



Develop Data Visualization Interfaces in Python With Dash

by [Dylan Castillo](#) 48 Comments data-science intermediate

Mark as Completed

Tweet Share Email

Table of Contents

- [What Is Dash?](#)
- [Get Started With Dash in Python](#)
 - [How to Set Up Your Local Environment](#)
 - [How to Build a Dash Application](#)
- [Style Your Dash Application](#)
 - [How to Apply a Custom Style to Your Components](#)
 - [How to Improve the Looks of Your Dashboard](#)
- [Add Interactivity to Your Dash Apps Using Callbacks](#)
 - [How to Create Interactive Components](#)
 - [How to Define Callbacks](#)
- [Deploy Your Dash Application to Heroku](#)
- [Conclusion](#)

[Remove ads](#)

Watch Now

This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Data Visualization Interfaces in Python With Dash](#)

In the past, creating analytical web applications was a task for seasoned developers that required knowledge of multiple programming languages and frameworks. That’s no longer the case. Nowadays, you can make data visualization interfaces using pure Python. One popular tool for this is [Dash](#).

Dash gives data scientists the ability to showcase their results in interactive web applications. You don’t need to be an expert in [web development](#). In an afternoon, you can build and deploy a Dash app to share with others.

In this tutorial, you’ll learn how to:

- Create a **Dash application**
- Use Dash **core components** and **HTML components**
- **Customize the style** of your Dash application
- Use **callbacks** to build interactive applications
- Deploy your application on **Heroku**

You can download the source code, data, and resources for the sample application you'll make in this tutorial by clicking the link below:

Get the Source Code: [Click here to get the source code you'll use](#) to learn about creating data visualization interfaces in Python with Dash in this tutorial.

What Is Dash?

Dash is an open source framework for building data visualization interfaces. Released in 2017 as a Python library, it's grown to include implementations for R and Julia. Dash helps data scientists build analytical web applications without requiring advanced web development knowledge.

Three technologies constitute the core of Dash:

1. **Flask** supplies the web server functionality.
2. **React.js** renders the user interface of the web page.
3. **Plotly.js** generates the charts used in your application.

But you don't have to worry about making all these technologies work together. Dash will do that for you. You just need to write Python, R, or Julia and sprinkle it with a bit of CSS.

[Plotly](#), a Canada-based company, built Dash and supports its development. You may know the company from the [popular graphing libraries](#) that share its name. Plotly (the company) open-sourced Dash and released it under an [MIT license](#), so you can use Dash at no cost.

Plotly also offers a commercial companion to Dash called [Dash Enterprise](#). This paid service provides companies with support services such as hosting, deploying, and handling authentication on Dash applications. But these features live outside of Dash's open source ecosystem.

Dash will help you build dashboards quickly. If you're used to analyzing data or building data visualizations using Python, then Dash will be a useful addition to your toolbox. Here are a few examples of what you can make with Dash:

- [A dashboard to analyze trading positions in real-time](#)
- [A visualization of millions of Uber rides](#)
- [An interactive financial report](#)

This is just a tiny sample. If you'd like to see other interesting use cases, then go check the [Dash App Gallery](#).

Note: You don't need advanced knowledge of web development to follow this tutorial, but some familiarity with HTML and CSS won't hurt.

The rest of this tutorial assumes you know the basics of the following topics:

- Python graphing libraries such as Plotly, [Bokeh](#), or [Matplotlib](#)
- HTML and the [structure of an HTML file](#)
- [CSS and style sheets](#)

If you feel comfortable with the requirements and want to learn how to use Dash in your next project, then continue to the following section!

 [Remove ads](#)

Get Started With Dash in Python

In this tutorial, you'll go through the end-to-end process of building a dashboard using Dash. If you follow along with the examples, then you'll go from a bare-bones dashboard on your local machine to a styled dashboard deployed on [Heroku](#).

To build the dashboard, you'll use a [dataset](#) of sales and prices of avocados in the United States between 2015 and 2018. This dataset was compiled by [Justin Kiggins](#) using data from the [Hass Avocado Board](#).

How to Set Up Your Local Environment

To develop your app, you'll need a new directory to store your code and data and a clean Python 3 [virtual environment](#). To create those, follow the instructions below, choosing the version that matches your operating system.

If you're using **Windows**, then open a command prompt and execute these commands:

Windows Command Prompt

```
c:\> mkdir avocado_analytics && cd avocado_analytics
c:\> c:\path\to\python\launcher\python -m venv venv
c:\> venv\Scripts\activate.bat
```

The first command creates a directory for your project and moves your current location there. The second command creates a virtual environment in that location. The last command activates the virtual environment. Make sure to replace the path in the second command with the path of your Python 3 launcher.

If you're using **macOS** or **Linux**, then follow these steps from a terminal:

Shell

```
$ mkdir avocado_analytics && cd avocado_analytics
$ python3 -m venv venv
$ source venv/bin/activate
```

The first two commands perform the following actions:

1. Create a directory called `avocado_analytics`
2. Move your current location to the `avocado_analytics` directory
3. Create a clean virtual environment called `venv` inside that directory

The last command activates the virtual environment you just created.

Next, you need to install the required libraries. You can do that using [pip](#) inside your virtual environment. Install the libraries as follows:

Shell

```
(venv) $ python -m pip install dash==1.13.3 pandas==1.0.5
```

This command will install Dash and [pandas](#) in your virtual environment. You'll use specific versions of these packages to make sure that you have the same environment as the one used throughout this tutorial. In addition to Dash, pandas will help you handle reading and wrangling the data you'll use in your app.

Finally, you need some data to feed into your dashboard. You can download the data as well as the code you see throughout this tutorial by clicking the link below:

Get the Source Code: [Click here to get the source code you'll use](#) to learn about creating data visualization interfaces in Python with Dash in this tutorial.

Save the data as `avocado.csv` in the root directory of the project. By now, you should have a virtual environment with the required libraries and the data in the root folder of your project. Your project's structure should look like this:

```
avocado_analytics/  
|  
├─ venv/  
|  
└─ avocado.csv
```

You're good to go! Next, you'll build your first Dash application.

 [Remove ads](#)

How to Build a Dash Application

For development purposes, it's useful to think of the process of building a Dash application in two steps:

1. Define the looks of your application using the app's **layout**.
2. Use **callbacks** to determine which parts of your app are interactive and what they react to.

In this section, you'll learn about the layout, and in a later section, you'll learn [how to make your dashboard interactive](#). You'll start by setting up everything you need to initialize your application and then you'll define the layout of your app.

Initializing Your Dash Application

Create an empty file named `app.py` in the root directory of your project, then review the code of `app.py` in this section. To make it easier for you to copy the full code, you'll find the entire contents of `app.py` at the end of this section.

Here are the first few lines of `app.py`:

Python

```
1 import dash  
2 import dash_core_components as dcc  
3 import dash_html_components as html  
4 import pandas as pd  
5  
6 data = pd.read_csv("avocado.csv")  
7 data = data.query("type == 'conventional' and region == 'Albany'")  
8 data["Date"] = pd.to_datetime(data["Date"], format="%Y-%m-%d")  
9 data.sort_values("Date", inplace=True)  
10  
11 app = dash.Dash(__name__)
```

On lines 1 to 4, you import the required libraries: `dash`, `dash_core_components`, `dash_html_components`, and `pandas`. Each library provides a building block for your application:

- **dash** helps you initialize your application.
- **dash_core_components** allows you to create interactive components like graphs, dropdowns, or date ranges.
- **dash_html_components** lets you access HTML tags.
- **pandas** helps you read and organize the data.

On lines 6 to 9, you [read the data](#) and preprocess it for use in the dashboard. You filter some of the data because the current version of your dashboard isn't interactive, and the plotted values wouldn't make sense otherwise.

On line 11, you create an instance of the `Dash` class. If you've used [Flask](#) before, then initializing a `Dash` class may look familiar. In `Flask`, you usually initialize a WSGI application using `Flask(__name__)`. Similarly, for a `Dash` app, you use `Dash(__name__)`.

Defining the Layout of Your Dash Application

Next, you'll define the `layout` property of your application. This property dictates the look of your app. In this case, you'll use a heading with a description below it and two graphs. Here's how you define it:

Python

```

13 app.layout = html.Div(
14     children=[
15         html.H1(children="Avocado Analytics"),
16         html.P(
17             children="Analyze the behavior of avocado prices"
18             " and the number of avocados sold in the US"
19             " between 2015 and 2018",
20         ),
21         dcc.Graph(
22             figure={
23                 "data": [
24                     {
25                         "x": data["Date"],
26                         "y": data["AveragePrice"],
27                         "type": "lines",
28                     },
29                 ],
30                 "layout": {"title": "Average Price of Avocados"},
31             },
32         ),
33         dcc.Graph(
34             figure={
35                 "data": [
36                     {
37                         "x": data["Date"],
38                         "y": data["Total Volume"],
39                         "type": "lines",
40                     },
41                 ],
42                 "layout": {"title": "Avocados Sold"},
43             },
44         ),
45     ]
46 )

```

This code defines the `layout` property of the `app` object. This property determines the looks of your application using a tree structure made of Dash components.

Dash components come prepackaged in Python libraries. Some of them come with Dash when you install it. The rest you have to install separately. You'll see two sets of components in almost every app:

1. [Dash HTML Components](#) provides you with Python wrappers for HTML elements. For example, you could use this library to create elements such as paragraphs, headings, or lists.
2. [Dash Core Components](#) provides you with Python abstractions for creating interactive user interfaces. You can use it to create interactive elements such as graphs, sliders, or dropdowns.

On lines 13 to 20, you can see the Dash HTML components in practice. You start by defining the parent component, an `html.Div`. Then you add two more elements, a heading (`html.H1`) and a paragraph (`html.P`), as its children.

These components are equivalent to the `div`, `h1`, and `p` HTML tags. You can use the components' arguments to modify attributes or the content of the tags. For example, to specify what goes inside the `div` tag, you use the `children` argument in `html.Div`.

There are also other arguments in the components, such as `style`, `className`, or `id`, that refer to attributes of the HTML tags. You'll see how to use some of these properties to style your dashboard in the next section.

The part of the layout shown on lines 13 to 20 will get transformed into the following HTML code:

HTML

```
<div>
  <h1>Avocado Analytics</h1>
  <p>
    Analyze the behavior of avocado prices and the number
    of avocados sold in the US between 2015 and 2018
  </p>
  <!-- Rest of the app -->
</div>
```

This HTML code is rendered when you open your application in the browser. It follows the same structure as your Python code, with a `div` tag containing an `h1` and a `p` element.

On lines 21 to 24 in the layout code snippet, you can see the graph component from Dash Core Components in practice. There are two `dcc.Graph` components in the `app.layout`. The first one plots the average prices of avocados during the period of study, and the second plots the number of avocados sold in the United States during the same period.

Under the hood, Dash uses Plotly.js to generate graphs. The `dcc.Graph` components expect a [figure object](#) or a [Python dictionary](#) containing the plot's data and layout. In this case, you provide the latter.

Finally, these two lines of code help you run your application:

Python

```
48 | if __name__ == "__main__":
49 |     app.run_server(debug=True)
```

Lines 48 and 49 make it possible to run your Dash application locally using Flask's built-in server. The `debug=True` parameter from `app.run_server` enables the **hot-reloading** option in your application. This means that when you make a change to your app, it reloads automatically, without you having to restart the server.

Finally, here's the full version of `app.py`. You can copy this code in the empty `app.py` you created earlier.

app.py

Show/Hide

This is the code for a bare-bones dashboard. It includes all the snippets of code you reviewed earlier in this section.

Now it's time to run your application. Open a terminal inside your project's root directory and in the project's virtual environment. Run `python app.py`, then go to `http://localhost:8050` using your preferred browser.

It's ALIVE! Your dashboard should look like this:



The good news is that you now have a working version of your dashboard. The bad news is that there’s still some work to do before you can show this to others. The dashboard is far from visually pleasing, and you still need to add some interactivity to it.

But don’t worry—you’ll learn how to fix these issues in the next sections.

 [Remove ads](#)

Style Your Dash Application

Dash provides you with a lot of flexibility to customize the look of your application. You can use your own CSS or JavaScript files, set a **favicon** (small icon shown on the web browser), and embed images, among other advanced options.

In this section, you’ll learn how to apply custom styles to components, and then you’ll style the dashboard you built in the previous section.

How to Apply a Custom Style to Your Components

You can style components in two ways:

1. Using the `style` argument of individual components
2. Providing an external CSS file

Using the `style` argument to customize your dashboard is straightforward. This argument takes a Python dictionary with key-value pairs consisting of the names of CSS properties and the values you want to set.

Note: When specifying CSS properties in the `style` argument, you should use `mixedCase` syntax instead of hyphen-separated words. For example, to change the background color of an element, you should use `backgroundColor` and not `background-color`.

If you wanted to change the size and color of the `H1` element in `app.py`, then you could set the element’s `style` argument as follows:

Python

```
html.H1(  
    children="Avocado Analytics",  
    style={"fontSize": "48px", "color": "red"},  
)
```

Here, you provide to `style` a dictionary with the properties and the values you want to set for them. In this case, the specified style is to have a red heading with a font size of 48 pixels.

The downside of using the `style` argument is that it doesn't scale well as your codebase grows. If your dashboard has multiple components that you want to look the same, then you'll end up repeating a lot of your code. Instead, you can use a custom CSS file.

If you want to include your own local CSS or [JavaScript](#) files, then you need to create a folder called `assets/` in the root directory of your project and save the files you want to add there. By default, Dash automatically serves any file included in `assets/`. This will also work for adding a favicon or embedding images, as you'll see in a bit.

Then you can use the `className` or `id` arguments of the components to adjust their styles using CSS. These arguments correspond with the [class](#) and [id](#) attributes when they're transformed into HTML tags.

If you wanted to adjust the font size and text color of the `H1` element in `app.py`, then you could use the `className` argument as follows:

Python

```
html.H1(
    children="Avocado Analytics",
    className="header-title",
),
```

Setting the `className` argument will define the class attribute for the `H1` element. You could then use a CSS file in the `assets` folder to specify how you want it to look:

CSS

```
.header-title {
    font-size: 48px;
    color: red;
}
```

You use a [class selector](#) to format the heading in your CSS file. This selector will adjust the heading format. You could also use it with other element that needs to share the format by setting `className="header-title"`.

Next, you'll style your dashboard.

 [Remove ads](#)

How to Improve the Looks of Your Dashboard

You just covered the basics of styling in Dash. Now, you'll learn how to customize your dashboard's looks. You'll make these improvements:

- Add a favicon and title to the page
- Change the font family of your dashboard
- Use an external CSS file to style Dash components

You'll start by learning how to use external assets in your application. That will allow you to add a favicon, a custom font family, and a CSS style sheet. Then you'll learn how to use the `className` argument to apply custom styles to your Dash components.

Adding External Assets to Your Application

Create a folder called `assets/` in your project's root directory. [Download a favicon](#) from the [Twemoji open source project](#) and save it as `favicon.ico` in `assets/`. Finally, create a CSS file in `assets/` called `style.css` and the code in the collapsible section below.

style.css

Show/Hide

The `assets/` file contains the styles you'll apply to components in your application's layout. By now, your project structure should look like this:


```
avocado_analytics/  
|  
├── assets/  
|   ├── favicon.ico  
|   └── style.css  
|  
├── venv/  
|  
├── app.py  
└── avocado.csv
```

Once you start the server, Dash will automatically serve the files located in `assets/`. You include two files in `assets/`: `favicon.ico` and `style.css`. For setting a default favicon, you don't have to take any additional steps. For using the styles you defined in `style.css`, you'll need to use the `className` argument in Dash components.

`app.py` requires a few changes. You'll include an external style sheet, add a title to your dashboard, and style the components using the `style.css` file. Review the changes below. Then, in the last part of this section, you'll find the full code for your updated version of `app.py`.

Here's how you include an external style sheet and add a title to your dashboard:

Python

```
11 external_stylesheets = [  
12     {  
13         "href": "https://fonts.googleapis.com/css2?"  
14         "family=Lato:wght@400;700&display=swap",  
15         "rel": "stylesheet",  
16     },  
17 ]  
18 app = dash.Dash(__name__, external_stylesheets=external_stylesheets)  
19 app.title = "Avocado Analytics: Understand Your Avocados!"
```

On lines 11 to 18, you specify an external CSS file, a font family, that you want to load in your application. External files are added to the head tag of your application and loaded before the body of your application loads. You use the `external_stylesheets` argument for adding external CSS files or `external_scripts` for external JavaScript files like Google Analytics.

On line 19, you set the title of your application. This is the text that appears in the title bar of your web browser, in Google's search results, and in social media cards when you share your site.

Customizing the Styles of Components

To use the styles in `style.css`, you'll need to use the `className` argument in Dash components. The code below adds a `className` with a corresponding class selector to each of the components that compose the header of your dashboard:

Python

```
21 app.layout = html.Div(  
22     children=[  
23         html.Div(  
24             children=[  
25                 html.P(children="🥑", className="header-emoji"),  
26                 html.H1(  
27                     children="Avocado Analytics", className="header-title"  
28                 ),  
29                 html.P(  
30                     children="Analyze the behavior of avocado prices"  
31                     " and the number of avocados sold in the US"  
32                     " between 2015 and 2018",  
33                     className="header-description",  
34                 ),  
35             ],  
36             className="header",  
37         ),
```

On lines 21 to 37, you can see that there have been two changes to initial version of the dashboard:

1. There's a new paragraph element with an avocado emoji that will serve as logo.
2. There's a `className` argument in each component. These class names should match a class selector in `style.css`, which will define the looks of each component.

For example, the header-description class assigned to the paragraph component starting with "Analyze the behavior of avocado prices" has a corresponding selector in `style.css`:

CSS

```
29 .header-description {
30     color: #CFCFCF;
31     margin: 4px auto;
32     text-align: center;
33     max-width: 384px;
34 }
```

Lines 29 to 34 of `style.css` define the format for the header-description class selector. These will change the color, margin, alignment, and maximum width of any component with `className="header-description"`. All the components have corresponding class selectors in the CSS file.

The other significant change is in the graphs. Here's the new code for the price chart:

Python

```
38 html.Div(
39     children=[
40         html.Div(
41             children=dcc.Graph(
42                 id="price-chart",
43                 config={"displayModeBar": False},
44                 figure={
45                     "data": [
46                         {
47                             "x": data["Date"],
48                             "y": data["AveragePrice"],
49                             "type": "lines",
50                             "hovertemplate": "%{y:.2f}"
51                             "<extra></extra>",
52                         },
53                     ],
54                     "layout": {
55                         "title": {
56                             "text": "Average Price of Avocados",
57                             "x": 0.05,
58                             "xanchor": "left",
59                         },
60                         "xaxis": {"fixedrange": True},
61                         "yaxis": {
62                             "tickprefix": "$",
63                             "fixedrange": True,
64                         },
65                         "colorway": ["#17B897"],
66                     },
67                 },
68             ),
69             className="card",
70         ),
```

In this code, you define a `className` and a few customizations for the `config` and `figure` parameters of your chart. Here are the changes:

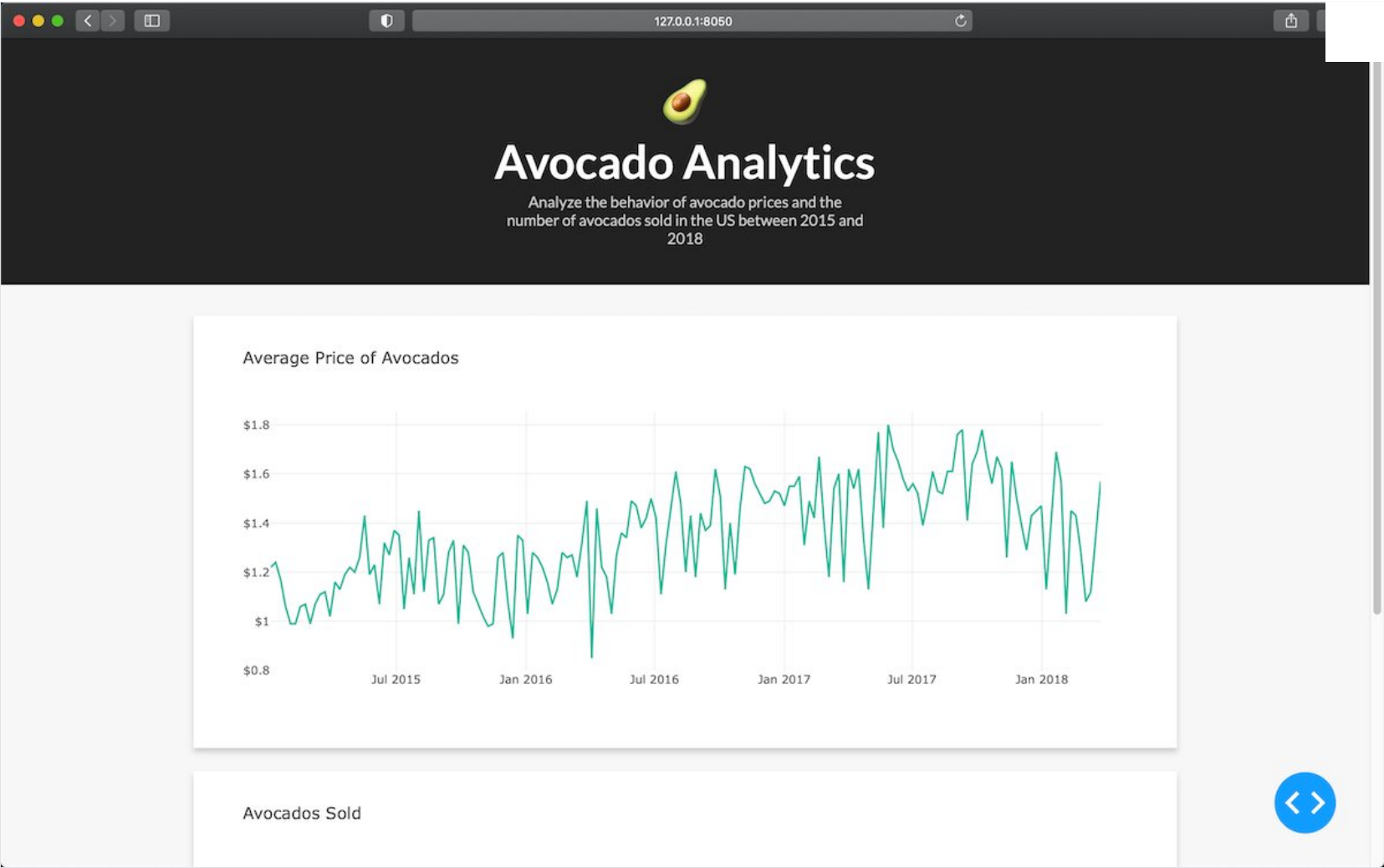
- **Line 43:** You remove the floating bar that Plotly shows by default.
- **Lines 50 and 51:** You set the hover template so that when users hover over a data point, it shows the price in dollars. Instead of 2.5, it'll show as \$2.5.
- **Lines 54 to 66:** You adjust the axis, the color of the figure, and the title format in the layout section of the graph.
- **Line 69:** You wrap the graph in an `html.Div` with a "card" class. This will give the graph a white background and add a small shadow below it.

There are similar adjustments to the sales and volume charts. You can see those in the full code for the updated `app.py` in the collapsible section below.

app.py

Show/Hide

This is the updated version of `app.py`. It has the required changes in the code to add a favicon and a page title, update the font family, and use an external CSS file. After these changes, your dashboard should look like this:



In the next section, you’ll learn how to add interactive components to your dashboard.

[Remove ads](#)

Add Interactivity to Your Dash Apps Using Callbacks

In this section, you’ll learn how to add interactive elements to your dashboard.

Dash’s interactivity is based on a [reactive programming](#) paradigm. This means that you can link components with elements of your app that you want to update. If a user interacts with an input component like a dropdown or a range slider, then the output, such as a graph, will react automatically to the changes in the input.

Now let’s make your dashboard interactive. This new version of your dashboard will allow the user to interact with the following filters:

- Region
- Type of avocado
- Date range

Start by replacing your local `app.py` with the new version in the collapsible section below.

app.py

Show/Hide

Next, replace `style.css` with the code in the collapsible section below.

style.css

Show/Hide

Now you’re ready to start adding interactive components to your application!

How to Create Interactive Components

First, you'll learn how to create components that users can interact with. For that, you'll include a new `html.Div` above your charts. It'll include two dropdowns and a date range selector that the user can use to filter the data and update the graphs.

Here's how that looks in `app.py`:

Python

```

24  html.Div(
25      children=[
26          html.Div(
27              children=[
28                  html.Div(children="Region", className="menu-title"),
29                  dcc.Dropdown(
30                      id="region-filter",
31                      options=[
32                          {"label": region, "value": region}
33                          for region in np.sort(data.region.unique())
34                      ],
35                      value="Albany",
36                      clearable=False,
37                      className="dropdown",
38                  ),
39              ]
40          ),
41          html.Div(
42              children=[
43                  html.Div(children="Type", className="menu-title"),
44                  dcc.Dropdown(
45                      id="type-filter",
46                      options=[
47                          {"label": avocado_type, "value": avocado_type}
48                          for avocado_type in data.type.unique()
49                      ],
50                      value="organic",
51                      clearable=False,
52                      searchable=False,
53                      className="dropdown",
54                  ),
55              ],
56          ),
57          html.Div(
58              children=[
59                  html.Div(
60                      children="Date Range",
61                      className="menu-title"
62                  ),
63                  dcc.DatePickerRange(
64                      id="date-range",
65                      min_date_allowed=data.Date.min().date(),
66                      max_date_allowed=data.Date.max().date(),
67                      start_date=data.Date.min().date(),
68                      end_date=data.Date.max().date(),
69                  ),
70              ]
71          ),
72      ],
73      className="menu",
74  ),

```

On lines 24 to 74, you define an `html.Div` on top of your graphs consisting of two dropdowns and a date range selector. It will serve as a menu that the user will use to interact with the data:

The image shows a user interface with three components arranged horizontally. Each component has a title in teal text above it. The first component, titled 'Region', is a dropdown menu with 'Albany' selected. The second component, titled 'Type', is a dropdown menu with 'organic' selected. The third component, titled 'Date Range', is a date range selector showing '01/04/2015' followed by a right-pointing arrow and '03/25/2018'.

The first component in the menu is the Region dropdown. Here's the code for that component:

Python

```
41 html.Div(
42     children=[
43         html.Div(children="Region", className="menu-title"),
44         dcc.Dropdown(
45             id="region-filter",
46             options=[
47                 {"label": region, "value": region}
48                 for region in np.sort(data.region.unique())
49             ],
50             value="Albany",
51             clearable=False,
52             className="dropdown",
53         ),
54     ],
55 ),
```

On lines 41 to 55, you define the dropdown that users will use to filter the data by region. In addition to the title, it has a `dcc.Dropdown` component. Here's what each of the parameters means:

- **id** is the identifier of this element.
- **options** is the options shown when the dropdown is selected. It expects a dictionary with labels and values.
- **value** is the default value when the page loads.
- **clearable** allows the user to leave this field empty if set to `True`.
- **className** is a class selector used for applying styles.


The Type and Date Range selectors follow the same structure as the Region dropdown. Feel free to review them on your own.

Next, take a look at the `dcc.Graphs` components:

Python

```
90 html.Div(
91     children=[
92         html.Div(
93             children=dcc.Graph(
94                 id="price-chart", config={"displayModeBar": False},
95             ),
96             className="card",
97         ),
98         html.Div(
99             children=dcc.Graph(
100                 id="volume-chart", config={"displayModeBar": False},
101             ),
102             className="card",
103         ),
104     ],
105     className="wrapper",
106 ),
```

On lines 90 to 106, you define the `dcc.Graph` components. You may have noticed that, compared to the previous version of the dashboard, the components are missing the `figure` argument. That's because the `figure` argument will now be generated by a [callback function](#) using the inputs the user sets using the Region, Type, and Date Range selectors.

 [Remove ads](#)

How to Define Callbacks

You've defined how the user will interact with your application. Now you need to make your application react to user interactions. For that, you'll use **callback functions**.

Dash's callback functions are regular Python functions with an `app.callback` [decorator](#). In Dash, when an input changes, a callback function is triggered. The function performs some predetermined operations, like filtering a dataset, and returns an output to the application. In essence, callbacks link inputs and outputs in your app.

Here's the callback function used for updating the graphs:

Python

```

111 @app.callback(
112     [Output("price-chart", "figure"), Output("volume-chart", "figure")],
113     [
114         Input("region-filter", "value"),
115         Input("type-filter", "value"),
116         Input("date-range", "start_date"),
117         Input("date-range", "end_date"),
118     ],
119 )
120 def update_charts(region, avocado_type, start_date, end_date):
121     mask = (
122         (data.region == region)
123         & (data.type == avocado_type)
124         & (data.Date >= start_date)
125         & (data.Date <= end_date)
126     )
127     filtered_data = data.loc[mask, :]
128     price_chart_figure = {
129         "data": [
130             {
131                 "x": filtered_data["Date"],
132                 "y": filtered_data["AveragePrice"],
133                 "type": "lines",
134                 "hovertemplate": "$%{y:.2f}<extra></extra>",
135             },
136         ],
137         "layout": {
138             "title": {
139                 "text": "Average Price of Avocados",
140                 "x": 0.05,
141                 "xanchor": "left",
142             },
143             "xaxis": {"fixedrange": True},
144             "yaxis": {"tickprefix": "$", "fixedrange": True},
145             "colorway": ["#17B897"],
146         },
147     }
148
149     volume_chart_figure = {
150         "data": [
151             {
152                 "x": filtered_data["Date"],
153                 "y": filtered_data["Total Volume"],
154                 "type": "lines",
155             },
156         ],
157         "layout": {
158             "title": {
159                 "text": "Avocados Sold",
160                 "x": 0.05,
161                 "xanchor": "left",
162             },
163             "xaxis": {"fixedrange": True},
164             "yaxis": {"fixedrange": True},
165             "colorway": ["#E12D39"],
166         },
167     }
168     return price_chart_figure, volume_chart_figure

```

On lines 111 to 119, you define the inputs and outputs inside the `app.callback` decorator.

First, you define the outputs using `Output` objects. These objects take two arguments:

1. The identifier of the element that they'll modify when the function executes

2. The property of the element to be modified

For example, `Output("price-chart", "figure")` will update the `figure` property of the `"price-chart"` element.

Then you define the inputs using `Input` objects. They also take two arguments:

1. The identifier of the element they'll be watching for changes
2. The property of the watched element that they should take when a change happens

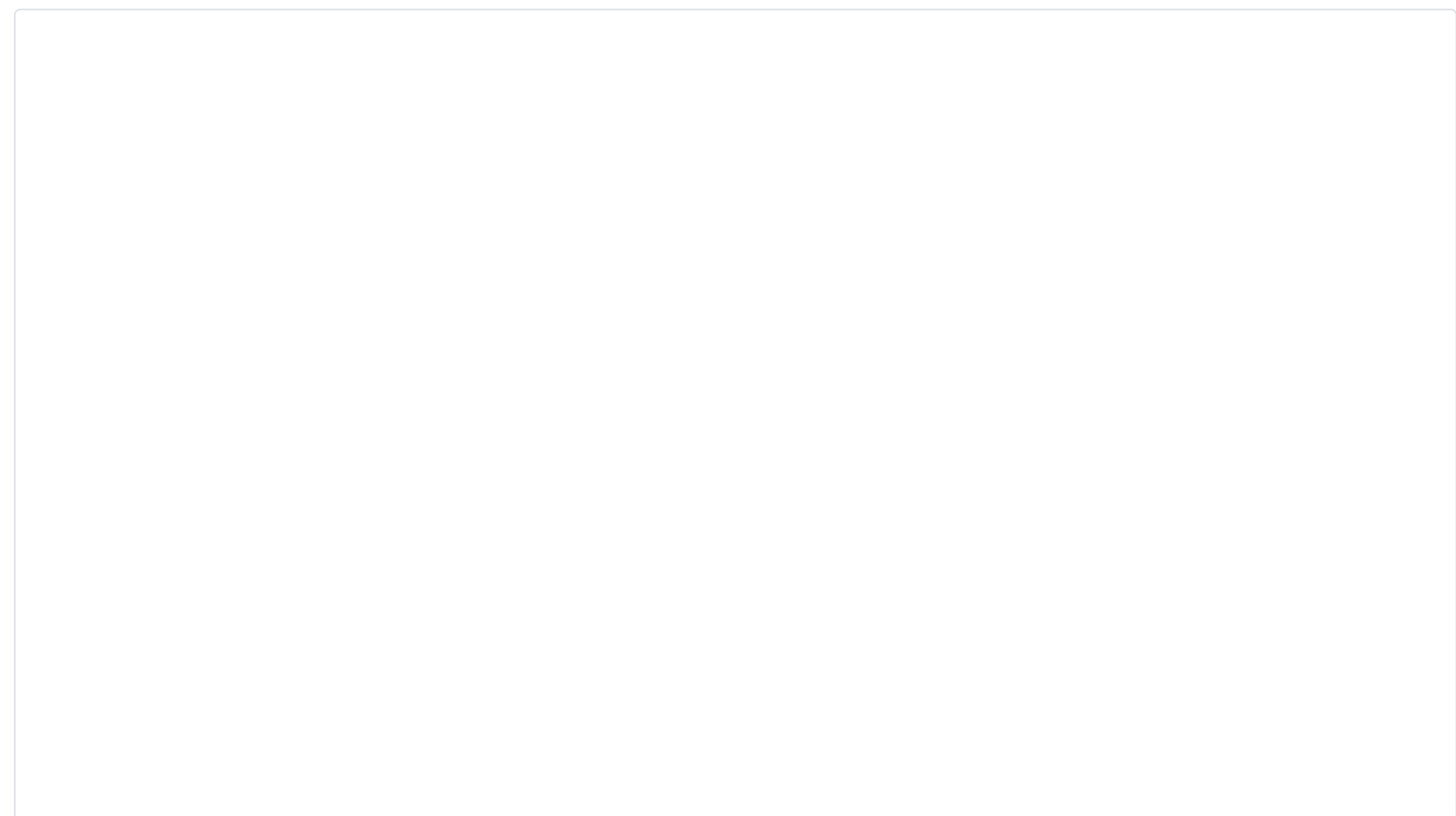
So, `Input("region-filter", "value")` will watch the `"region-filter"` element for changes and will take its `value` property if the element changes.

Note: The `Input` object discussed here is imported from `dash.dependencies`. Be careful not to confuse it with the component coming from `dash_core_components`. These objects are not interchangeable and have different purposes.

On line 120, you define the function that will be applied when an input changes. One thing to notice here is that the arguments of the function will correspond with the order of the `Input` objects supplied to the callback. There's no explicit relationship between the names of the arguments in the function and the values specified in the `Input` objects.

Finally, on lines 121 to 164, you define the body of the function. In this case, the function takes the inputs (region, type of avocado, and date range), filters the data, and generates the figure objects for the price and volume charts.

That's all! If you've followed along to this point, then your dashboard should look like this:



Way to go! That's the final version of your dashboard. In addition to making it look beautiful, you also made it interactive. The only missing step is making it public so you can share it with others.

Deploy Your Dash Application to Heroku

You're done building your application, and you have a beautiful, fully interactive dashboard. Now you'll learn how to deploy it.

Dash apps are Flask apps, so both share the same [deployment options](#). In this section, you'll deploy your app on Heroku.

Before you get started, make sure you've installed the [Heroku command-line interface \(CLI\)](#) and [Git](#). You can verify that both exist in your system by running these commands at a command prompt (Windows) or at a terminal (macOS, Linux):

Shell

```
$ git --version
git version 2.21.1 (Apple Git-122.3)
$ heroku --version
heroku/7.42.2 darwin-x64 node-v12.16.2
```

The output may change a bit depending on your operating system and the version you have installed, but you shouldn't get an error.

Let's get to it!

First, there's a small change you need to make in `app.py`. After you initialize the app on line 18, add a new [variable](#) called `server`:

Python

```
18 app = dash.Dash(__name__, external_stylesheets=external_stylesheets)
19 server = app.server
```

This addition is necessary to run your app using a [WSGI server](#). It's not advisable to use Flask's built-in server in production since it won't be able to handle much traffic.

Next, in the project's root directory, create a file called `runtime.txt` where you'll specify a Python version for your Heroku app:

Text

```
python-3.8.6
```

When you deploy your app, Heroku will automatically detect that it's a Python application and will use the correct buildpack. If you also provide a `runtime.txt`, then it'll pin down the Python version that your app will use.

Next, create a `requirements.txt` file in the project's root directory where you'll copy the libraries required to set up your Dash application on a web server:

Python Requirements

```
dash==1.13.3
pandas==1.0.5
gunicorn==20.0.4
```

You may have noticed that there's a package in `requirements.txt` you haven't seen until now: `gunicorn`. Gunicorn is a WSGI [HTTP](#) server that is frequently used for deploying Flask apps to production. You'll use it to deploy your dashboard.

Now create a file named `Procfile` with the following content:

Shell

```
web: gunicorn app:server
```

This file tells the Heroku app what commands should be executed to start your app. In this case, it starts a `gunicorn` server for your dashboard.

Next, you'll need to initialize a [Git](#) repository. To do that, go to your project's root directory and execute the following command:

Shell

```
$ git init
```

This will start a `Git` repository in `avocado_analytics/`. It'll start tracking all the changes you make to the files in that directory.

However, there are files you don't want to track using `Git`. For example, you usually want to remove Python compiled files, the contents of your virtual environment folder, or metadata files such as `.DS_Store`.

To avoid tracking unnecessary files, create a file called `.gitignore` in the root directory. Then copy the following content in it:

Text

```
venv
*.pyc
.DS_Store # Only if you are using macOS
```

This will make sure your repository doesn't track unnecessary files. Now commit your project files:

Shell

```
$ git add .
$ git commit -m 'Add dashboard files'
```

Before the final step, make sure you have everything in place. Your project's structure should look like this:

```
avocado_analytics/
├── assets/
│   ├── favicon.ico
│   └── style.css
├── venv/
├── app.py
├── avocado.csv
├── Procfile
├── requirements.txt
└── runtime.txt
```

Finally, you need to create an app in Heroku, push your code there using Git, and start the app in one of Heroku's free server options. You can do that by running the following commands:

Shell

```
$ heroku create APP-NAME # Choose a name for your app
$ git push heroku master
$ heroku ps:scale web=1
```

The first command will create a new application on Heroku and an associated Git repository. The second will push the changes to that repository, and the third will start your app in one of Heroku's free server options.

That's it! You've built and deployed your dashboard. Now you just need to access it to share it with your friends. To access your app, copy `https://APP-NAME.herokuapp.com/` in your browser and replace `APP-NAME` with the name you defined in the previous step.

If you're curious, take a look at a [sample app](#).

 [Remove ads](#)

Conclusion

Congratulations! You just built, customized, and deployed your first dashboard using Dash. You went from a bare-bones dashboard to a fully interactive one deployed on Heroku.

With this knowledge, you can use Dash to build analytical applications to share with others. As more companies put more weight on the use of data, knowing how to use Dash will increase the impact you have in your workplace. What used to be a task only experts could perform, you can now do in an afternoon.

In this tutorial, you've learned:

- How to **create a dashboard** using Dash
- How to **customize** the styling of your Dash application
- How to make your app **interactive** by using Dash components

- What **callbacks** are and how you can use them to create interactive applications
- How to **deploy** your application on Heroku

Now you’re ready to develop new Dash applications. Find a dataset, think of some exciting visualizations, and build another dashboard!

You can download the source code, data, and resources for the sample applications you made in this tutorial by clicking the link below:



Get the Source Code: [Click here to get the source code you’ll use](#) to learn about creating data visualization interfaces in Python with Dash in this tutorial.

Mark as Completed

🔖

▶ Watch Now

This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Data Visualization Interfaces in Python With Dash](#)

 Python Tricks 


Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »


About **Dylan Castillo**




Dylan is a Data Scientist and self-taught developer specialized in Natural Language Processing (NLP). He has experience working on large-scale Machine Learning projects and enjoys writing about data-related topics.

» [More about Dylan](#)


Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:



Aldren



David



Geir Arne

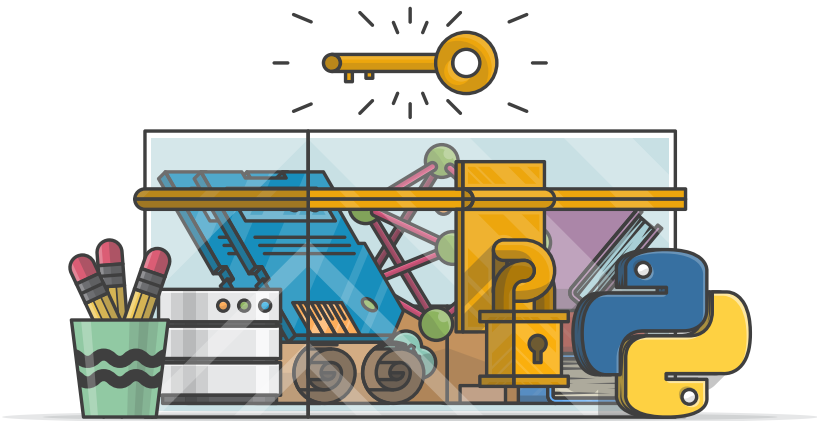


[Joanna](#)



[Jacob](#)

Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

[Level Up Your Python Skills »](#)

What Do You Think?

[Tweet](#) [Share](#) [Email](#)

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won’t make the cut here.

What’s your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: [data-science](#) [intermediate](#)

Recommended Video Course: [Data Visualization Interfaces in Python With Dash](#)

— FREE Email Series —



```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
```

```
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Email...

Get Python Tricks »

No spam. Unsubscribe any time.

All Tutorial Topics

- advancedapibasicsbest-practicescommunitydatabasesdata-science
- devopsdjangodockerflaskfront-endgamedevguiintermediate
- machine-learningprojectspythontestingtoolsweb-devweb-scraping

Table of Contents

- [What Is Dash?](#)
- [Get Started With Dash in Python](#)
- [Style Your Dash Application](#)
- [Add Interactivity to Your Dash Apps Using Callbacks](#)
- [Deploy Your Dash Application to Heroku](#)
- [Conclusion](#)

Mark as Completed

Tweet Share Email

Recommended Video Course

[Data Visualization Interfaces in Python With Dash](#)

[Remove ads](#)