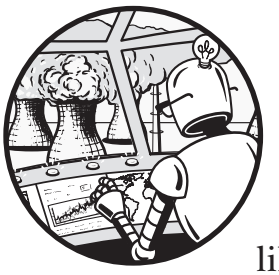
4

FIRST DASH APP



In this chapter you'll build your first Dash app. We'll analyze the number of Twitter likes received by 16 chosen celebrities since 2011. You can download the data with the book's resources at *https://nostarch.com/python-dash*. The type of analysis we'll do is common in the field of social media analytics, typically used to better understand audience behavior, the effectiveness of posts, and the overall performance of an account.

This first dashboard will plot the number of likes per tweet. Once you master this simple plotting process with Dash, you'll be able to scale your skills to plot bigger and more complex data in other areas: Instagram post views, Facebook profile visits, LinkedIn post click-through rates, and YouTube video performance.

This chapter should give you sufficient knowledge of Dash to create your own dashboard app. You'll learn how to incorporate data into your

app, manage numerous dashboard app components, build basic charts such as line charts, and add interactive capabilities to your dashboard through the callback decorator. First, let's download the code and run the app to see what it does.

Setting Up the Project

Open PyCharm, create a new project folder, and call it "my-first-app" (the project name should be the suffix text after the last backslash in the Location field of the name). Set up your virtual environment using the standard Virtualenv.

NOTE *The code in this chapter assumes you're using a Python IDE, such as PyCharm. If you don't have an IDE installed and a virtual environment set, refer back to Chapter 2 and complete your Python setup. If you're using a different coding environment, just adapt the instructions here to your environment. The code in this chapter also requires Python 3.6 or higher.*

Next, you need to download this chapter's dashboard app files into your project folder. Instead of cloning the repository as we did in Chapter 2, we'll download the ZIP files directly. It's worth trying various ways to set up a project because you'll probably stumble upon some projects that are not directly available as Git repositories. To use the ZIP files, go to the GitHub repository, *https://github.com/DashBookProject/Plotly-Dash*, click **Code**, and then click **Download ZIP**, as shown in Figure 4-1.

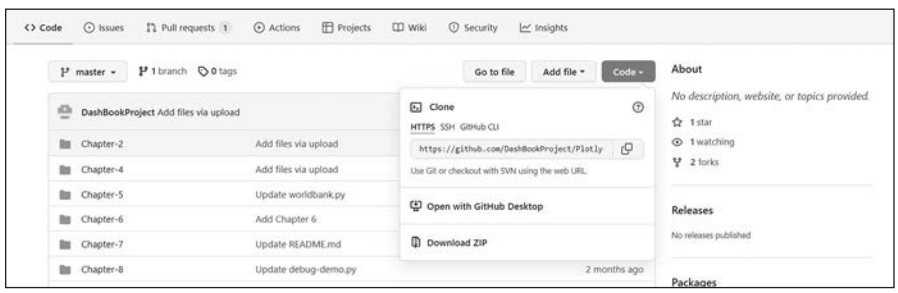


Figure 4-1: Downloading the app code from GitHub

Once you have the *Plotly-Dash-master.zip* file on your computer, open it and go into the *Chapter-4* folder. Copy all the files from that folder into your recently created *my-first-app* project folder. The project folder should have files in the following structure:

```
– my-first-app
|--assets
     --mystyles.css
|--tweets.csv
|--twitter_app.py
```

The *assets* folder will hold the CSS script. The *tweets.csv* file holds the data we'll use, and *twitter_app.py* is the main app file we'll use to run the app.

We'll now install the necessary libraries in our virtual environment. Go to the Terminal tab in the bottom part of the PyCharm page, shown in Figure 4-2.

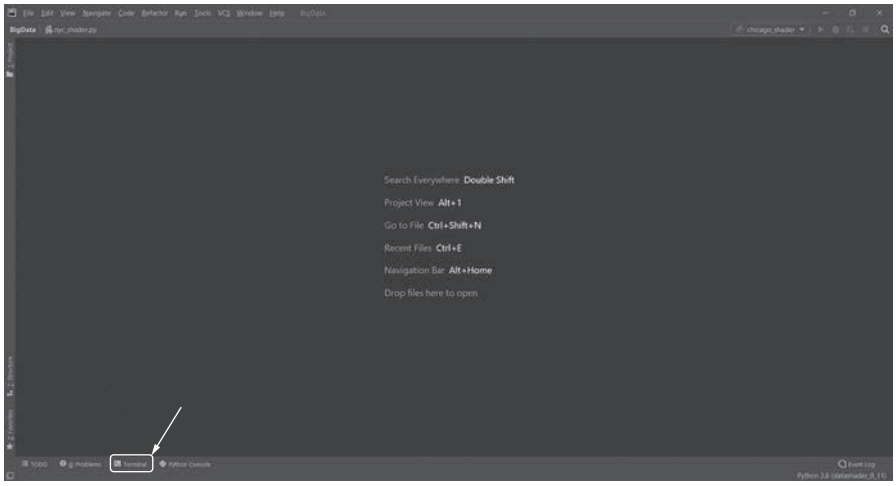


Figure 4-2: Installing the libraries

Enter and execute the following lines of code to install pandas and Dash (the Plotly package is automatically installed with Dash, so there is no need to install Plotly, and the NumPy package is automatically installed with pandas):

```
$ pip install pandas
$ pip install dash
```

To check that the libraries are installed correctly, enter:

```
$ pip list
```

This will create a list of all the Python packages currently in your virtual environment. If they're all listed, you can move on. Note that all dependencies of pandas and Dash will also be listed, so you might see many more libraries than just the two you installed.

Next, open *twitter_app.py* inside PyCharm and run the script. You should see the following message:

```
* Serving Flask app "twitter_app" (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
* Debug mode: on
Dash is running on http://127.0.0.1:8050/
```

The warning just reminds us that the app is in a development server and it is completely normal. To open your app, click the HTTP link or copy and paste it into your browser's address bar.

Congratulations! You should now see your first Dash app, which should look like Figure 4-3.

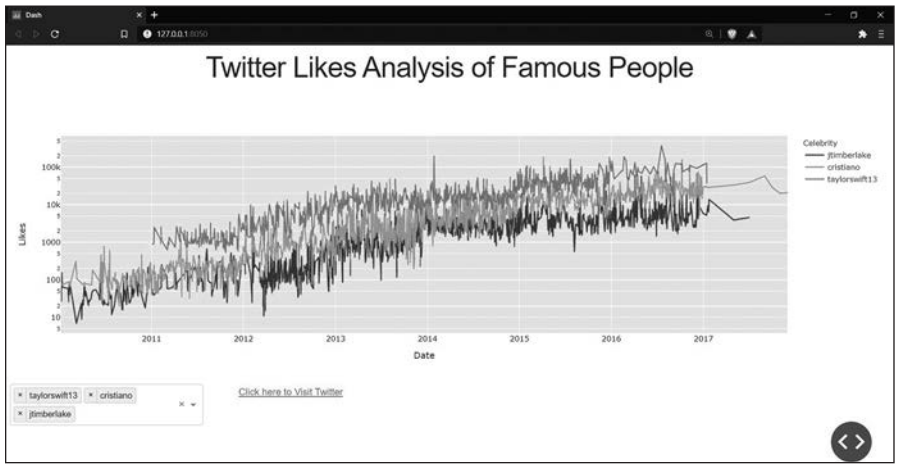


Figure 4-3: The Twitter Likes Analysis app

Have fun! Play around with your dashboard app. Change the dropdown values, click the links, click on the graph legend, and zoom in to a certain date range by holding down the mouse's left-click button and dragging the mouse. See what information you can deduce. Now let's take a look at the code of the app.

Most Dash apps have a similar code layout:

1. Import the necessary Python libraries.
2. Read in the data.
3. Assign a stylesheet to describe how the app should be displayed.
4. Build the app layout that will define how to display all the elements.
5. Create the callbacks to enable interactivity between the app components.

Because Dash apps mostly follow this outline, we'll go through the code in this order.

Importing the Libraries

Let's first look at the libraries we'll use, shown in Listing 4-1.

```
import pandas as pd
import plotly.express as px
from dash import Dash, dcc, html, Input, Output
```

Listing 4-1: The import section of twitter_app.py

We first import pandas to handle the data. We then import Plotly, a popular Python visualization library. There are two main ways to create graphs in Plotly. We're using *Plotly Express*, a high-level interface for creating graphs in single-function calls, with very few lines of code. It has enough features to allow you to build graphs seamlessly and quickly, and is the easier of the two to use for simpler apps.

The alternative is *Plotly Graph Objects*, a low-level interface for creating graphs from the bottom up. When using Graph Objects, you need to define the data, layout, and, at times, frames, all of which make the graph-building process more involved. That said, its full set of features allows you to customize your graphs in ways that add much richness to them, so you might want to use Plotly Graph Objects once you've mastered Dash basics and you have more complicated graphs to build. We'll use Plotly Express in most cases and revert to Graph Objects in more complex situations.

Next, we import some Dash libraries to handle components and dependencies. *Components* are the building blocks that can be combined to create rich, complex interfaces for your users, such as dropdown menus, range sliders, and radio buttons. Dash comes bundled with two key component libraries maintained by Plotly: dash-html-components (HTML) and dash-core-components (DCC). The *dash-html-components* library contains structural elements such as headings and dividers that style and position elements on the page, while *dash-core-components* provides core functionality for your app, such as user input fields and figures.

Data Management

In this app, we're using a CSV spreadsheet as our data source. To use the data, we need to read it into memory via pandas, but before that we have to *clean* the data. This means we prepare the data for analysis and plotting by doing things like standardizing capitalization of strings and formats of time, stripping whitespace, and adding nulls for missing values. When the data is *dirty*, it's often unorganized and might contain missing values. If you try to use dirty data, the plot may not work, the analysis is likely to be inaccurate, and you'll find filtering difficult. Cleaning the data ensures that it is readable, presentable, and plottable.

Listing 4-2 shows the data management section of the code.

```
❶ df = pd.read_csv("tweets.csv")
df["name"] = pd.Series(df["name"]).str.lower()
df["date_time"] = pd.to_datetime(df["date_time"])
df = (
   df.groupby([df["date_time"].dt.date, "name"])[
      ["number_of_likes", "number_of_shares"]
   ]
   .mean()
   .astype(int)
)
df = df.reset_index()
```

Listing 4-2: The data management section of twitter_app.py

At ❶ we take the CSV spreadsheet and read it into a pandas DataFrame called `df`. The DataFrame at the beginning of a Dash app is commonly referred to as a *global DataFrame* and the data is a *global variable* (*global* means the object is declared outside a function, meaning it's accessible throughout the app).

To clean the data, we change the strings of the celebrity `name` column to lowercase so that we can readily compare them; we convert the `date_time` column into a date recognizable by pandas; and we group the data by `date_time` and `name` so that each row has a unique date stamp and name. If we did not group the data this way, we would end up with multiple rows with the same date and name, which would create a messy line chart that's impossible to read.

To check the data, add the following line of code to the script, right after `df = df.reset_index()`:

```
print(df.head())
```

Once you run the script anew, you should see something like the following inside the Python terminal:

```
      date_time      name           number_of_likes    number_of_shares
0     2010-01-06     selenagomez    278                695
1     2010-01-07     jtimberlake    62                 189
2     2010-01-07     selenagomez    201                630
3     2010-01-08     jtimberlake    27                 107
4     2010-01-08     selenagomez    349                935
```

As you can see, the result is a neat pandas DataFrame with rows of data that represent the average number of likes and shares per celebrity, per day.

It's always a good practice to read in and prepare your data at the beginning of the app because reading data can be a memory-expensive task; by inserting the data at the beginning, you ensure that the app loads the data into memory only once and does not repeat this process every time a user interacts with the dashboard.

Layout and Styling

The next step is to manage the layout and styling of the app components, such as the title, graph, and dropdown menus. We'll learn more about the components in "Dash Components" later in this chapter; here we'll just focus on the layout section.

In a Dash app, the *layout* refers to the alignment of the components within the app. The *style* refers to how the elements look, such as the color, size, spacing, and other properties (known in Dash as *props*). Styling the app allows for a more customized, professional presentation. Without styling, you could end up with an app like the one shown in Figure 4-4, where the title is not centered, the dropdown field stretches over the whole page, and the links are glued to the dropdown with no space in between them.

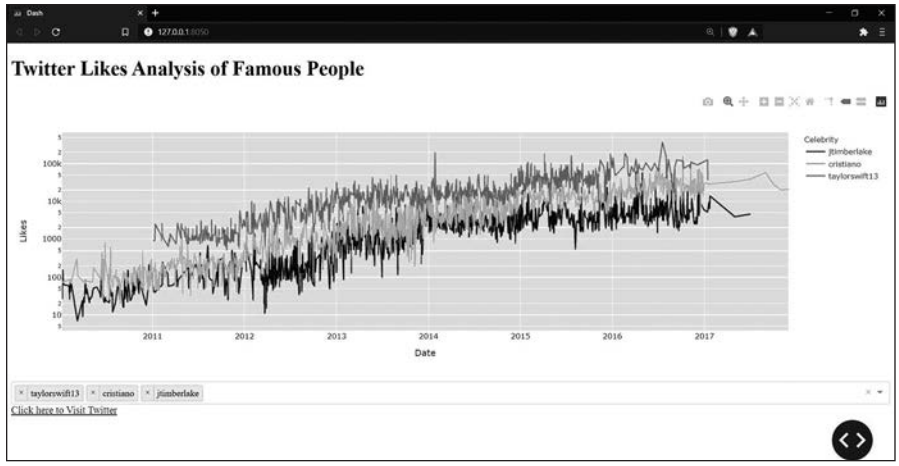


Figure 4-4: The Twitter Likes Analysis app without proper layout and styling

Alignment

Dash apps are web-based, so they use the standard language of web pages: HTML (HyperText Markup Language). Luckily, Dash includes the Dash HTML Components module, which converts Python to HTML, meaning we can use Python to write our HTML.

One of the most essential components of HTML is the `Div`, short for *division*, which is simply a container for other elements and a way to group elements together. Every component we use in a Dash app will be contained inside a `Div`, and a `Div` can contain multiple components. We build the `Div`, then style it to tell the web browser exactly where to position it and how much space it should take up.

Say we're creating a dashboard app with three dropdown menus, represented by the keyword `Dropdown`, as in Listing 4-3.

```
app.layout = html.Div([
   html.Div(dcc.Dropdown()),
   html.Div(dcc.Dropdown()),
   html.Div(dcc.Dropdown()),
])
```

Listing 4-3: Example `Div` code (not part of the main app)

The line `app.layout` creates a layout for this Dash app. Everything related to the layout must be placed within `app.layout`. We then create a `div` that contains three dropdown menus.

A `Div` by default will take up the full width of the parent container, meaning it's assumed to be one big cell that takes up the width of the page. As it is, the first `Dropdown` will appear in the top left and fill the whole page from left to right. The second `Dropdown` will appear right below the first `Dropdown` and fill the whole width of the page as well, and so on with the third `Dropdown`. In other words, each `Div` will take up the full width of the page and force neighboring elements onto a new line. To best control how much space a `Div`

is allocated, we should define the web page as a grid of rows and columns and place the Divs within a specific cell inside that grid. We can quickly define rows and columns using a premade CSS stylesheet. *CSS* (Cascading Style Sheets) is another web language used to define how a page should be displayed. We put the stylesheet in an external file or call one from an online directory into our app. We're using a stylesheet from *https://codepen.io*. Written by Chris Parmer, the creator of Plotly Dash, the stylesheet is comprehensive and suitable to use for a basic Dash app. In Listing 4-4, we import the CSS. We also tell *twitter_app.py* to grab the CSS stylesheet from the web and incorporate it into the app, and we instantiate our app with Dash.

```
stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']
app = Dash(__name__, external_stylesheets=stylesheets)
```

Listing 4-4: Importing a stylesheet into twitter_app.py

Our CSS stylesheet describes the width and height of the columns and rows on the page using CSS classes. We just need to refer to these classes within our Dash code to place the Div content in specific cells inside the grid.

First, we must assign the rows because the columns should be wrapped by rows. To do so, we set a string value "row" to the className. Let's build on the Div example in Listing 4-3, assuming this code has imported the custom stylesheet; the new code is in bold (see Listing 4-5).

```
app.layout = html.Div([
   html.Div(dcc.Dropdown()),
   html.Div(dcc.Dropdown()),
   html.Div(dcc.Dropdown()),
], className="row")
```

Listing 4-5: Example Div code with className (not part of the main app)

Here we assign one row to the html.Div that houses all three dropdowns, so all these dropdowns will be displayed in the same row on the page (Figure 4-5). className is a prop that can be assigned classes from a CSS stylesheet to tell Dash how to style an element. Here we assign it the row class, which tells the app that all the components inside this Div should be on the same row. Every Dash component will have a className, commonly used to style and define layouts. We use the className prop of html.Div to describe the row and column layout of each Div.

After defining the row, we need to define the columns' width so that Dash knows how many columns of space to allocate to each component within that row. Next, we define the columns of each html.Div contained inside that row, shown in bold in Listing 4-6.

```
app.layout = html.Div([
   html.Div(dcc.Dropdown(), className="four columns"),
   html.Div(dcc.Dropdown(), className="four columns"),
   html.Div(dcc.Dropdown(), className="four columns"),
], className="row")
```

Listing 4-6: Setting the column width (not part of the main app)

We set the number of columns of space each `Div` component should fill with a string value set to `className` and formatted like `"one column"` or `"two columns"` and so on. Most web pages will have a maximum of 12 columns (and a potentially unlimited number of rows), meaning the sum of the components' column widths must never surpass 12, so here we set them to fill four columns each. Note that we don't have to fill all 12 columns.

Figure 4-5 shows how this simple page would be displayed.

Figure 4-5: Demo of three dropdowns on one row

With all this in mind, let's have a look at Listing 4-7, the `html.Div` section of our *twitter_app.py* file, which has fewer than 12 columns.

```
html.Div(
   [
   ❶ html.Div(
        dcc.Dropdown(
           id="my-dropdown",
           multi=True,
           options=[
              {"label": x, "value": x}
              for x in sorted(df["name"].unique())
           ],
           value=["taylorswift13", "cristiano", "jtimberlake"],
        ),
        className="three columns",
     ),
   ❷ html.Div(
        html.A(
           id="my-link",
           children="Click here to Visit Twitter",
           href="https://twitter.com/explore",
           target="_blank",
        ),
        className="two columns",
     ),
   ],
   className="row",
),
```

Listing 4-7: The `Dropdown` section of twitter_app.py

We see that the row contains two `Div`s: a `Dropdown` that offers multiple celebrities to choose from ❶ and a link for the user to click ❷. Those two `Div`s have the sum of just five columns, meaning they're left-aligned on the page, as shown in Figure 4-6.

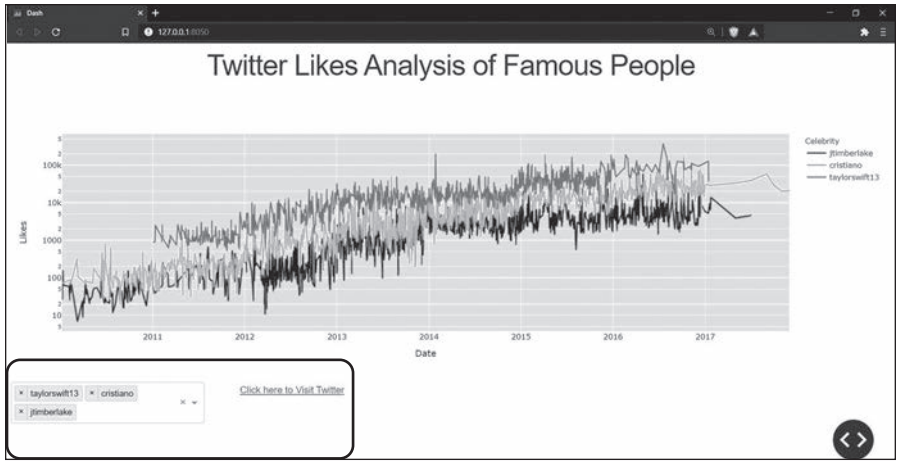


Figure 4-6: Components that are five columns wide

Note that some stylesheets, including the one we're working with here, require us to create the parent `Div` first and assign a row to it. Then, within the children of the parent `Div`, we define the column width of each inner `Div`.

Styling: Embellishing Your App

The styling is what gives life to the app. We can add color, change the font and size of the text, underline text, and much more. There are two main ways to alter the style of the app. The first is to use the `style` prop inside the Dash `HTML` component. This allows the user to specify CSS styling declarations that will be applied directly to the component.

The second method is to refer to a CSS stylesheet, like we did to create rows and columns. In this section, we'll integrate the additional stylesheet *mystyles.css* into the app; if you downloaded the files as described in "Setting Up the Project" earlier in this chapter, this should be in your *assets* folder. Let's first look at how to use the `style` prop to alter the app.

Using the style Prop

The `style` prop expects a Python dictionary, with keys that specify what aspect we want to alter and values that set the style. In our *twitter_app.py* file, we'll change the text color of the link to red by defining the `style` prop within the `html.A` component used for adding URL links, as shown in Listing 4-8.

```
html.Div(
   html.A(id="my-link", children="Click here to Visit Twitter",
      href="https://twitter.com/explore", target="_blank",
   ❶ style={"color": "red"}),
   className="two columns")
```

Listing 4-8: Styling HTML elements of twitter_app.py

At ❶ we assign a dictionary to the `style` prop, where the key is `color` and the value is `red`. This tells the browser to render this link with red text.

Now we'll add a yellow background color to the same link by adding another key-value pair to the dictionary:

```
style={"color": "red", "backgroundColor": "yellow"}
```

Notice that the dictionary key is a camelCased string. In Dash, the keys in the `style` dictionary should always be camelCased.

Lastly, we change the link font size to 40 pixels:

```
style={"color": "red", "backgroundColor": "yellow", "fontSize": "40px"}
```

A beautiful thing about Dash is that styling is not limited to HTML components; we can also style the Core Components such as the `Dropdown`. For example, to change the text color of the dropdown options to green, we add the `style` prop within `dcc.Dropdown`, as shown in Listing 4-9.

```
html.Div(
   dcc.Dropdown(id="my-dropdown", multi=True,
           options=[{"label": x, "value": x}
                 for x in sorted(df["name"].unique())],
           value=["taylorswift13", "cristiano", "jtimberlake"],
           style={"color": "green"}),
   className="three columns"),
```

Listing 4-9: Styling Core Components in twitter_app.py

Notice in Figure 4-7 how the dropdown options at the bottom-left corner are now green instead of black.

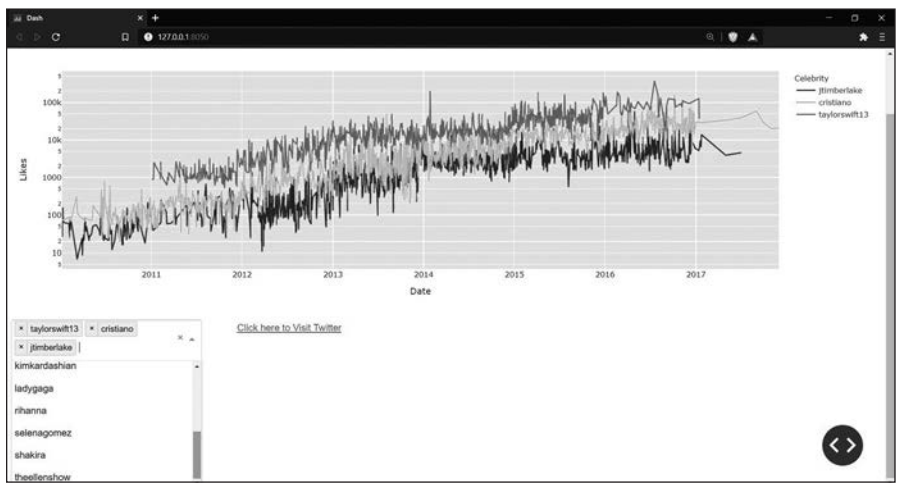


Figure 4-7: Green dropdown options

Using a Stylesheet

The second way to style app components is to define styles through elements or classes. Typically, we use this method when a lot of code is needed for the styling. To reduce the amount of code present in the app itself, we use styling code in an external CSS stylesheet. CSS stylesheets are also reusable; you can define a particular class once and apply it to multiple components.

The CSS stylesheet we'll use is *mystyles.css* and it should already be in the *assets* folder you downloaded with the book's resources. Open the CSS stylesheet inside PyCharm or your preferred text editor by double-clicking it, and you should see these lines of code:

```
/*
h1 { font-size: 8.6rem; line-height: 1.35; letter-spacing: -.08rem;
margin-bottom: 1.2rem; margin-top: 1.2rem;}
*/
```

The /* is comment syntax, so to enable the styling, delete the /* and */ symbols below and above the CSS code. Here `h1` is the *selector*, which specifies the element we want to apply the subsequent styles to; in this case, it's all `h1` elements. Inside the curly braces we declare properties and property values that will set various styles inside the app. In this example, we set the element's font size to 8.6, the line height to 1.35, the spacing between letters to—.08, and the top and bottom margins to 1.2.

Listing 4-10 shows how the `H1` heading component in our app uses this CSS stylesheet.

```
html.Div(html.H1("Twitter Likes Analysis of Famous People",
                 style={"textAlign": "center"}),
         className="row"),
```

Listing 4-10: The `html.H1` component in twitter_app.py

The `html.H1` through `html.H6` components are used to define headings, with `H1` representing the highest heading level and `H6` representing the lowest heading level. Figure 4-8 shows how this header styling should look.

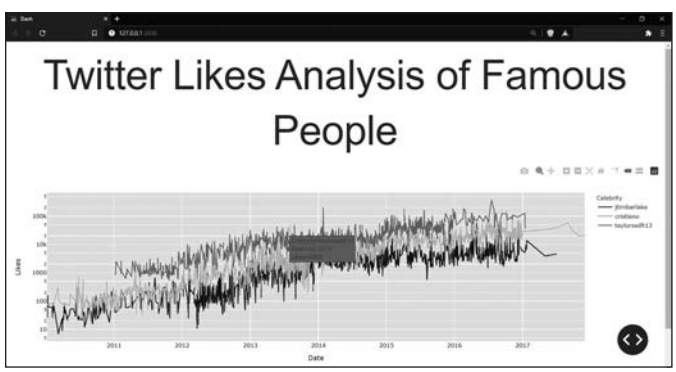


Figure 4-8: App title styled with CSS

As you can see if you compare Figure 4-8 to Figure 4-6, the result is a much larger font size for the app's title, with more top and bottom margin space around the title and less space between the letters. If your app's title did not change in size, restart your app to see the result.

If you'd like to revert back to a smaller font size for the title, simply comment out the CSS code by reinserting the /* and */ symbols, as such:

```
/*
h1 { font-size: 8.6rem; line-height: 1.35; letter-spacing: -.08rem;
margin-bottom: 1.2rem; margin-top: 1.2rem;}
*/
```

You have learned how to manipulate the style and layout of your app with pure Python. This is just the beginning, though. In Chapter 5, we will dive into dash-bootstrap-components, which will make the layout design and styling of the dashboard app even easier and more varied.

Dash Components

Here we'll provide an overview of some common components in Dash, provided by the dash-html-components and dash-core-components libraries. There are many other component libraries, and you can even write your own! But dash-html-components and dash-core-components contain most of the basic functionality we need. The HTML components are generally for composing the layout of the web page and include `Div`, `Button`, `H1`, and `Form`, among many others. The Core Components—such as `Dropdown`, `Checklist`, `RangeSlider`, and many more—are for creating an interactive experience. All HTML and Core Components have props that add to their functionality. For a full list of these props and their components, visit the Dash documentation on HTML and Core Components at *https://dash.plotly.com/dash-core-components*.

HTML Components

Dash HTML Components are written in Python and are automatically converted to HTML, so there's no need to become an expert on HTML or CSS to use Dash apps. The following line of code in Python

```
<h1> Twitter Likes Analysis of Famous People </h1>
```

is roughly equivalent to the following line of HTML that is read by a web browser:

```
html.H1("Twitter Likes Analysis of Famous People")
```

Writing a complete dashboard app is now possible in pure Python: Python forever!

To create an HTML component, you use dot notation between the `html` keyword and the component name. For example, for a `Div` component you would use `html.Div`, as we saw earlier. We also saw two additional

HTML components: `html.H1`, which creates a top-level heading, and `html.A`, which creates a hyperlink. Let's take a closer look at the use of `html.H1` to represent the title of the page, with the title itself written as a string, like so:

```
html.H1("Twitter Likes Analysis of Famous People")
```

This assigns the string to the `children` prop, which is usually the first positional argument of any component that accepts children. `children`, in this context, is a prop that places a component or element (like a text label) within another component. Written in full, the previous line looks like this:

```
html.H1(children="Twitter Likes Analysis of Famous People")
```

In the first three examples of the following code, the `children` prop adds text to the page. In the last example, with `html.Div`, the `children` prop adds the `html.H1` component to the page, which has text as well. The `children` prop can take an integer, a string, a Dash component, or a list of any of these. All these examples are possible:

```
html.H1(children=2),
html.H1(children="Twitter Likes Analysis of Famous People"),
html.H1(children=["Twitter Likes Analysis of Famous People"]),
html.Div(children=[
   html.H1("Twitter Likes Analysis of Famous People"),
   html.H2("Twitter Likes Analysis of Famous People")
])
```

The `html.A` component creates an `<a>` HTML5 element, which is used to create hyperlinks. In this component, shown in Listing 4-11, we use four props: `id`, `children`, `href`, and `target`.

```
html.A(id="my-link", children="Click here to Visit Twitter",
   href="https://twitter.com/explore", target="_blank")
```

Listing 4-11: The HTML link component in twitter_app.py

The value we assign to `href` is the full link destination, where the user will end up once clicking on the link. The `target` prop indicates where the link will open: if its assigned value is `_self`, the link will open in the same tab of the browser the user is in; if its assigned value is `_blank`, the link will open in a new browser tab. The `children` prop defines the content of the component, which here is a string value representing the link's text that the user sees on the page.

The `id` prop is important because Dash components use `id` to identify and interact with each other, which gives the dashboard app its interactive capabilities. We'll go over this in more detail in "Callback Decorator" later in this chapter. For now, just note that the value assigned to `id` must be a unique string so that it can be used to identify the component.

Core Components

The Dash Core Components are prebuilt components from the Dash library that allow the user to interact with the app in an intuitive way. In this app we use two Core components: `Graph` and `Dropdown`. To build or access a particular Core component, we use the `dcc` keyword and the dot notation before the component name, such as `dcc.Dropdown`.

The Graph Component

The `Graph` component allows you to incorporate data visualizations into your app in the form of plots, charts, and graphs written with Plotly. It's one of the most popular of the Core components, and you'll likely see it in every analytic dashboard app.

A `Graph` component has two main props: `id` and `figure`. Here's the template for defining a `Graph` component:

```
dcc.Graph(id="line-chart", figure={})
```

The `id` prop gives the `Graph` component a unique ID. The `figure` prop is the placeholder for the Plotly chart. Once a Plotly chart is created, we would assign it to the `figure` prop in place of the empty dictionary. For example, in our app we create a Plotly line chart with the line shown in Listing 4-12.

```
import plotly.express as px

--snip--

fig = px.line(data_frame=df_filtered, x="date_time", y="number_of_likes",
   color="name", log_y=True)
```

Listing 4-12: Creating a Plotly chart in twitter_app.py

We'll go through Plotly charts in "Plotly Express Line Chart" later in this chapter. For now, this line simply describes how the chart should look and assigns it to the `fig` object, making it a Plotly figure. We can then insert `fig` into `dcc.Graph`'s `figure` prop to display the line chart on the page. Listing 4-13 shows the code from the *twitter_app.py* file that does just that, assigned to `app.layout`.

```
html.Div(dcc.Graph(id="line-chart", figure=fig), className="row")
```

Listing 4-13: Pulling the chart into the Graph component in twitter_app.py

We put the `Graph` component inside the `Div` component and assign it to a single row on the page. Once the complete app script is activated, the line chart should display on the page.

For a complete video tutorial on the Dash `Graph` component and its usage, see the video "All About the Graph Component" at *https://learnplotlydash.com*.

The Dropdown Component

The `Dropdown` component allows users to choose options from a dropdown menu to dynamically filter data and update graphs. We define the `Dropdown` component by providing values for four props: `id`, `multi`, `options`, and `value`, as shown in Listing 4-14. This menu is shown in Figure 4-9.

```
dcc.Dropdown(id="my-dropdown", multi=True,
             options=[{"label": x, "value": x}
                for x in sorted(df["name"].unique())],
             value=["taylorswift13", "cristiano", "jtimberlake"])
```

Listing 4-14: Creating a `Dropdown` component in twitter_app.py

The `multi` prop allows us to choose whether the user can select multiple values at once or just one value at a time. When this prop is set to `True`, the app user can select multiple values. When it's set to `False`, the app user can select only one value at a time.

The `options` prop represents the values the user can choose from when they click the `Dropdown`. We assign it a list of dictionaries of `label` and `value` keys, where each dictionary represents one menu option. The `label` is the name the user sees as the option, and the `value` is the actual data read by the app.

In Listing 4-14, we assigned the list of dictionaries using *list comprehension*, a Python shortcut that creates a new list based on values of another list (or any other Python iterable). For every unique value in the `name` column of our pandas DataFrame, this line creates a dictionary of `label` and `value` keys.

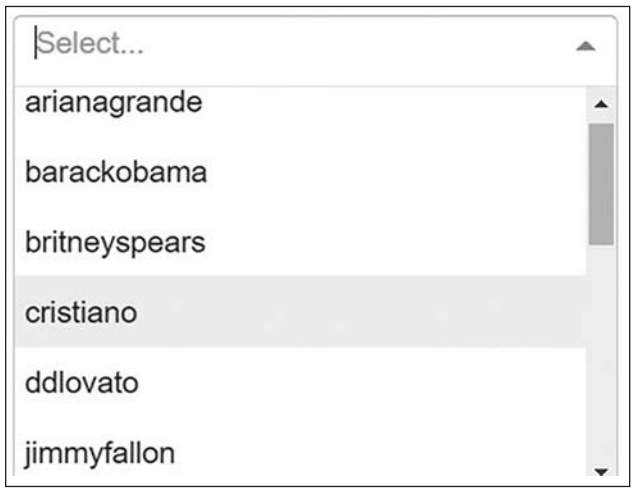


Figure 4-9: App dropdown options

If instead we only have a few values, it may be easier to write out each dictionary instead of using list comprehension. For example, in Listing 4-15 we build a `Dropdown` with only two values: `taylorswift13` and `cristiano`.

```
dcc.Dropdown(id="my-dropdown", multi=True,
             options=[{"label": "Taylor", "value": "taylorswift13"},
                  {"label": "Ronaldo", "value": "cristiano"}]
)
```

Listing 4-15: A `Dropdown` example not in twitter_app.py

Here we choose values as they appear in the DataFrame so that filtering is easier. But we can then choose a human-friendly representation for the `label` key to make it more recognizable to the user. When the user clicks on the dropdown, they will see the two options *Taylor* and *Ronaldo*, which are read by the app as `taylorswift13` and `cristiano`, respectively.

The last `Dropdown` prop is `value` (not to be confused with the dictionary `value` key), and it consists of the default value the `Dropdown` will take when the user starts the app. Since we have a multivalue `Dropdown`, we use an initial value of three strings from the `name` column of the DataFrame: `taylorswift13`, `cristiano`, and `jtimberlake`.

These strings correspond to the values generated in the `options` prop in Listing 4-14. The strings are preloaded, so these three values are automatically chosen before the user even clicks the dropdown menu. Once the user chooses a different value in the dropdown menu, these values change accordingly.

For a complete video tutorial on the Dash `Dropdown` component and its usage, see the video "Dropdown Selector" at *https://learnplotlydash.com*.

Dash Callback

A *Dash callback* enables user interactivity within the dashboard app; it is the mechanism that connects the Dash components to each other so that performing one action causes something else to happen. When the app user selects a dropdown value, the figure is updated; when the user clicks a button, the color of the app's title changes or another graph is added to the page. The possible interactions between Dash components is infinite, and without callbacks, the app is static and the user cannot modify anything.

The Dash callback has two parts: the callback decorator that identifies the relevant components, defined in the layout section:

```
@app.callback()
```

and the callback function that defines how those Dash components should interact:

```
def function_name(y):
   return x
```

This simple app only has one callback, though more complicated apps will have many.

Callback Decorator

A callback decorator registers the callback function with your Dash app, telling it when to call the function and how to use the return value of the function to update the app. (We discussed decorators in Chapter 1.)

The callback decorator should be placed right above the callback function, and there must be no space between the decorator and the function. The decorator takes two main arguments: `Output` and `Input`, which refer to the component that should change (`Output`) in response to the user's action on a different component (`Input`). For example, the output might be the line chart, which should change depending on the user's input in the `Dropdown` component, as shown in Listing 4-16.

```
@app.callback(
   Output(component_id="line-chart", component_property="figure"),
   [Input(component_id="my-dropdown", component_property="value")],
)
```

Listing 4-16: A callback decorator from twitter_app.py

Both `Output` and `Input` take two arguments: `component_id`, which should correspond to the `id` of a particular Dash component; and `component_property`, which should correspond to a particular prop of that same component. In Listing 4-16, the `component_id` for `Input` refers to the `my-dropdown` `Dropdown` we defined earlier. The `component_property` refers specifically to the `value` prop of `my-dropdown`, which is the Twitter users' data to show, initially set to `["taylorswift13","cristiano","jtimberlake"]`, as in Listing 4-14.

In the `Output` we refer to the `figure` prop of `dcc.Graph`, which we also defined earlier in the layout, as shown in Listing 4-17.

```
dcc.Graph(id="line-chart", figure={})
```

Listing 4-17: The Graph component within the layout section in twitter_app.py

Here the `figure` prop is currently an empty dictionary, because the callback function will create a line chart based on the input and assign it to `figure`. Let's dive into the callback function to fully understand how this happens.

Callback Function

Our app's callback function, named `update_graph()`, holds a series of if-else statements that filter the DataFrame `df` and create a line chart depending on the input values chosen. Listing 4-18 shows the callback function in our app.

```
def update_graph(chosen_value):
   print(f"Values chosen by user: {chosen_value}")

   if len(chosen_value) == 0:
      return {}
   else:
```

```
    df_filtered = df[df["name"].isin(chosen_value)]
    fig = px.line(
       data_frame=df_filtered,
       x="date_time",
       y="number_of_likes",
       color="name",
       log_y=True,
       labels={
          "number_of_likes": "Likes",
          "date_time": "Date",
          "name": "Celebrity",
       },
    )
    return fig
```

Listing 4-18: The callback function in twitter_app.py

We'll go over the logic here line by line in a moment. First, though, let's discuss what this function achieves. When executed, `update_graph()` returns an object named `fig`, which in this case contains the Plotly Express line chart. The object `fig` is returned to the component and property we specified in `Output` in the callback decorator. As we know, the callback decorator refers to a Dash component in the layout. Here, then, `fig` is assigned to the `figure` prop of the `Graph` component in the layout section, so the callback is telling the app to display a line chart. Here's what the `Graph` component would look like after the callback function `update_graph()` executes:

```
dcc.Graph(id="line-chart", figure=fig)
```

The `figure` prop is now assigned the object `fig` instead of the empty dictionary we saw originally, in Listing 4-17.

We'll summarize this because this process is extremely important! Once the callback function is activated by user input, it returns an object that is tied to the `component_property` of the `Output` in the callback decorator. Given that the component property represents an actual prop of a component inside the app layout, the result is an app that is constantly being updated through user interaction.

For a complete video tutorial on the Dash callback decorator and its usage, see the video "The Dash Callback—Input, Output, State, and More" at *https://learnplotlydash.com*.

Activating the Callback

To activate the callback, the user must interact with the component specified in `Input` inside the callback decorator. In this app, the component property represents the value of the `Dropdown`, so every time the app user chooses a different dropdown value (a Twitter handle), the callback function is triggered.

If the callback decorator had three `Inputs`, the user would need to supply three arguments to trigger the callback function. In our case, the

callback decorator has only one `Input`; therefore, the callback function will take only one argument: `chosen_value`.

How the Function Works

Let's examine Listing 4-19, which shows what is happening inside the app's callback function.

```
❶ def update_graph(chosen_value):
    print(f"Values chosen by user: {chosen_value}")

❷ if len(chosen_value) == 0:
    return {}
  else:
    df_filtered = df[df["name"].isin(chosen_value)]
    fig = px.line(
      data_frame=df_filtered,
      x="date_time",
      y="number_of_likes",
      color="name",
      log_y=True,
      labels={
        "number_of_likes": "Likes",
        "date_time": "Date",
        "name": "Celebrity",
      },
    )
    return fig
```

Listing 4-19: The callback function for twitter_app.py

The `chosen_value` argument ❶ refers to the value of the `dcc.Dropdown`, which is a list of Twitter usernames. Whenever a user chooses new options, the function is activated. The user can choose any number of available celebrities, and the number of items inside the `chosen_value` list will increase or decrease accordingly. It may be a list of three values, 10 values, or even zero values. We therefore check the length of the `chosen_value` list ❷. If it is equal to zero and so is an empty list, the function returns an empty dictionary, and the `fig` object returned displays an empty graph.

If the length of the `chosen_value` list does not equal zero, in the `else` branch we use pandas to filter the DataFrame to only those rows that contain the selected Twitter usernames. The filtered DataFrame is saved to `df_filtered` and is then used as the data to create the line chart, which is saved as a `fig` object. The `fig` object is returned to display the line chart on the app page.

One important note on these functions: if the original DataFrame is altered in any way, you should always make a copy of the original DataFrame, as we did when we created `df_filtered`. The original DataFrame defined at the beginning of the app, in Listing 4-2, is considered a global variable. Global variables should never be altered, because doing so affects the variables seen by other users of the app. For example, if one user

changed the global variable `price_values` in a financial dashboard app, all users would see these changed prices. This could cause significant damage and confusion.

Callback Diagram

Dash has a powerful callback diagram tool that displays the structure of the callback and delineates how elements are tied together. You should use this tool when defining callbacks, especially when they have multiple `Inputs` and `Outputs`, where it is harder to grasp the callback structure. To open the callback diagram, click the blue button in the bottom right corner of the app page, shown in Figure 4-10.

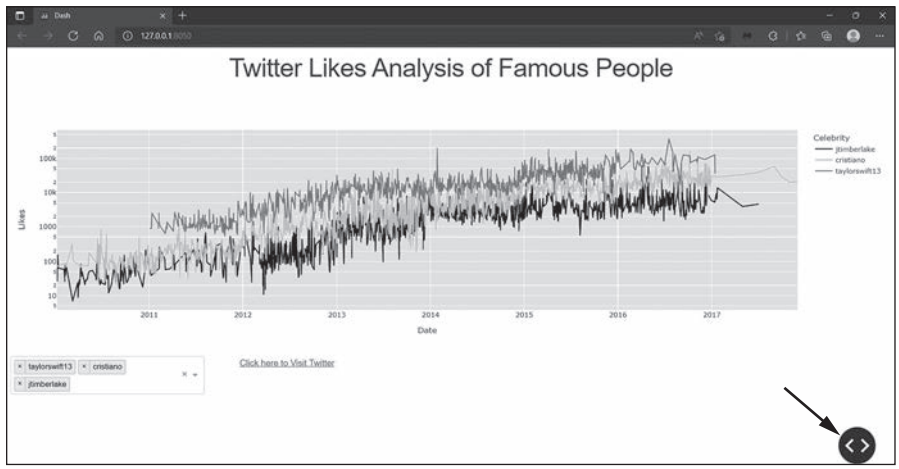


Figure 4-10: Click the button in the bottom right corner to open the menu.

Then click the gray Callbacks button, shown in Figure 4-11.

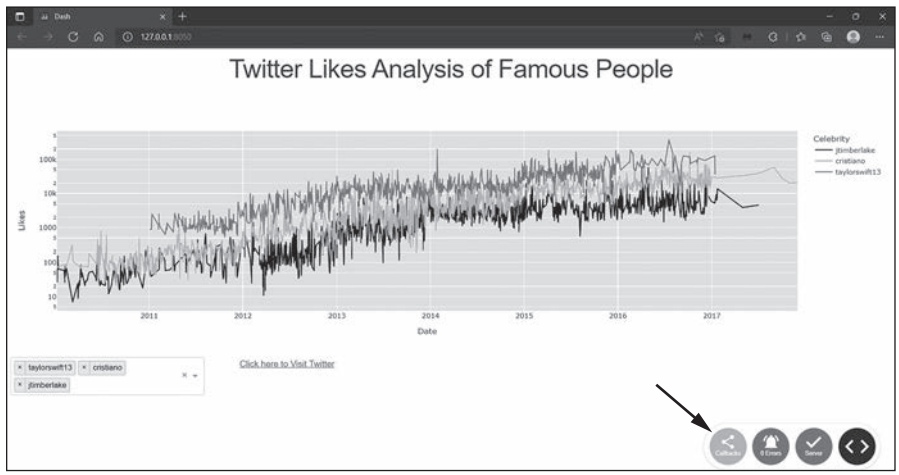


Figure 4-11: Click the Callbacks button to see the callback diagram.

The result should look like Figure 4-12.

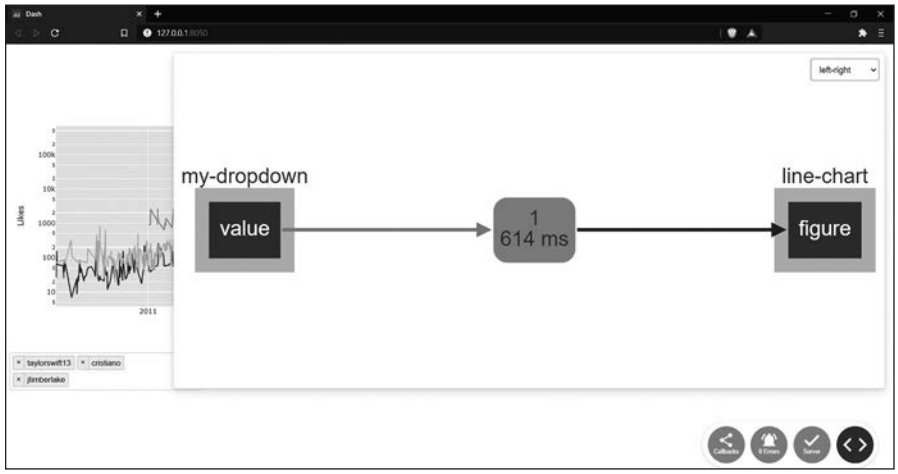


Figure 4-12: The callback diagram for twitter_app.py

The element on the left is the component property of the `Input`. The element in the middle describes the number of times the callback has been triggered in this session (once, in this case) as well as the time it took for the callback to fully execute (614 ms). The element on the right is the `component` property of the `Output`. The diagram helps paint a clear picture of how the `Dropdown` values (`Input`) influence the line chart's figure (`Output`).

Go ahead and trigger the callback by changing the `Dropdown` celebrity names on the main app page. See how the green element in the middle changes? Explore this diagram by clicking the left and right elements; you should see extra information within each element.

Make sure to turn debug mode off with `debug = False` before you deploy your app to the web in order to turn off the diagram. Otherwise, the end user will have access to the diagram as well.

Plotly Express Line Chart

Here we'll review how to create Plotly graphs. We'll focus on line charts, since that's what we use in this app, and we'll review other types of graphs in future chapters.

Plotly Express is a high-level interface for creating graphs quickly and intuitively. It contains dozens of figures to choose from, ranging from scientific, statistical, and financial graphs to 3D charts and maps. Every figure has numerous attributes that allow you to customize figures according to users' needs. Here's a complete list of the attributes available for the Plotly Express line chart, all currently set to `None`:

```
plotly.express.line(data_frame=None, x=None, y=None, line_group=None, color=None, line
_dash=None, hover_name=None, hover_data=None, custom_data=None, text=None, facet_row=None,
facet_col=None, facet_col_wrap=0, facet_row_spacing=None, facet_col_spacing=None, error_x=None,
error_x_minus=None, error_y=None, error_y_minus=None, animation_frame=None, animation_group
=None, category_orders={}, labels={}, orientation=None, color_discrete_sequence=None, color
_discrete_map={}, line_dash_sequence=None, line_dash_map={}, log_x=False, log_y=False, range_x
=None, range_y=None, line_shape=None, render_mode='auto', title=None, template=None,width
=None, height=None)
```

The beautiful thing about Plotly Express is that, in most cases, all you need to know to create a graph are the first three attributes: `data_frame`, `x`, and `y`, shown in bold in the example. These represent the DataFrame, the column of data to use for the x-axis, and the column to use for the y-axis, respectively. Here we plot a really simple line chart:

```
import plotly.express as px
px.line(data_frame=df, x="some_xaxis_data", y="some_yaxis_data")
fig.show()
```

This creates the most basic line chart, charting the relationship between two data columns, giving us something like Figure 4-13.

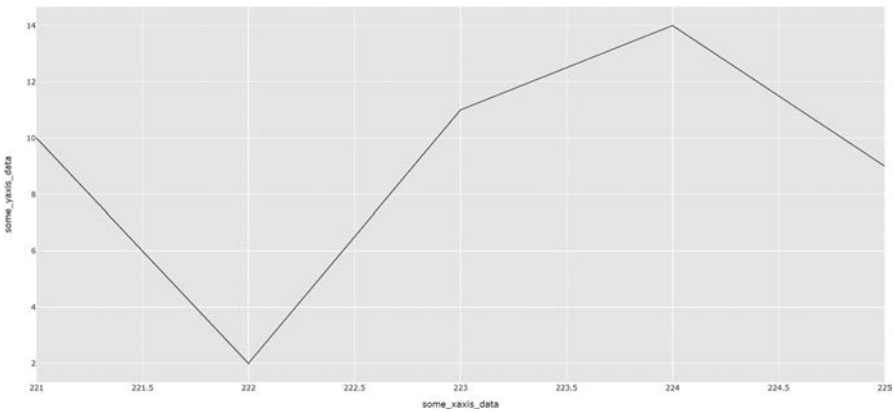


Figure 4-13: The simplest line chart

The more comfortable you become with Plotly Express, the more attributes you will find yourself adding to the figure. For example, to differentiate groups of data with color, we add the `color` attribute and assign it a column from the hypothetical DataFrame used:

```
px.line(data_frame=df, x="some_ xaxis _data", y="some_yaxis_data", color="some_data")
```

As a result, we would see something like Figure 4-14.

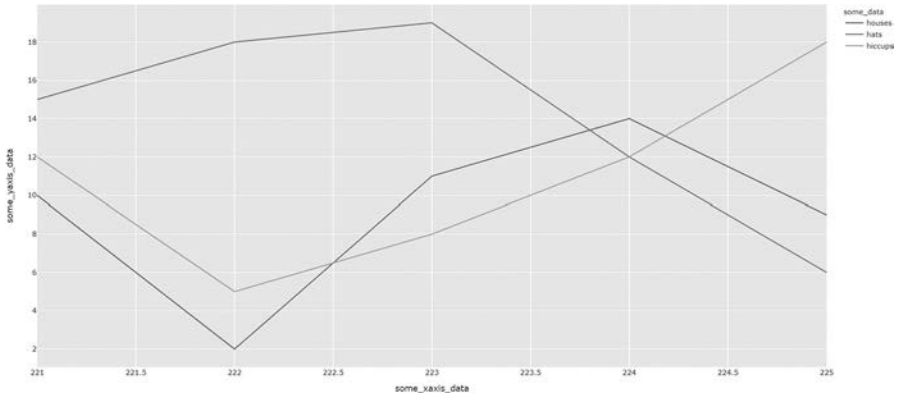


Figure 4-14: Adding a `color` *attribute to the simple chart*

To change the height of the figure, we add the `height` attribute and assign it a number of pixels:

```
px.line(data_frame=df, x="some_xaxis_data", y='some_yaxis_data',
height=300)
```

Here we make the height of the entire graph 300 pixels.

In our Twitter Analysis app, the line chart includes the `data_frame`, `x`, `y`, and `color` attributes, as well as the `labels` and `log_y` attributes. Listing 4-20 shows our Plotly chart code.

```
fig = px.line(
   data_frame=df_filtered,
   x="date_time",
   y="number_of_likes",
   color="name",
   log_y=True,
   labels={
      "number_of_likes": "Likes",
      "date_time": "Date",
      "name": "Celebrity",
   },
)
```

Listing 4-20: The Plotly line chart for twitter_app.py

The `log_y` attribute tells the app to use a logarithmic scale on the y-axis data. Logarithmic scaling is recommended when the chart has a few data points that are much larger or smaller than the bulk of the data, as it makes for a clearer visualization. We won't go into the details of logarithmic scales here, but try changing the attribute from `True` to `False` and refresh the app to see the updated graph. Which one do you prefer?

The `labels` attribute changes the axis labels seen by the app users. The three columns used to plot the line chart are `date_time` (x-axis), `number_of_likes` (y-axis), and `name` (color). These are the names of the columns in the pandas

DataFrame, and we must maintain their format and spelling to match to the right column. With the `labels` attribute, we change what the user sees on the app page to make it more user friendly so that `number_of_likes` simply becomes `Likes`.

Each attribute is described in detail in the Plotly documentation at *https://plotly.com/python-api-reference*. It's worth spending time reading the descriptions because it will help you understand all the ways you can customize the line chart and other types of figures.

For a complete video tutorial on the Plotly Express line chart with `Dropdown`, see the video "Line Plot (Dropdown)" at *https://learnplotlydash.com*.

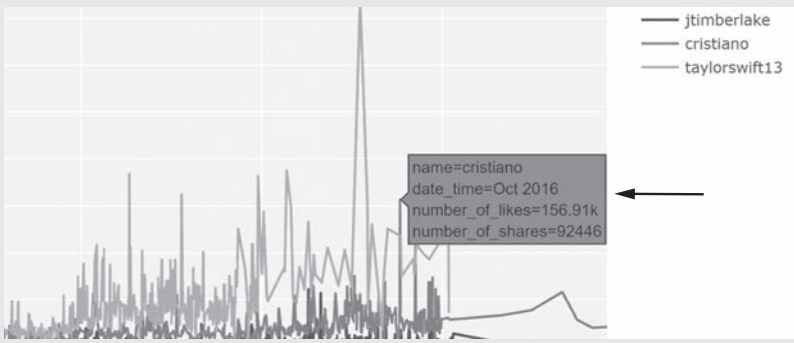
TOOL TIPS

There's one common attribute that we don't use in the app but is common enough that it's worth mentioning here: `hover_data`, which allows you to provide extra information in tool tips that appear when the user hovers over particular elements of the graph with the mouse cursor. You place the values assigned to `hover_data` inside a list or a dictionary.

When you use a list, the graph's hover tool tip will include the values in the list. For example, if we use the `number_of_shares` column as the `hover_data` list, the hover tool tip will include those pieces of data when the user hovers over the lines of our graph. To try this out, make the following change and rerun the app:

```
fig = px.line(data_frame=df_filtered, x="date_time", y="number_of_likes",
color="name", hover_data=["number_of_shares"])
```

The following figure shows the difference in the hover information.



Example tool tip with "number of shares" included in hover data

Make sure to delete the change when you're done.

When you use a dictionary instead of a list, the keys are DataFrame columns and the values are Booleans you use to display (`True`) or not display (`False`) the data in the hover tool tip. For example, if you add the `number_of_likes` column as the dictionary key and `False` as the dictionary value, the data representing the number of likes per celebrity will no longer show in the hover tool tip:

(continued)

```
hover_data={"number_of_likes": False}
```

We can also use the hover_data dictionary to format the hover data seen in the tool tip. For example, by default the number_of_likes are displayed with the letter "k" to represent ten thousand (200000 is written as 200k). However, if we'd prefer to show the full number with a comma as the group separator (200,000), we would use:

```
hover_data={"number_of_likes": ':,'}
```

Summary

This chapter introduced you to the essential elements of a basic Dash app: Python libraries needed to program the app; the data used; Dash HTML and Core Components; using the layout to position the app components on the page; using callbacks to connect the components to each other and create interactivity; and the Plotly Express graphing library. In the next chapter we'll build on the skills learned here to develop more sophisticated Dash apps.