'` in the Dash constructor to load the files from there instead of locally. Dash will index your local assets folder to find all of your assets, map their relative path onto `assets_external_path` and then request the resources from there. `app.scripts.config.serve_locally = False` must also be set in order for this to work.

Example:

```
import dash
import dash_html_components as html

app = dash.Dash(
    __name__,
    assets_external_path='http://your-external-assets-folder-url/'
)
app.scripts.config.serve_locally = False
```

Embedding Images in Your Dash Apps

In addition to CSS and javascript files, you can include images in the `assets` folder. An example of the folder structure:

```
- app.py
- assets/
    |-- image.png
```

In your `app.py` file you can use the relative path to that image:

```
import dash
import dash_html_components as html

app = dash.Dash(__name__)

app.layout = html.Div([
    html.Img(src='/assets/image.png')
])

if __name__ == '__main__':
    app.run_server(debug=True)
```

Adding external CSS/Javascript

You can add resources hosted externally to your Dash app with the `external_stylesheets/stylesheets` init keywords.

The resources can be either a string or a dict containing the tag attributes (`src`, `integrity`, `crossorigin`, etc). You can mix both.

External css/js files are loaded before the assets.

Example:

```
import dash
import dash_html_components as html


# external JavaScript files
external_scripts = [
    'https://www.google-analytics.com/analytics.js',
    {'src': 'https://cdn.polyfill.io/v2/polyfill.min.js'},
    {
        'src': 'https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.10
        'integrity': 'sha256-Qqd/EfdABZUcAxjOkMi8eGEivtdTkh3b65xCZL4qAQA
        'crossorigin': 'anonymous'
    }
]

# external CSS stylesheets
external_stylesheets = [
    'https://codepen.io/chriddyp/pen/bWLwgP.css',
    {
        'href': 'https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/
        'rel': 'stylesheet',
        'integrity': 'sha384-MCw98/SFnGE8fJT3GXwEOngsV7Zt27NXFoaoApmYm81
        'crossorigin': 'anonymous'
    }
]


app = dash.Dash(__name__,
                external_scripts=external_scripts,
                external_stylesheets=external_stylesheets)

app.layout = html.Div()

if __name__ == '__main__':
    app.run_server(debug=True)
```

Customizing Dash's HTML Index Template

New in dash 0.22.0

Dash's UI is generated dynamically with Dash's React.js front-end. So, on page load, Dash serves a very small HTML template string that includes the CSS and JavaScript that is necessary to render the page and some simple HTML meta tags.

This simple HTML string is customizable. You might want to customize this string if you wanted to:

- Include a different `<title>` for your app (the `<title>` tag is the name that appears in your brower's tab. By default, it is "Dash")
- Customize the way that your CSS or JavaScript is included in the page. For example, if you wanted to include remote scripts or if you wanted to include the CSS before the Dash component CSS
- Include custom meta tags in your app. Note that meta tags can also be added with the `meta_tags` argument (example below).

Usage

Option 1 - `index_string`

Add an `index_string` to modify the default HTML Index Template:

```
import dash
import dash_html_components as html

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.index_string = '''
<!DOCTYPE html>
<html>
    <head>
        {%metas%}
        <title>{%title%}</title>
        {%favicon%}
        {%css%}
    </head>
    <body>
        <div>My Custom header</div>
        {%app_entry%}
```

```
        <footer>
            {%config%}
            {%scripts%}
        </footer>
        <div>My Custom footer</div>
    </body>
</html>
'''

app.layout = html.Div('Simple Dash App')

if __name__ == '__main__':
    app.run_server(debug=True)
```

The `{%key%}`s are template variables that Dash will fill in automatically with default properties. The available keys are:

`{%metas%}` (optional)

The registered meta tags included by the `meta_tags` argument in `dash.Dash`

`{%favicon%}` (optional)

A favicon link tag if found in the `assets` folder.

`{%css%}` (optional)

`<link/>` tags to css resources. These resources include the Dash component library CSS resources as well as any CSS resources found in the `assets` folder.

`{%title%}` (optional)

The contents of the page `<title>` tag. Learn more about `<title/>`

`{%config%}` (required)

An auto-generated tag that includes configuration settings passed from Dash's backend to Dash's front-end (`dash-renderer`).

`{%app_entry%}` (required)

The container in which the Dash layout is rendered.

`{%scripts%}` (required)

The set of JavaScript scripts required to render the Dash app. This includes the Dash component JavaScript files as well as any JavaScript files found in the `assets` folder.

Option 2 - `interpolate_index`

If your HTML content isn't static or if you would like to introspect or modify the templated variables, then you can override the `Dash.interpolate_index` method.

```
import dash
import dash_html_components as html


class CustomDash(dash.Dash):
    def interpolate_index(self, **kwargs):
        # Inspect the arguments by printing them
        print(kwargs)
        return '''
        <!DOCTYPE html>
        <html>
            <head>
                <title>My App</title>
            </head>
            <body>

                <div id="custom-header">My custom header</div>
                {app_entry}
                {config}
                {scripts}
                <div id="custom-footer">My custom footer</div>
            </body>
        </html>
        '''.format(
            app_entry=kwargs['app_entry'],
            config=kwargs['config'],
            scripts=kwargs['scripts'])

app = CustomDash()

app.layout = html.Div('Simple Dash App')

if __name__ == '__main__':
    app.run_server(debug=True)
```

Unlike the `index_string` method, where we worked with template string variables, the keyword variables that are passed into `interpolate_index` are already evaluated.

In the example above, when we print the input arguments of `interpolate_index` we should see an output like this:

```
{
    'title': 'Dash',
```

```
    'app_entry': '\n<div id="react-entry-point">\n    <div class="_dash-l
    'favicon': '',
    'metas': '<meta charset="UTF-8"/>',
    'scripts': '<script src="https://unpkg.com/react@15.4.2/dist/react.mi
    'config': '<script id="_dash-config" type="application/json">{"reques
    'css': ''
}
```

The values of the `scripts` and `css` keys may be different depending on which component libraries you have included or which files might be in your assets folder.

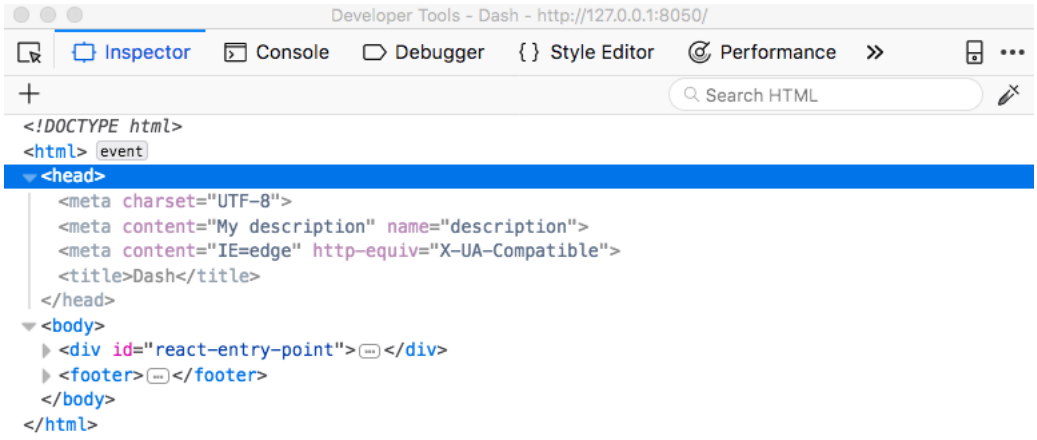
Customizing Meta Tags

> Not sure what meta tags are? Check out this tutorial on meta tags and why you might want to use them.

To add custom meta tags to your application, you can always override Dash's HTML Index Template. Alternatively, Dash provides a shortcut: you can specify meta tags directly in the Dash constructor:

```
import dash
import dash_html_components as html

app = dash.Dash(meta_tags=[
    {
        'name': 'description',
        'content': 'My description'
    },
    {
        'http-equiv': 'X-UA-Compatible',
        'content': 'IE=edge'
    }
])

app.layout = html.Div('Simple Dash App')

if __name__ == '__main__':
    app.run_server(debug=True)
```

If you inspect the source of your app, you should see the meta tags appear:



Serving Dash's Component Libraries Locally or from a CDN

Dash's component libraries, like `dash_core_components` and `dash_html_components`, are bundled with JavaScript and CSS files. Dash automatically checks with component libraries are being used in your application and will automatically serve these files in order to render the application.

By default, dash serves the JavaScript and CSS resources from the online CDNs. This is usually much faster than loading the resources from the file system.

However, if you are working in an offline or firewalled environment or if the CDN is unavailable, then your dash app itself can serve these files. These files are stored as part of the component's site-packages folder.

Here's how to enable this option:

```
from dash import Dash

app = Dash()

app.css.config.serve_locally = True
app.scripts.config.serve_locally = True
```

Note that in the future, we will likely make "offline" the default option. Follow dash#284 for more information.

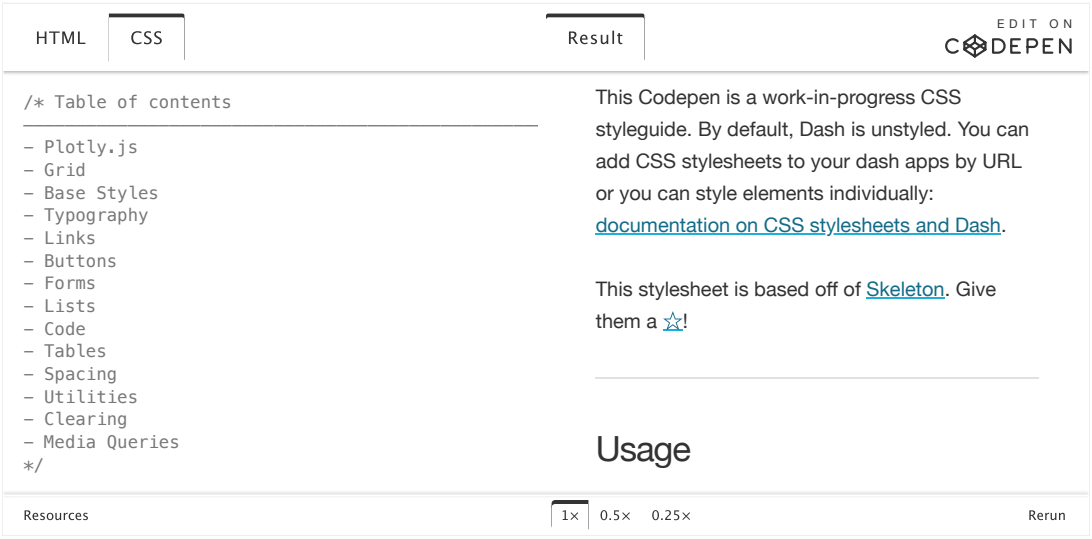
Sample Dash CSS Stylesheet

Currently, Dash does not include styles by default.

To get started with Dash styles, we recommend starting with this `CSS stylesheet` hosted on `Codepen` .

To include this stylesheet in your application, copy and paste it into a file in your `assets` folder. You can view the raw CSS source here: `https://codepen.io/chriddyp/pen/bWLwgP.css` .

Here is an embedded version of this stylesheet.



Multi-Page Apps and URL Support

Dash renders web applications as a "single-page app". This means that the application does not completely reload when the user navigates the application, making browsing very fast.

There are two new components that aid page navigation: `dash_core_components.Location` and `dash_core_components.Link`.

`dash_core_components.Location` represents the location bar in your web browser through the `pathname` property. Here's a simple example:

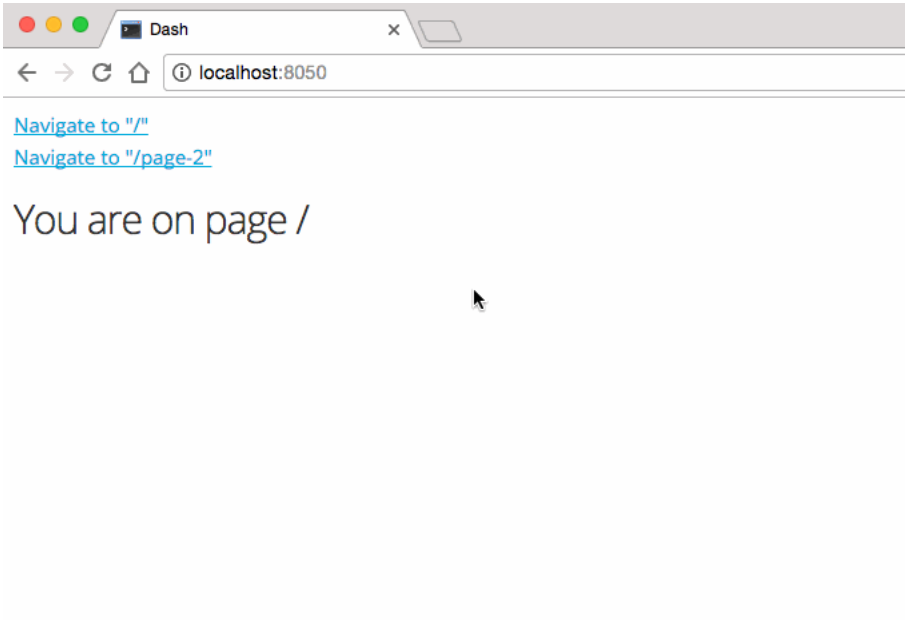
```
import dash
import dash_core_components as dcc
import dash_html_components as html

print(dcc.__version__) # 0.6.0 or above is required

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div([
    # represents the URL bar, doesn't render anything
    dcc.Location(id='url', refresh=False),

    dcc.Link('Navigate to "/"', href='/'),
    html.Br(),
    dcc.Link('Navigate to "/page-2"', href='/page-2'),

    # content will be rendered in this element
    html.Div(id='page-content')
])


@app.callback(dash.dependencies.Output('page-content', 'children'),
              [dash.dependencies.Input('url', 'pathname')])
def display_page(pathname):
    return html.Div([
        html.H3('You are on page {}'.format(pathname))
    ])
```

```
if __name__ == '__main__':
    app.run_server(debug=True)
```

In this example, the callback `display_page` receives the current pathname (the last part of the URL) of the page. The callback simply displays the `pathname` on page but it could use the `pathname` to display different content.

The `Link` element updates the `pathname` of the browser without refreshing the page. If you used a `html.A` element instead, the `pathname` would update but the page would refresh.

Here is a GIF of what this example looks like. Note how clicking on the `Link` doesn't refresh the page even though it updates the URL!

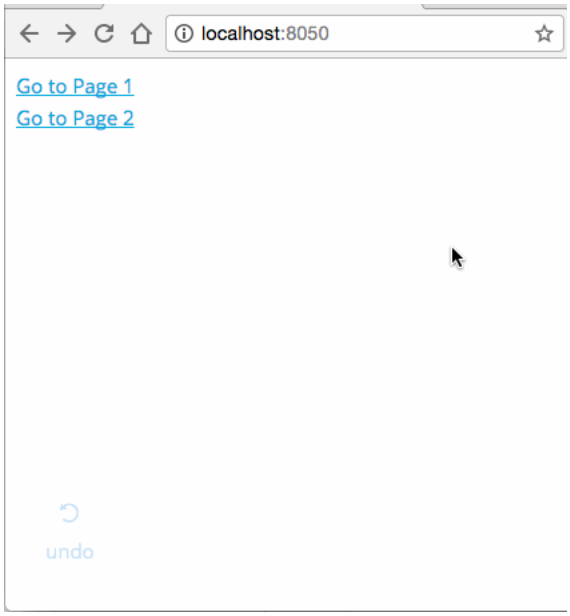


You can modify the example above to display different pages depending on the URL:

```
import dash
import dash_core_components as dcc
import dash_html_components as html

print(dcc.__version__) # 0.6.0 or above is required

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

# Since we're adding callbacks to elements that don't exist in the app.l
```

```
# Dash will raise an exception to warn us that we might be
# doing something wrong.
# In this case, we're adding the elements through a callback, so we can
# the exception.
app.config.suppress_callback_exceptions = True

app.layout = html.Div([
    dcc.Location(id='url', refresh=False),
    html.Div(id='page-content')
])


index_page = html.Div([
    dcc.Link('Go to Page 1', href='/page-1'),
    html.Br(),
    dcc.Link('Go to Page 2', href='/page-2'),
])

page_1_layout = html.Div([
    html.H1('Page 1'),
    dcc.Dropdown(
        id='page-1-dropdown',
        options=[{'label': i, 'value': i} for i in ['LA', 'NYC', 'MTL']]
        value='LA'
    ),
    html.Div(id='page-1-content'),
    html.Br(),
    dcc.Link('Go to Page 2', href='/page-2'),
    html.Br(),
    dcc.Link('Go back to home', href='/'),

])

@app.callback(dash.dependencies.Output('page-1-content', 'children'),
              [dash.dependencies.Input('page-1-dropdown', 'value')])
def page_1_dropdown(value):
    return 'You have selected "{}"'.format(value)


page_2_layout = html.Div([
    html.H1('Page 2'),
    dcc.RadioItems(
```

```
        id='page-2-radios',
        options=[{'label': i, 'value': i} for i in ['Orange', 'Blue', 'R
        value='Orange'
    ),
    html.Div(id='page-2-content'),
    html.Br(),
    dcc.Link('Go to Page 1', href='/page-1'),
    html.Br(),
    dcc.Link('Go back to home', href='/')
])

@app.callback(dash.dependencies.Output('page-2-content', 'children'),
              [dash.dependencies.Input('page-2-radios', 'value')])
def page_2_radios(value):
    return 'You have selected "{}"'.format(value)


# Update the index
@app.callback(dash.dependencies.Output('page-content', 'children'),
              [dash.dependencies.Input('url', 'pathname')])
def display_page(pathname):
    if pathname == '/page-1':
        return page_1_layout
    elif pathname == '/page-2':
        return page_2_layout
    else:
        return index_page
    # You could also return a 404 "URL not found" page here


if __name__ == '__main__':
    app.run_server(debug=True)
```



In this example, we're displaying different layouts through the `display_page` function. A few notes:

- Each page can have interactive elements even though those elements may not be in the initial view. Dash handles these "dynamically generated" components gracefully: as they are rendered, they will trigger the callbacks with their initial values.
- Since we're adding callbacks to elements that don't exist in the app.layout, Dash will raise an exception to warn us that we might be doing something wrong. In this case, we're adding the elements through a callback, so we can ignore the exception by setting `app.config.suppress_callback_exceptions = True`. It is also possible to do this without suppressing callback exceptions. See the example below for details.
- You can modify this example to import the different page's `layout`s in different files.
- This Dash Userguide that you're looking at is itself a multi-page Dash app, using these same principles.

Dynamically Create a Layout for Multi-Page App Validation

Dash applies validation to your callbacks, which performs checks such as validating the types of callback arguments and checking to see whether the specified Input and Output components actually have the specified properties.

For full validation, all components within your callback must therefore appear in the initial layout of your app, and you will see an error if they do not. However, in the case of more complex Dash apps that involve dynamic modification of the layout (such as multi-page apps), not every component appearing in your callbacks will be included in the initial layout.

Since this validation is done before flask has any request context, you can create a layout function that checks `flask.has_request_context()` and returns a complete layout to the validator if there is no request context and returns the incomplete index layout otherwise.

```
import dash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Input, Output, State

import flask

app = dash.Dash(
    __name__,
    external_stylesheets=['https://codepen.io/chriddyp/pen/bWLwgP.css']
)

url_bar_and_content_div = html.Div([
    dcc.Location(id='url', refresh=False),
    html.Div(id='page-content')
])

layout_index = html.Div([
    dcc.Link('Navigate to "/page-1"', href='/page-1'),
    html.Br(),
    dcc.Link('Navigate to "/page-2"', href='/page-2'),
])

layout_page_1 = html.Div([
    html.H2('Page 1'),
    dcc.Input(id='input-1-state', type='text', value='Montreal'),
    dcc.Input(id='input-2-state', type='text', value='Canada'),
    html.Button(id='submit-button', n_clicks=0, children='Submit'),
    html.Div(id='output-state'),
    html.Br(),
    dcc.Link('Navigate to "/"', href='/'),
    html.Br(),
```

```
    dcc.Link('Navigate to "/page-2"', href='/page-2'),
])

layout_page_2 = html.Div([
    html.H2('Page 2'),
    dcc.Dropdown(
        id='page-2-dropdown',
        options=[{'label': i, 'value': i} for i in ['LA', 'NYC', 'MTL']]
        value='LA'
    ),
    html.Div(id='page-2-display-value'),
    html.Br(),
    dcc.Link('Navigate to "/"', href='/'),
    html.Br(),
    dcc.Link('Navigate to "/page-1"', href='/page-1'),
])


def serve_layout():
    if flask.has_request_context():
        return url_bar_and_content_div
    return html.Div([
        url_bar_and_content_div,
        layout_index,
        layout_page_1,
        layout_page_2,
    ])


app.layout = serve_layout


# Index callbacks
@app.callback(Output('page-content', 'children'),
              [Input('url', 'pathname')])
def display_page(pathname):
    if pathname == "/page-1":
        return layout_page_1
    elif pathname == "/page-2":
        return layout_page_2
    else:
        return layout_index
```

```
# Page 1 callbacks
@app.callback(Output('output-state', 'children'),
              [Input('submit-button', 'n_clicks')],
              [State('input-1-state', 'value'),
               State('input-2-state', 'value')])
def update_output(n_clicks, input1, input2):
    return ('The Button has been pressed {} times,'
            'Input 1 is "{}",'
            'and Input 2 is "{}"').format(n_clicks, input1, input2)


# Page 2 callbacks
@app.callback(Output('page-2-display-value', 'children'),
              [Input('page-2-dropdown', 'value')])
def display_value(value):
    print('display_value')
    return 'You have selected "{}"'.format(value)


if __name__ == '__main__':
    app.run_server(debug=True)
```

Structuring a Multi-Page App

Here's how to structure a multi-page app, where each app is contained in a separate file.

File structure:

```
- app.py
- index.py
- apps
   |-- __init__.py
   |-- app1.py
   |-- app2.py
```

`app.py`

```
import dash

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
server = app.server
app.config.suppress_callback_exceptions = True
```

`apps/app1.py`

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

from app import app

layout = html.Div([
    html.H3('App 1'),
    dcc.Dropdown(
        id='app-1-dropdown',
        options=[
            {'label': 'App 1 - {}'.format(i), 'value': i} for i in [
                'NYC', 'MTL', 'LA'
            ]
        ]
    ),
    html.Div(id='app-1-display-value'),
    dcc.Link('Go to App 2', href='/apps/app2')
])


@app.callback(
    Output('app-1-display-value', 'children'),
    [Input('app-1-dropdown', 'value')])
def display_value(value):
    return 'You have selected "{}"'.format(value)
```

And similarly for other apps

`index.py`

`index.py` loads different apps on different urls like this:

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

from app import app
from apps import app1, app2


app.layout = html.Div([
    dcc.Location(id='url', refresh=False),
    html.Div(id='page-content')
])


@app.callback(Output('page-content', 'children'),
              [Input('url', 'pathname')])
def display_page(pathname):
    if pathname == '/apps/app1':
         return app1.layout
    elif pathname == '/apps/app2':
         return app2.layout
    else:
        return '404'

if __name__ == '__main__':
    app.run_server(debug=True)
```

Dash Dev Tools

Dash Dev Tools is an initiative to make debugging and developing Dash apps more pleasant. This initiative was sponsored by an organization and you can see our work in our GitHub project.

All dev_tools features are activated by default when you run the app with `app.run_server(debug=True)`

Hot Reloading

New in dash 0.30.0 and dash-renderer 0.15.0

By default, Dash includes "hot-reloading". This means that Dash will automatically refresh your browser when you make a change in your Python or CSS code.

Hot reloading works by running a "file watcher" that examines your working directory to check for changes. When a change is detected, Dash reloads your application in an efficient way automatically. A few notes:

- Hot reloading is triggered when you save a file.
- Dash examines the files in your working directory.
- CSS files are automatically "watched" by examining the `assets/` folder. Learn more about css
- If only CSS changed, then Dash will only refresh that CSS file.
- When your Python code has changed, Dash will re-run the entire file and then refresh the application in the browser.
- If your application initialization is slow, then you might want to consider how to save certain initialization steps to file. For example, if your app initialization downloads a static file from a remote service, perhaps you could include it locally.
- Hot reloading will not save the application's state. For example, if you've selected some items in a dropdown, then that item will be cleared on hot-reload.
- Hot reloading is configurable through a set of parameters. See the Reference section at the end of this page.

Serving the dev bundles

Component library bundles are minified by default, if you encounter a front-end error in your code, the variables names will be mangled. By serving the dev bundles, you will get the full names.

Reference

The full set of dev tools parameters are included in `app.run_server` or `app.enable_dev_tools`:

- `debug`, bool, activate all the dev tools.
- `dev_tools_hot_reload`, bool, set to true to enable hot reload (default=False).
- `dev_tools_hot_reload_interval`, int, interval in millisecond at which the renderer will request the reload hash (default=3000).
- `dev_tools_hot_reload_watch_interval`, float, delay in seconds between each walk of the assets folder to detect file changes. (default=0.5 seconds)
- `dev_tools_hot_reload_max_retry`, int, number of times the reloader is allowed to fail before stopping and sending an alert. (default=8)
- `dev_tools_silence_routes_logging`, bool, remove the routes access logging from the console.
- `dev_tools_serve_dev_bundles`, bool, serve the dev bundles.

Settings with environment variables

All the `dev_tools` variables can be set with environment variables, just replace the `dev_tools_` with `dash_`. This allows you to have different run configs without changing the code.

Linux/macOS:

`export DASH_HOT_RELOAD=false`

Windows:

`set DASH_DEBUG=true`

Production

Authentication

Authentication for dash apps is provided through a separate `dash-auth` package.

`dash-auth` provides two methods of authentication: **HTTP Basic Auth** and **Plotly OAuth**.

HTTP Basic Auth is one of the simplest forms of authentication on the web. As a Dash developer, you hardcode a set of usernames and passwords in your code and send those usernames and passwords to your viewers. There are a few limitations to HTTP Basic Auth:

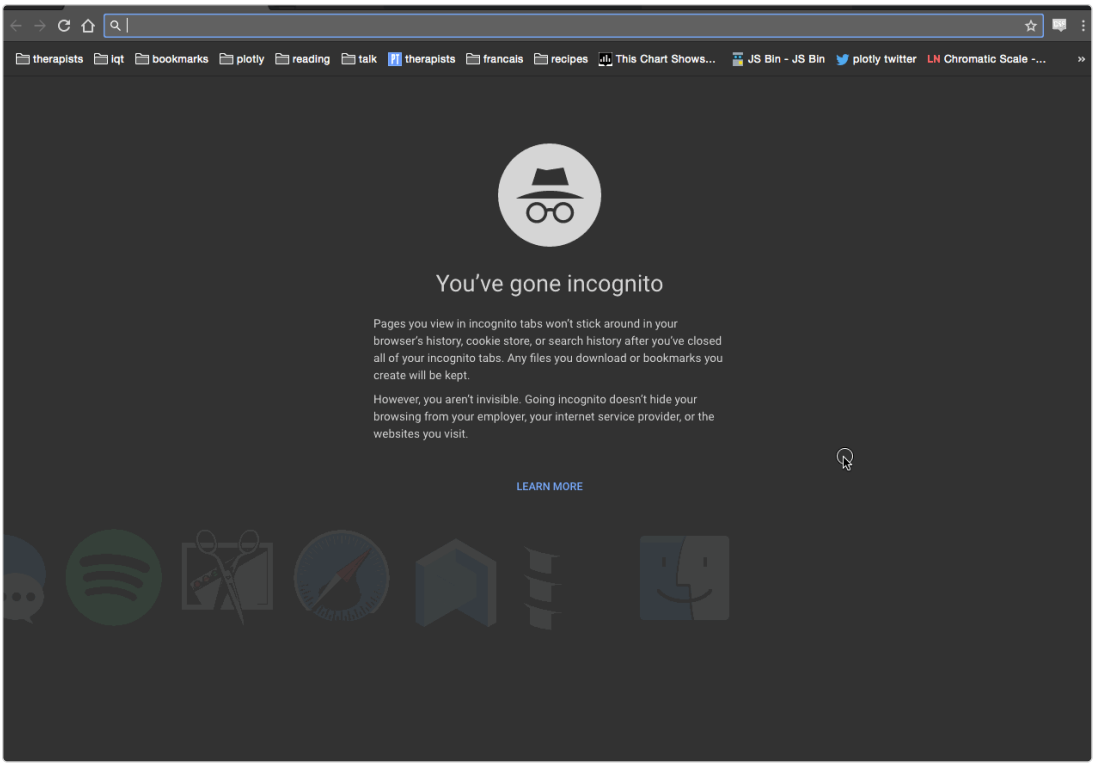
- Users can not log out of applications
- You are responsible for sending the usernames and passwords to your viewers over a secure channel
- Your viewers can not create their own account and cannot change their password
- You are responsible for safely storing the username and password pairs in your code.

Plotly OAuth provides authentication through your online Plotly account or through your company's Plotly Enterprise server. As a Dash developer, this requires a paid Plotly subscription. Here's where you can subscribe to Plotly Cloud, and here's where you can contact us about Plotly Enterprise. The viewers of your app will need a Plotly account but they do not need to upgrade to a paid subscription.

Plotly OAuth allows you to share your apps with other users who have Plotly accounts. With Plotly Enterprise, this includes sharing apps through the integrated LDAP system. Apps that you have saved will appear in your list of files at https://plot.ly/organize and you can manage the permissions of the apps there. Viewers create and manage their own accounts.

Basic Auth Example

Logging in through Basic Auth looks like this:



Installation:

```
pip install dash==0.35.1
pip install dash-auth==1.3.1
```

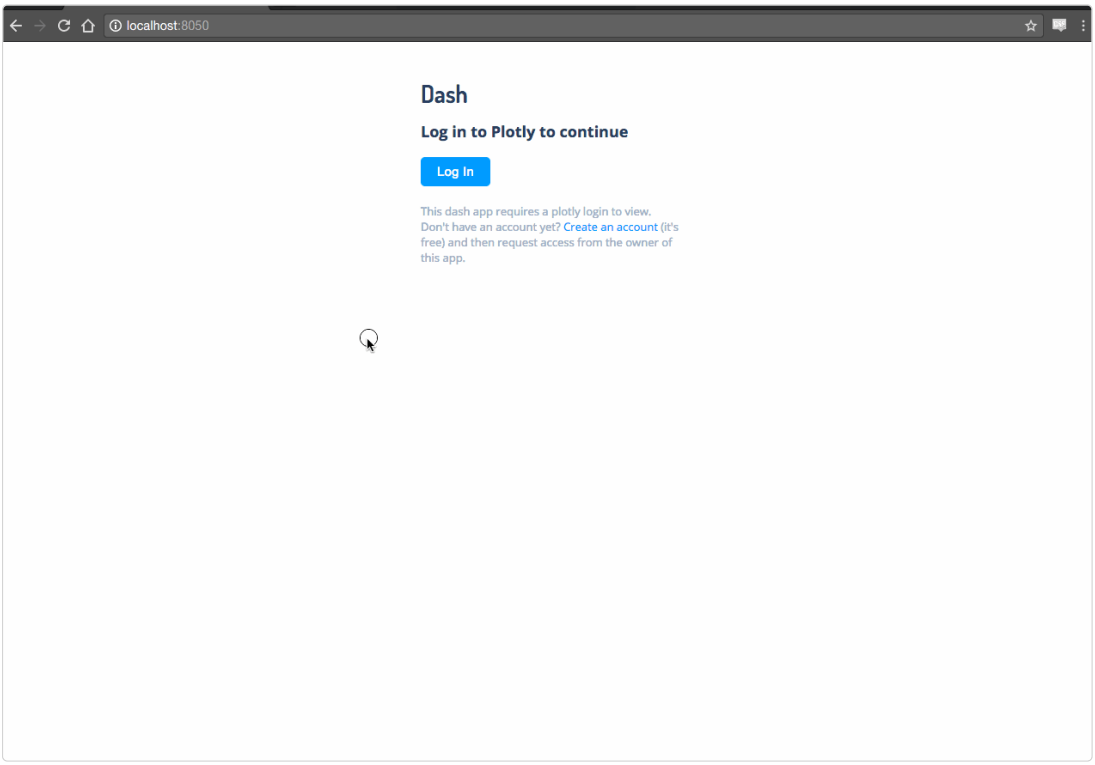
Example Code:

```
import dash
import dash_auth
import dash_core_components as dcc
import dash_html_components as html
import plotly

# Keep this out of source code repository - save in a file or a database
VALID_USERNAME_PASSWORD_PAIRS = [
    ['hello', 'world']
]

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
auth = dash_auth.BasicAuth(
    app,
```

```
    VALID_USERNAME_PASSWORD_PAIRS
)

app.layout = html.Div([
    html.H1('Welcome to the app'),
    html.H3('You are successfully authorized'),
    dcc.Dropdown(
        id='dropdown',
        options=[{'label': i, 'value': i} for i in ['A', 'B']],
        value='A'
    ),
    dcc.Graph(id='graph')
], className='container')

@app.callback(
    dash.dependencies.Output('graph', 'figure'),
    [dash.dependencies.Input('dropdown', 'value')])
def update_graph(dropdown_value):
    return {
        'layout': {
            'title': 'Graph of {}'.format(dropdown_value),
            'margin': {
                'l': 20,
                'b': 20,
                'r': 10,
                't': 60
            }
        },
        'data': [{'x': [1, 2, 3], 'y': [4, 1, 2]}]
    }

app.scripts.config.serve_locally = True


if __name__ == '__main__':
    app.run_server(debug=True)
```

Plotly OAuth Example

Logging in through Plotly OAuth looks like this:



Installation:

```
pip install dash==0.35.1
pip install dash-auth==1.3.1
```

Example Code:

```
import dash
import dash_auth
import dash_core_components as dcc
import dash_html_components as html
import plotly


# Modify these variables with your own info
APP_NAME = 'Dash Authentication Sample App'
APP_URL = 'https://my-dash-app.herokuapps.com'

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

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

```
auth = dash_auth.PlotlyAuth(
    app,
    APP_NAME,
    'private',
    APP_URL
)

app.layout = html.Div([
    html.H1('Welcome to the app'),
    html.H3('You are successfully authorized'),
    dcc.Dropdown(
        id='dropdown',
        options=[{'label': i, 'value': i} for i in ['A', 'B']],
        value='A'
    ),
    dcc.Graph(id='graph')
], className='container')

@app.callback(
    dash.dependencies.Output('graph', 'figure'),
    [dash.dependencies.Input('dropdown', 'value')])
def update_graph(dropdown_value):
    return {
        'layout': {
            'title': 'Graph of {}'.format(dropdown_value),
            'margin': {
                'l': 20,
                'b': 20,
                'r': 10,
                't': 60
            }
        },
        'data': [{'x': [1, 2, 3], 'y': [4, 1, 2]}]
    }

app.scripts.config.serve_locally = True

if __name__ == '__main__':
    app.run_server(debug=True)
```

Methods on PlotlyAuth Objects

With Plotly OAuth, it is possible to create create cookies that store information related to a user. In the example below we use the `@auth.is_authorized_hook` and `auth.set_user_data` to create a cookie containing an object associated with their account and then we determine whether they can view the graph by checking their permissions in that cookie using `auth.get_user_data`.

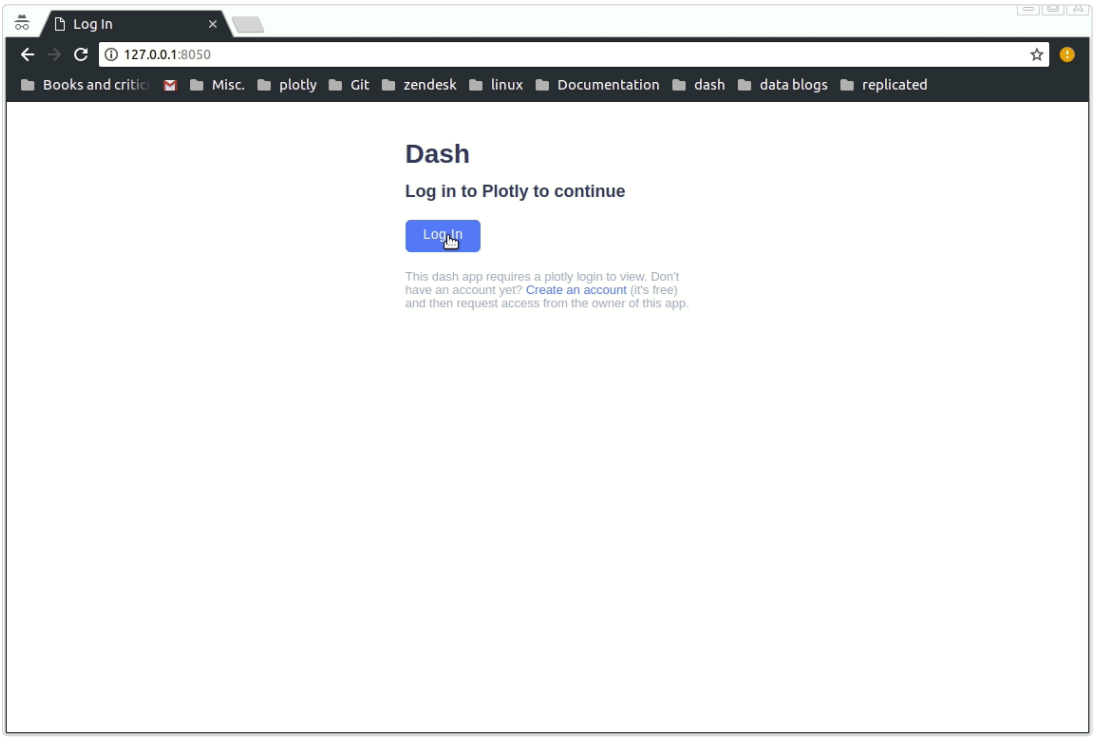
Finally, we can logout the user by clearing the cookies. To do so, you can create a logout button and insert it in the layout or use `auth.logout()` in a callback.

Plotly Auth uses the environment variables `PLOTLY_USERNAME` and `PLOTLY_API_KEY`. You can find your username and API key at https://plot.ly/settings/api or, if you are using Dash Deployment Server, at https://your-plotly-server.com/settings/api.

You can set these variables directly in your code with:

```
import os

  os.environ['PLOTLY_USERNAME'] = 'your-username'
  os.environ['PLOTLY_API_KEY'] = 'your-api-key'
```

or, if you are using Dash Deployment Server, you can keep your environment variables secret (view the docs).



Installation:

```
pip install dash==0.35.1  # The core dash backend
pip install dash-auth==1.3.1  # Dash Auth components
```

Example Code:

```
import dash
import dash_auth
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Output, Input

import os
# Modify these variables with your own info
APP_NAME = 'Dash Authentication Sample App'
APP_URL = 'http://127.0.0.1:8050/'


external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
auth = dash_auth.PlotlyAuth(
    app,
    APP_NAME,
    'public',
    APP_URL
)

app.layout = html.Div([
    html.H1('Welcome to the app', id='title'),
    html.Div(id='authorized'),
    html.Button('View Graph', id='btn'),
    dcc.Graph(id='graph', figure={
        'layout': {
            'title': 'Private Graph',
            'margin': {
                'l': 20,
                'b': 20,
                'r': 10,
                't': 60
            }
        },
        'data': [{'x': [1, 2, 3], 'y': [4, 1, 2]}]
```

```
    }, style={'display': 'none'}),
    auth.create_logout_button(
        label='Sign out',
        redirect_to='https://plot.ly')],
     className='container')


@app.callback(Output('title', 'children'), [Input('title', 'id')])
def give_name(title):
    username = auth.get_username()
    return 'Welcome to the app, {}'.format(username)


@auth.is_authorized_hook
def is_authorized(data):
    active = data.get('is_active')
    if active:
        auth.set_user_data({'graph_1': True})
    return active

@app.callback(Output('authorized', 'children'), [Input('btn', 'n_clicks'
def check_perms(n_clicks):
    if n_clicks:
        perms = auth.get_user_data()
        perm_view_graph = perms.get('graph_1')
        if not perm_view_graph:
            return 'You are not authorized to view this content'
        else:
            return 'You are authorized!'


@app.callback(Output('graph', 'style'), [Input('btn', 'n_clicks')])
def check_perms_graph_update(n_clicks):
    if n_clicks:
        perms = auth.get_user_data()
        perm_view_graph = perms.get('graph_1')
        if perm_view_graph:
            return {}
        else:
            return {'display': 'none'}
    else:
        return {'display': 'none'}
```

```
if __name__ == '__main__':
    app.run_server(debug=True)
```

Deploying Dash Apps

By default, Dash apps run on `localhost` - you can only access them on your own machine. To share a Dash app, you need to "deploy" your Dash app to a server and open up the server's firewall to the public or to a restricted set of IP addresses.

Dash uses Flask under the hood. This makes deployment easy: you can deploy a Dash app just like you would deploy a Flask app. Almost every cloud server provider has a guide for deploying Flask apps. For more, see the official Flask Guide to Deployment or view the tutorial on deploying to Heroku below.

Dash Deployment Server

Dash Deployment Server is Plotly's commercial product for deploying Dash Apps on your company's servers or on AWS, Google Cloud, or Azure. It offers an enterprise-wide Dash App Portal, easy git-based deployment, automatic URL namespacing, built-in SSL support, LDAP authentication, and more. Learn more about Dash Deployment Server or get in touch to start a trial.

For existing customers, see the Dash Deployment Server Documentation.

Dash and Flask

Dash apps are web applications. Dash uses Flask as the web framework. The underlying Flask app is available at `app.server`, that is:

```
import dash

app = dash.Dash(__name__)

server = app.server # the Flask app
```

You can also pass your own flask app instance into Dash:

```
import flask

server = flask.Flask(__name__)
app = dash.Dash(__name__, server=server)
```

By exposing this `server` variable, you can deploy Dash apps like you would any Flask app. For more, see the official Flask Guide to Deployment. Note that

> While lightweight and easy to use, Flask's built-in server is not suitable for production as it doesn't scale well and by default serves only one request at a time

Heroku Example

Heroku is one of the easiest platforms for deploying and managing public Flask applications.

View the official Heroku guide to Python .

Here is a simple example. This example requires a Heroku account, `git`, and `virtualenv`.

Step 1. Create a new folder for your project:

```
$ mkdir dash_app_example
$ cd dash_app_example
```

Step 2. Initialize the folder with `git` and a `virtualenv`

```
$ git init        # initializes an empty git repo
$ virtualenv venv # creates a virtualenv called "venv"
$ source venv/bin/activate # uses the virtualenv
```

`virtualenv` creates a fresh Python instance. You will need to reinstall your app's dependencies with this virtualenv:

```
$ pip install dash
$ pip install dash-renderer
$ pip install dash-core-components
$ pip install dash-html-components
$ pip install plotly
```

You will also need a new dependency, `gunicorn`, for deploying the app:

```
$ pip install gunicorn
```

Step 3. Initialize the folder with a sample app (`app.py`), a `.gitignore` file, `requirements.txt`, and a `Procfile` for deployment

Create the following files in your project folder:

`app.py`

```
import os

import dash
```

```
import dash_core_components as dcc
import dash_html_components as html

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

server = app.server

app.layout = html.Div([
    html.H2('Hello World'),
    dcc.Dropdown(
        id='dropdown',
        options=[{'label': i, 'value': i} for i in ['LA', 'NYC', 'MTL']]
        value='LA'
    ),
    html.Div(id='display-value')
])

@app.callback(dash.dependencies.Output('display-value', 'children'),
              [dash.dependencies.Input('dropdown', 'value')])
def display_value(value):
    return 'You have selected "{}"'.format(value)

if __name__ == '__main__':
    app.run_server(debug=True)
```

.gitignore

```
venv
*.pyc
.DS_Store
.env
```

Procfile

```
web: gunicorn app:server
```

(Note that `app` refers to the filename `app.py`. `server` refers to the variable `server` inside that file).

`requirements.txt`

`requirements.txt` describes your Python dependencies. You can fill this file in automatically with:

```
$ pip freeze > requirements.txt
```

---

4. Initialize Heroku, add files to Git, and deploy

```
$ heroku create my-dash-app # change my-dash-app to a unique name
$ git add . # add all files to git
$ git commit -m 'Initial app boilerplate'
$ git push heroku master # deploy code to heroku
$ heroku ps:scale web=1  # run the app with a 1 heroku "dyno"
```

You should be able to view your app at `https://my-dash-app.herokuapp.com` (changing `my-dash-app` to the name of your app).

5. Update the code and redeploy

When you modify `app.py` with your own code, you will need to add the changes to git and push those changes to heroku.

```
$ git status # view the changes
$ git add .  # add all the changes
$ git commit -m 'a description of the changes'
$ git push heroku master
```

This workflow for deploying apps on heroku is very similar to how deployment works with the Plotly Enterprise's Dash Deployment Server. Learn more or get in touch.

Getting Help

Dash Support and Contact

Dash is an open-source product that is developed and maintained by Plotly.

Dash Demos and Enterprise Trials

If you would like to trial or purchase a Dash Deployment Server, get in touch with us directly.

Our sales engineering team is happy to give you or your team a demo of Dash and Dash Deployment Server too.

Dash Workshops

We lead two day Dash workshops every few months. These frequently sell out, so please register early

These workshops are lead by the authors of Dash and are tailored towards both beginners and experts.

Sponsored Feature Requests and Customizations

If you or your company would like to sponsor a specific feature or enterprise customization, get in touch with our advanced development team.

Community Support

Our community forum at community.plot.ly has a topic dedicated on Dash. This forum is great for showing off projects, feature requests, and general questions.

If you have found a bug, you can open an issue on GitHub at plotly/dash.

Direct Contact

If you would like to reach out to me directly, you can email me at chris@plot.ly or on Twitter at @chriddyp.

Plotly is also on Twitter at @plotlygraphs.

We are based in Montréal, Canada and our headquarters are in the Mile End. If you're in the neighborhood, come say hi!