Streamlit Documentation

Release 0.73.1

Streamlit Inc.

Dec 22, 2020

TUTORIALS

The easiest way to learn how to use Streamlit is to try things out yourself. As you read through this guide, test each method. As long as your app is running, every time you add a new element to your script and save, Streamlit's UI will ask if you'd like to rerun the app and view the changes. This allows you to work in a fast interactive loop: you write some code, save it, review the output, write some more, and so on, until you're happy with the results. The goal is to use Streamlit to create an interactive app for your data or model and along the way to use Streamlit to review, debug, perfect, and share your code.

CHAPTER

ONE

CREATE YOUR FIRST STREAMLIT APP

First, we'll create a new Python script and import Streamlit.

1. Create a new Python file named `first_app.py`, then open it with your IDE or text editor.
2. Next, import Streamlit.

```
import streamlit as st
# To make things easier later, we're also importing numpy and pandas for
# working with sample data.
import numpy as np
import pandas as pd
```

3. Run your app. A new tab will open in your default browser. It'll be blank for now. That's OK.

```
streamlit run first_app.py
```

   Running a Streamlit app is no different than any other Python script. Whenever you need to view the app, you can use this command.

   Tip: Did you know you can also pass a URL to *streamlit run*? This is great when combined with Github Gists. For example:

   $ streamlit run https://raw.githubusercontent.com/streamlit/demo-uber-nyc-pickups/master/app.py

   ---

4. You can kill the app at any time by typing **Ctrl+c** in the terminal.

CHAPTER

TWO

ADD TEXT AND DATA

2.1 Add a title

Streamlit has a number of ways to add text to your app. Check out our *API reference* for a complete list.

Let's add a title to test things out:

```
st.title('My first app')
```

That's it! Your app has a title. You can use specific text functions to add content to your app, or you can use `st.write()` and add your own markdown.

2.2 Write a data frame

Along with magic commands, `st.write()` is Streamlit's "Swiss Army knife". You can pass almost anything to `st.write()`: text, data, Matplotlib figures, Altair charts, and more. Don't worry, Streamlit will figure it out and render things the right way.

```
st.write("Here's our first attempt at using data to create a table:")
st.write(pd.DataFrame({
    'first column': [1, 2, 3, 4],
    'second column': [10, 20, 30, 40]
}))
```

There are other data specific functions like `st.dataframe()` and `st.table()` that you can also use for displaying data. Check our advanced guides on displaying data to understand when to use these features and how to add colors and styling to your data frames.

Tip: For this guide we're using small amounts of data so that we can move quickly. You can check out our Tutorial on creating a data explorer to see an example of how to load data from an API and use @st.cache to cache it.

CHAPTER

THREE

USE MAGIC

You can also write to your app without calling any Streamlit methods. Streamlit supports "magic commands," which means you don't have to use `st.write()` at all! Try replacing the code above with this snippet:

```
"""
# My first app
Here's our first attempt at using data to create a table:
"""

df = pd.DataFrame({
  'first column': [1, 2, 3, 4],
  'second column': [10, 20, 30, 40]
})

df
```

Any time that Streamlit sees a variable or a literal value on its own line, it automatically writes that to your app using `st.write()`. For more information, refer to the documentation on magic commands.

CHAPTER

FOUR

DRAW CHARTS AND MAPS

Streamlit supports several popular data charting libraries like Matplotlib, Altair, deck.gl, and more. In this section, you'll add a bar chart, line chart, and a map to your app.

4.1 Draw a line chart

You can easily add a line chart to your app with `st.line_chart()`. We'll generate a random sample using Numpy and then chart it.

```
chart_data = pd.DataFrame(
     np.random.randn(20, 3),
     columns=['a', 'b', 'c'])

st.line_chart(chart_data)
```

4.2 Plot a map

With `st.map()` you can display data points on a map. Let's use Numpy to generate some sample data and plot it on a map of San Francisco.

```
map_data = pd.DataFrame(
    np.random.randn(1000, 2) / [50, 50] + [37.76, -122.4],
    columns=['lat', 'lon'])

st.map(map_data)
```

CHAPTER

FIVE

ADD INTERACTIVITY WITH WIDGETS

With widgets, Streamlit allows you to bake interactivity directly into your apps with checkboxes, buttons, sliders, and more. Check out our *API reference* for a full list of interactive widgets.

5.1 Use checkboxes to show/hide data

One use case for checkboxes is to hide or show a specific chart or section in an app. `st.checkbox()` takes a single argument, which is the widget label. In this sample, the checkbox is used to toggle a conditional statement.

```
if st.checkbox('Show dataframe'):
    chart_data = pd.DataFrame(
       np.random.randn(20, 3),
       columns=['a', 'b', 'c'])

    st.line_chart(chart_data)
```

5.2 Use a selectbox for options

Use `st.selectbox` to choose from a series. You can write in the options you want, or pass through an array or data frame column.

Let's use the `df` data frame we created earlier.

```
option = st.selectbox(
    'Which number do you like best?',
     df['first column'])

'You selected: ', option
```

CHAPTER

SIX

LAY OUT YOUR APP

For a cleaner look, you can move your widgets into a sidebar. This keeps your app central, while widgets are pinned to the left. Let's take a look at how you can use `st.sidebar` in your app.

```
option = st.sidebar.selectbox(
    'Which number do you like best?',
     df['first column'])

'You selected:', option
```

Most of the elements you can put into your app can also be put into a sidebar using this syntax: `st.sidebar.[element_name]()`. Here are a few examples that show how it's used: `st.sidebar.markdown()`, `st.sidebar.slider()`, `st.sidebar.line_chart()`.

You can also use `st.beta_columns` to lay out widgets side-by-side, or `st.beta_expander` to conserve space by hiding away large content.

```
left_column, right_column = st.beta_columns(2)
pressed = left_column.button('Press me?')
if pressed:
    right_column.write("Woohoo!")

expander = st.beta_expander("FAQ")
expander.write("Here you could put in some really, really long explanations...")
```

The only exceptions right now are `st.echo` and `st.spinner`. Rest assured, though, we're currently working on adding support for those too!

CHAPTER

SEVEN

SHOW PROGRESS

When adding long running computations to an app, you can use `st.progress()` to display status in real time.

First, let's import time. We're going to use the `time.sleep()` method to simulate a long running computation:

```
import time
```

Now, let's create a progress bar:

```
'Starting a long computation...'

# Add a placeholder
latest_iteration = st.empty()
bar = st.progress(0)

for i in range(100):
  # Update the progress bar with each iteration.
  latest_iteration.text(f'Iteration {i+1}')
  bar.progress(i + 1)
  time.sleep(0.1)

'...and now we\'re done!'
```

CHAPTER

EIGHT

SHARE YOUR APP

After you've built a Streamlit app, it's time to share it! To show it off to the world you can use **Streamlit sharing** to deploy, manage, and share your app for free. Streamlit sharing is currently invitation only, so please request an invite and we'll get you one soon!

It works in 3 simple steps:

1. Put your app in a public Github repo (and make sure it has a requirements.txt!)
2. Sign into share.streamlit.io
3. Click 'Deploy an app' and then paste in your GitHub URL

That's it! You now have a publicly deployed app that you can share with the world. Click to learn more about *how to use Streamlit sharing*. If you're looking for private sharing for your team, check out Streamlit for Teams.

CHAPTER

NINE

GET HELP

That's it for getting started, now you can go and build your own apps! If you run into difficulties here are a few things you can do.

- Check out our community forum and post a question
- Quick help from command line with `$ streamlit --help`
- Read more documentation! Check out:
  - *Streamlit Cookbook* for things like caching and inserting elements out of order
  - *API reference* for examples of every Streamlit command

CHAPTER

TEN

CREATE A DATA EXPLORER APP

If you've made it this far, chances are you've installed Streamlit and run through the basics in our *get started guide*. If not, now is a good time to take a look.

In this tutorial, you're going to use Streamlit's core features to create an interactive app; exploring a public Uber dataset for pickups and drop-offs in New York City. When you're finished, you'll know how to fetch and cache data, draw charts, plot information on a map, and use interactive widgets, like a slider, to filter results.

Tip: If you'd like to skip ahead and see everything at once, the *complete script is available below*.

10.1 Create an app

1. The first step is to create a new Python script. Let's call it `uber_pickups.py`.
2. Open `uber_pickups.py` in your favorite IDE or text editor, then add these lines:

```
import streamlit as st
import pandas as pd
import numpy as np
```

3. Every good app has a title, so let's add one:

```
st.title('Uber pickups in NYC')
```

4. Now it's time to run Streamlit from the command line:

```
streamlit run uber_pickups.py
```

5. As usual, the app should automatically open in a new tab in your browser.

10.2 Fetch some data

Now that you have an app, the next thing you'll need to do is fetch the Uber dataset for pickups and drop-offs in New York City.

1. Let's start by writing a function to load the data. Add this code to your script:

```
DATE_COLUMN = 'date/time'
DATA_URL = ('https://s3-us-west-2.amazonaws.com/'
         'streamlit-demo-data/uber-raw-data-sep14.csv.gz')

def load_data(nrows):
    data = pd.read_csv(DATA_URL, nrows=nrows)
    lowercase = lambda x: str(x).lower()
    data.rename(lowercase, axis='columns', inplace=True)
    data[DATE_COLUMN] = pd.to_datetime(data[DATE_COLUMN])
    return data
```

You'll notice that `load_data` is a plain old function that downloads some data, puts it in a Pandas dataframe, and converts the date column from text to datetime. The function accepts a single parameter (`nrows`), which specifies the number of rows that you want to load into the dataframe.

2. Now let's test the function and review the output. Below your function, add these lines:

```
# Create a text element and let the reader know the data is loading.
data_load_state = st.text('Loading data...')
# Load 10,000 rows of data into the dataframe.
data = load_data(10000)
# Notify the reader that the data was successfully loaded.
data_load_state.text('Loading data...done!')
```

You'll see a few buttons in the upper-right corner of your app asking if you'd like to rerun the app. Choose **Always rerun**, and you'll see your changes automatically each time you save.

Ok, that's underwhelming...

It turns out that it takes a long time to download data, and load 10,000 lines into a dataframe. Converting the date column into datetime isn't a quick job either. You don't want to reload the data each time the app is updated – luckily Streamlit allows you to cache the data.

10.3 Effortless caching

1. Try adding `@st.cache` before the `load_data` declaration:

```
@st.cache
def load_data(nrows):
```

2. Then save the script, and Streamlit will automatically rerun your app. Since this is the first time you're running the script with `@st.cache`, you won't see anything change. Let's tweak your file a little bit more so that you can see the power of caching.

3. Replace the line `data_load_state.text('Loading data...done!')` with this:

```
data_load_state.text("Done! (using st.cache)")
```

4. Now save. See how the line you added appeared immediately? If you take a step back for a second, this is actually quite amazing. Something magical is happening behind the scenes, and it only takes one line of code to activate it.

10.3.1 How's it work?

Let's take a few minutes to discuss how `@st.cache` actually works.

When you mark a function with Streamlit's cache annotation, it tells Streamlit that whenever the function is called that it should check three things:

1. The actual bytecode that makes up the body of the function
2. Code, variables, and files that the function depends on
3. The input parameters that you called the function with

If this is the first time Streamlit has seen these items, with these exact values, and in this exact combination, it runs the function and stores the result in a local cache. The next time the function is called, if the three values haven't changed, then Streamlit knows it can skip executing the function altogether. Instead, it reads the output from the local cache and passes it on to the caller – like magic.

"But, wait a second," you're saying to yourself, "this sounds too good to be true. What are the limitations of all this awesomesauce?"

Well, there are a few:

1. Streamlit will only check for changes within the current working directory. If you upgrade a Python library, Streamlit's cache will only notice this if that library is installed inside your working directory.
2. If your function is not deterministic (that is, its output depends on random numbers), or if it pulls data from an external time-varying source (for example, a live stock market ticker service) the cached value will be none-the-wiser.
3. Lastly, you should not mutate the output of a cached function since cached values are stored by reference (for performance reasons and to be able to support libraries such as TensorFlow). Note that, here, Streamlit is smart enough to detect these mutations and show a loud warning explaining how to fix the problem.

While these limitations are important to keep in mind, they tend not to be an issue a surprising amount of the time. Those times, this cache is really transformational.

Tip: Whenever you have a long-running computation in your code, consider refactoring it so you can use *@st.cache*, if possible.

Now that you know how caching with Streamlit works, let's get back to the Uber pickup data.

10.4 Inspect the raw data

It's always a good idea to take a look at the raw data you're working with before you start working with it. Let's add a subheader and a printout of the raw data to the app:

```
st.subheader('Raw data')
st.write(data)
```

In the *get started guide* you learned that `st.write` will render almost anything you pass to it. In this case, you're passing in a dataframe and it's rendering as an interactive table.

`st.write` tries to do the right thing based on the data type of the input. If it isn't doing what you expect you can use a specialized command like `st.dataframe` instead. For a full list, see *API reference*.

10.5 Draw a histogram

Now that you've had a chance to take a look at the dataset and observe what's available, let's take things a step further and draw a histogram to see what Uber's busiest hours are in New York City.

1. To start, let's add a subheader just below the raw data section:

```
st.subheader('Number of pickups by hour')
```

2. Use NumPy to generate a histogram that breaks down pickup times binned by hour:

```
hist_values = np.histogram(
    data[DATE_COLUMN].dt.hour, bins=24, range=(0,24))[0]
```

3. Now, let's use Streamlit's `st.bar_chart()` method to draw this histogram.

```
st.bar_chart(hist_values)
```

4. Save your script. This histogram should show up in your app right away. After a quick review, it looks like the busiest time is 17:00 (5 P.M.).

To draw this diagram we used Streamlit's native `bar_chart()` method, but it's important to know that Streamlit supports more complex charting libraries like Altair, Bokeh, Plotly, Matplotlib and more. For a full list, see supported charting libraries.

10.6 Plot data on a map

Using a histogram with Uber's dataset helped us determine what the busiest times are for pickups, but what if we wanted to figure out where pickups were concentrated throughout the city. While you could use a bar chart to show this data, it wouldn't be easy to interpret unless you were intimately familiar with latitudinal and longitudinal coordinates in the city. To show pickup concentration, let's use Streamlit `st.map()` function to overlay the data on a map of New York City.

1. Add a subheader for the section:

```
st.subheader('Map of all pickups')
```

2. Use the `st.map()` function to plot the data:

```
st.map(data)
```

3. Save your script. The map is fully interactive. Give it a try by panning or zooming in a bit.

After drawing your histogram, you determined that the busiest hour for Uber pickups was 17:00. Let's redraw the map to show the concentration of pickups at 17:00.

1. Locate the following code snippet:

```
st.subheader('Map of all pickups')
st.map(data)
```

2. Replace it with:

```
hour_to_filter = 17
filtered_data = data[data[DATE_COLUMN].dt.hour == hour_to_filter]
```

(continues on next page)

(continued from previous page)

```
st.subheader(f'Map of all pickups at {hour_to_filter}:00')
st.map(filtered_data)
```

3. You should see the data update instantly.

To draw this map we used the `st.map` function that's built into Streamlit, but if you'd like to visualize complex map data, we encourage you to take a look at the `st.pydeck_chart`.

10.7 Filter results with a slider

In the last section, when you drew the map, the time used to filter results was hardcoded into the script, but what if we wanted to let a reader dynamically filter the data in real time? Using Streamlit's widgets you can. Let's add a slider to the app with the `st.slider()` method.

1. Locate `hour_to_filter` and replace it with this code snippet:

```
hour_to_filter = st.slider('hour', 0, 23, 17)  # min: 0h, max: 23h, default: 17h
```

2. Use the slider and watch the map update in real time.

10.8 Use a button to toggle data

Sliders are just one way to dynamically change the composition of your app. Let's use the `st.checkbox` function to add a checkbox to your app. We'll use this checkbox to show/hide the raw data table at the top of your app.

1. Locate these lines:

```
st.subheader('Raw data')
st.write(data)
```

2. Replace these lines with the following code:

```
if st.checkbox('Show raw data'):
    st.subheader('Raw data')
    st.write(data)
```

We're sure you've got your own ideas. When you're done with this tutorial, check out all the widgets that Streamlit exposes in our *API reference*.

10.9 Let's put it all together

That's it, you've made it to the end. Here's the complete script for our interactive app.

Tip: If you've skipped ahead, after you've created your script, the command to run Streamlit is *streamlit run [app name]*.

```
import streamlit as st
import pandas as pd
import numpy as np

st.title('Uber pickups in NYC')

DATE_COLUMN = 'date/time'
DATA_URL = ('https://s3-us-west-2.amazonaws.com/'
            'streamlit-demo-data/uber-raw-data-sep14.csv.gz')

@st.cache
def load_data(nrows):
    data = pd.read_csv(DATA_URL, nrows=nrows)
    lowercase = lambda x: str(x).lower()
    data.rename(lowercase, axis='columns', inplace=True)
    data[DATE_COLUMN] = pd.to_datetime(data[DATE_COLUMN])
    return data

data_load_state = st.text('Loading data...')
data = load_data(10000)
data_load_state.text("Done! (using st.cache)")

if st.checkbox('Show raw data'):
    st.subheader('Raw data')
    st.write(data)

st.subheader('Number of pickups by hour')
hist_values = np.histogram(data[DATE_COLUMN].dt.hour, bins=24, range=(0,24))[0]
st.bar_chart(hist_values)

# Some number in the range 0-23
hour_to_filter = st.slider('hour', 0, 23, 17)
filtered_data = data[data[DATE_COLUMN].dt.hour == hour_to_filter]

st.subheader('Map of all pickups at %s:00' % hour_to_filter)
st.map(filtered_data)
```

CHAPTER

ELEVEN

CREATE A STREAMLIT COMPONENT

Note: If you are only interested in **using Streamlit Components**, then you can skip this section and head over to the Streamlit Components Gallery to find and install components created by the community!

Starting with version 0.63.0, developers can write JavaScript and HTML "components" that can be rendered in Streamlit apps. Streamlit Components can receive data from, and also send data to, Streamlit Python scripts.

Streamlit Components let you expand the functionality provided in the base Streamlit package. Use Streamlit Components to create the needed functionality for your use case, then wrap it up in a Python package and share with the broader Streamlit community!

Types of Streamlit Components you could create include:

- Custom versions of existing Streamlit elements and widgets, such as `st.slider` or `st.file_uploader`
- Completely new Streamlit elements and widgets by wrapping existing React.js, Vue.js, or other JavaScript widget toolkits
- Rendering Python objects having methods that output HTML, such as IPython `__repr_html__`
- Convenience functions for commonly-used web features like GitHub gists and Pastebin

Check out these Streamlit Components Tutorial videos by Streamlit engineer Tim Conkling to get started:

11.1 Part 1: Setup and Architecture

11.2 Part 2: Make a Slider Widget

CHAPTER

TWELVE

INSTALL STREAMLIT

12.1 Prerequisites

Before you get started, you're going to need a few things:

- Your favorite IDE or text editor
- Python 3.6 - 3.8
- PIP

If you haven't already, take a few minutes to read through *Main concepts* to understand Streamlit's data flow model.

12.2 Set up your virtual environment

Regardless of which package management tool you're using, we recommend running the commands on this page in a virtual environment. This ensures that the dependencies pulled in for Streamlit don't impact any other Python projects you're working on.

Below are a few tools you can use for environment management:

- pipenv
- poetry
- venv
- virtualenv
- conda

12.3 Install Streamlit

```
pip install streamlit
```

Now run the hello world app to make sure everything is working:

```
streamlit hello
```

12.4 Import Streamlit

Now that everything's installed, let's create a new Python script and import Streamlit.

1. Create a new Python file named `first_app.py`, then open it with your IDE or text editor.
2. Next, import Streamlit.

```
import streamlit as st
# To make things easier later, we're also importing numpy and pandas for
# working with sample data.
import numpy as np
import pandas as pd
```

3. Run your app. A new tab will open in your default browser. It'll be blank for now. That's OK.

```
streamlit run first_app.py
```

   Running a Streamlit app is no different than any other Python script. Whenever you need to view the app, you can use this command.

   Tip: Did you know you can also pass a URL to *streamlit run*? This is great when combined with Github Gists. For example:

   $ streamlit run https://raw.githubusercontent.com/streamlit/demo-uber-nyc-pickups/master/app.py

   ---

4. You can kill the app at any time by typing **Ctrl+c** in the terminal.

CHAPTER

THIRTEEN

CREATE AN APP

Working with Streamlit is simple. First you sprinkle a few Streamlit commands into a normal Python script, then you run it with `streamlit run`:

```
streamlit run your_script.py [-- script args]
```

As soon as you run the script as shown above, a local Streamlit server will spin up and your app will open in a new tab your default web browser. The app is your canvas, where you'll draw charts, text, widgets, tables, and more.

What gets drawn in the app is up to you. For example `st.text` writes raw text to your app, and `st.line_chart` draws — you guessed it — a line chart. Refer to our *API documentation* to see all commands that are available to you.

Note: When passing your script some custom arguments, they must be passed after two dashes. Otherwise the arguments get interpreted as arguments to Streamlit itself.

Tip: You can also pass a URL to *streamlit run*! This is great when combined with Github Gists. For example:

$ streamlit run https://raw.githubusercontent.com/streamlit/demo-uber-nyc-pickups/master/app.py

---

13.1 Development flow

Every time you want to update your app, save the source file. When you do that, Streamlit detects if there is a change and asks you whether you want to rerun your app. Choose "Always rerun" at the top-right of your screen to automatically update your app every time you change its source code.

This allows you to work in a fast interactive loop: you type some code, save it, try it out live, then type some more code, save it, try it out, and so on until you're happy with the results. This tight loop between coding and viewing results live is one of the ways Streamlit makes your life easier.

Tip: While developing a Streamlit app, it's recommended to lay out your editor and browser windows side by side, so the code and the app can be seen at the same time. Give it a try!

13.2 Data flow

Streamlit's architecture allows you to write apps the same way you write plain Python scripts. To unlock this, Streamlit apps have a unique data flow: any time something must be updated on the screen, Streamlit reruns your entire Python script from top to bottom.

This can happen in two situations:

- Whenever you modify your app's source code.
- Whenever a user interacts with widgets in the app. For example, when dragging a slider, entering text in an input box, or clicking a button.

And to make all of this fast and seamless, Streamlit does some heavy lifting for you behind the scenes. A big player in this story is the `@st.cache` decorator, which allows developers to skip certain costly computations when their apps rerun. We'll cover caching later in this page.

13.3 Display and style data

There are a few ways to display data (tables, arrays, data frames) in Streamlit apps. In *getting started*, you were introduced to *magic* and `st.write()`, which can be used to write anything from text to tables. Now let's take a look at methods designed specifically for visualizing data.

You might be asking yourself, "why wouldn't I always use st.write()?" There are a few reasons:

1. *Magic* and `st.write()` inspect the type of data that you've passed in, and then decide how to best render it in the app. Sometimes you want to draw it another way. For example, instead of drawing a dataframe as an interactive table, you may want to draw it as a static table by using st.table(df).
2. The second reason is that other methods return an object that can be used and modified, either by adding data to it or replacing it.
3. Finally, if you use a more specific Streamlit method you can pass additional arguments to customize its behavior.

For example, let's create a data frame and change its formatting with a Pandas `Styler` object. In this example, you'll use Numpy to generate a random sample, and the `st.dataframe()` method to draw an interactive table.

Note: This example uses Numpy to generate a random sample, but you can use Pandas DataFrames, Numpy arrays, or plain Python arrays.

```
dataframe = np.random.randn(10, 20)
st.dataframe(dataframe)
```

Let's expand on the first example using the Pandas `Styler` object to highlight some elements in the interactive table.

Note: If you used PIP to install Streamlit, you'll need to install Jinja2 to use the Styler object. To install Jinja2, run: pip install jinja2.

```
dataframe = pd.DataFrame(
    np.random.randn(10, 20),
    columns=('col %d' % i for i in range(20)))

st.dataframe(dataframe.style.highlight_max(axis=0))
```

Streamlit also has a method for static table generation: `st.table()`.

```
dataframe = pd.DataFrame(
    np.random.randn(10, 20),
    columns=('col %d' % i for i in range(20)))
st.table(dataframe)
```

13.4 Widgets

When you've got the data or model into the state that you want to explore, you can add in widgets like `st.slider()`, `st.button()` or `st.selectbox()`. It's really straightforward — treat widgets as variables:

```
import streamlit as st
x = st.slider('x')  #  this is a widget
st.write(x, 'squared is', x * x)
```

On first run, the app above should output the text "0 squared is 0". Then every time a user interacts with a widget, Streamlit simply reruns your script from top to bottom, assigning the current state of the widget to your variable in the process.

For example, if the user moves the slider to position `10`, Streamlit will rerun the code above and set `x` to `10` accordingly. So now you should see the text "10 squared is 100".

13.5 Layout

Streamlit makes it easy to organize your widgets in a left panel sidebar with `st.sidebar`. Each element that's passed to `st.sidebar` is pinned to the left, allowing users to focus on the content in your app while still having access to UI controls.

For example, if you want to add a selectbox and a slider to a sidebar, use `st.sidebar.slider` and `st.siderbar.selectbox` instead of `st.slider` and `st.selectbox`:

```
import streamlit as st

# Add a selectbox to the sidebar:
add_selectbox = st.sidebar.selectbox(
    'How would you like to be contacted?',
    ('Email', 'Home phone', 'Mobile phone')
)

# Add a slider to the sidebar:
add_slider = st.sidebar.slider(
    'Select a range of values',
    0.0, 100.0, (25.0, 75.0)
)
```

Beyond the sidebar, Streamlit offers several other ways to control the layout of your app. `st.beta_columns` lets you place widgets side-by-side, and `st.beta_expander` lets you conserve space by hiding away large content.

```
import streamlit as st

left_column, right_column = st.beta_columns(2)
# You can use a column just like st.sidebar:
```

(continues on next page)

(continued from previous page)

```
left_column.button('Press me!')

# Or even better, call Streamlit functions inside a "with" block:
with right_column:
    chosen = st.radio(
        'Sorting hat',
        ("Gryffindor", "Ravenclaw", "Hufflepuff", "Slytherin"))
    st.write(f"You are in {chosen} house!")
```

Note: *st.echo* and *st.spinner* are not currently supported inside the sidebar or layout options.

13.6 Caching

The Streamlit cache allows your app to execute quickly even when loading data from the web, manipulating large datasets, or performing expensive computations.

To use the cache, wrap functions with the `@st.cache` decorator:

```
@st.cache  #  This function will be cached
def my_slow_function(arg1, arg2):
    # Do something really slow in here!
    return the_output
```

When you mark a function with the `@st.cache` decorator, it tells Streamlit that whenever the function is called it needs to check a few things:

1. The input parameters that you called the function with
2. The value of any external variable used in the function
3. The body of the function
4. The body of any function used inside the cached function

If this is the first time Streamlit has seen these four components with these exact values and in this exact combination and order, it runs the function and stores the result in a local cache. Then, next time the cached function is called, if none of these components changed, Streamlit will skip executing the function altogether and, instead, return the output previously stored in the cache.

For more information about the Streamlit cache, its configuration parameters, and its limitations, see *Caching*.

13.7 App model

Now that you know a little more about all the individual pieces, let's close the loop and review how it works together:

1. Streamlit apps are Python scripts that run from top to bottom
2. Every time a user opens a browser tab pointing to your app, the script is re-executed
3. As the script executes, Streamlit draws its output live in a browser
4. Scripts use the Streamlit cache to avoid recomputing expensive functions, so updates happen very fast
5. Every time a user interacts with a widget, your script is re-executed and the output value of that widget is set to the new value during that run.


cache data 1
cache data 2
cache data 3
cache data 4
cache data 5
cache data 6
cache data 7
cache data 8
Cache
app state
user event
app state
Uber Pickups in New York City
Run the script
Render the app
Rerun the script with new state
Update the app (as needed)
time

CHAPTER

FOURTEEN

DEPLOY AN APP

Now that you've created your app, you're ready to share it! Use **Streamlit sharing** to share it with the world completely for free. Streamlit sharing is the perfect solution if your app is hosted in a public GitHub repo and you'd like anyone in the world to be able to access it. If that doesn't sound like your app, then check out Streamlit for Teams for more information on how to get secure, private sharing for your apps.

Of course, if you want to host your app using another hosting provider, go for it! Streamlit apps work anywhere a Python app works. You can find guides for other hosting providers on our community-supported deployment wiki.

14.1 Get a Streamlit sharing account

To get started, first request an invite at streamlit.io/sharing. Streamlit sharing is currently available by invitation only while we ramp things up. We promise to get you one very quickly

Once you have your invite you're ready to deploy! It's really straightforward, just follow the next few steps.

14.2 Put your Streamlit app on GitHub

Make sure your app is in a public GitHub repo and that you have a requirements.txt file

- If you need to generate a requirements file, try using `pipreqs`

```
pip install pipreqs
pipreqs /home/project/location
```

Note: Only include packages in requirements.txt that are not distributed with a standard Python installation (i.e. only packages that need to be installed with pip or conda). If any of the modules from base Python are included in the requirements.txt file, you will get an error when you try to deploy. Additionally, use versions **0.69.2+** of Streamlit to ensure full sharing functionality.

- If you have requirements for apt-get, add them to `packages.txt`, one package name per line. See our streamlit-apps demo repo for an example packages.txt file.

14.3 Log in to share.streamlit.io

The first thing you'll see is a button to login with GitHub. Click on the button to login with the primary email associated with your GitHub account.

Important: If the email you originally signed-up with isn't the primary email associated with your GitHub account, just reply to your invite email telling us your primary Github email so we can grant access to the correct account.

You can find your GitHub account email here:

14.4 Deploy your app

Click "New app", then fill in your repo, branch, and file path, and click "Deploy". Your app will take a minute or two to deploy and then you'll be ready to share!

If your app has a lot of dependencies it may take some time to deploy the first time. But after that, any change that does not touch your dependencies should show up immediately.

That's it — you're done! Your app can be found at:

```
https://share.streamlit.io/[user name]/[repo name]/[branch name]/[app path]
```

for example:

```
http://share.streamlit.io/streamlit/demo-self-driving/master/streamlit_app.py
```

Most times, your app is also given a shortened URL of the form `https://share.streamlit.io/[user name]/[repo name]`. The only time you need the full URL is when you deployed multiple apps from the same repo. So you can also reach the example URL above at the short URL http://share.streamlit.io/streamlit/demo-self-driving.

14.5 Share, update, and collaborate

Now that your app is deployed you can easily share it and collaborate on it. But first, let's take a moment and do a little joy dance for getting that app deployed!

14.5.1 Share your app

Your app is now live at that fixed URL, so go wild and share it with whomever you want. From your deployed app you can click on the "" menu on the top right and select 'Share this app' to post it directly into social media or to share with the community on the Forum. We'd love to see what you make!

14.5.2 Update your app

Your GitHub repository is the source for the app, so that means that any time you push an update to your repo you'll see it reflected in the app in almost real time. Try it out!

Streamlit also smartly detects whether you touched your dependencies, in which case it will automatically do a full redeploy for you - which will take a little more time. But since most updates don't involve dependency changes, you should usually see your app update in real time.

14.5.3 Collaborate on your app

You can collaborate with others by simply having multiple contributors pushing to the same GitHub repo. If you want to try out something new while still keeping your original app running, just create a new branch, make some changes, and deploy a new version of the Streamlit app.

Every deployed app has its Github source code linked in the "" menu on the top right. So if you are looking to understand the code of another Streamlit app, you can simply navigate to the GitHub page from there and read or fork the app.

14.6 App access and usage

Streamlit sharing is completely free and is meant to get you started with sharing your Streamlit apps. If you need a solution with access controls, ability to deploy from private repos, ability to customize resources, and much more, please check out Streamlit for Teams.

14.6.1 Access

- Apps are always visible to the entire world.
- You can only deploy apps that are in a public GitHub repo.
- Your source code must live in Github. We're looking to expand to other Git hosts soon.
- Everyone with push access to your repo is automatically a maintainer of the app.

14.6.2 Resource limits

- You can deploy up to 3 apps per account.
- Apps get up to 1 CPU, 800 MB of RAM, and 800 MB of dedicated storage in a shared execution environment.
- Apps do not have access to a GPU.
- If you have a special good-for-the-world case that needs more resources, send us an email and we'll see about making an exception!

14.7 Managing apps

To view or change your deployed Streamlit apps, use your app dashboard at share.streamlit.io to view your apps, deploy a new app, delete an app, or reboot an app.

14.7.1 App dashboard

When you first log into share.streamlit.io you will land on your app dashboard, which gives you a list of all your deployed apps. This list does include apps deployed by other people who have push access to the same repos as you, since you're all managers of those apps. Such apps are indicated with an icon like this one:

14.7.2 Reboot an app

If your app needs a hard reboot, just click on the "" overflow menu to the right of the app and click to Reboot. This will interrupt any user that may currently be using that app. It may also take a few minutes for your app to re-deploy, and in that time you — and anyone visiting the app — will see the 'Your app is in the oven' screen.

14.7.3 Delete an app

If you're not using the app anymore, go ahead and delete it! That will free up space for you to host new apps. Simply click the "" overflow menu to the right of the app and select delete. To make sure that you do want to delete the app we ask you to type in the name of the repo to confirm that app will be deleted. Don't worry if you have multiple apps in that repo, we'll just delete the one you selected.

14.7.4 View logs

You can see logs for your app by just navigating to your app and expanding the "Manage app" button on the bottom right. That will open up a terminal view that will let you see live all the logs for your app.

14.7.5 Add or remove dependencies

You can add/remove dependencies at any point by updating `requirements.txt` (Python deps) or `packages.txt` (Debian deps) and doing a `git push` to your remote repo. This will cause Streamlit to detect there was a change in its dependencies, which will automatically trigger its installation.

It is best practice to pin your Streamlit version in `requirements.txt`. Otherwise, the version may be auto-upgraded at any point without your knowledge, which could lead to undesired results (e.g. when we deprecate a feature in Streamlit).

14.7.6 Secrets management

This feature is in the works. So watch for it coming soon to beta!

14.8 Limitations and known issues

Here are some limitations and known issues that we're actively working to resolve. If you find an issue please let us know!

- **If you're having trouble logging in,** check your Streamlit sharing invitation email and make sure you signed up using your Primary Github email, which you can find here.
- When you print something to the terminal, you may need to do a `sys.stdout.flush()` before it shows up.
- Apps execute in a Linux environment running Debian Buster (slim) with Python 3.7. There is no way to change these, and we may upgrade the environment at any point. If we do upgrade it, we will *usually* not touch existing apps, so they'll continue to work as expected. But if there's a critical fix in the update, we *may* force-upgrade all apps.
- Matplotlib doesn't work well with threads. . So if you're using Matplotlib you should wrap your code with locks as shown in the snippet below. This Matplotlib bug is more prominent when you share your app apps since you're more likely to get more concurrent users then.

```
from matplotlib.backends.backend_agg import RendererAgg
_lock = RendererAgg.lock

with _lock:
  fig.title('This is a figure)')
  fig.plot([1,20,3,40])
  st.pyplot(fig)
```

- All apps are hosted in the United States. This is currently not configurable.

CHAPTER

FIFTEEN

IMPROVE APP PERFORMANCE

Streamlit provides a caching mechanism that allows your app to stay performant even when loading data from the web, manipulating large datasets, or performing expensive computations. This is done with the `@st.cache` decorator.

When you mark a function with the `@st.cache` decorator, it tells Streamlit that whenever the function is called it needs to check a few things:

1. The input parameters that you called the function with
2. The value of any external variable used in the function
3. The body of the function
4. The body of any function used inside the cached function

If this is the first time Streamlit has seen these four components with these exact values and in this exact combination and order, it runs the function and stores the result in a local cache. Then, next time the cached function is called, if none of these components changed, Streamlit will just skip executing the function altogether and, instead, return the output previously stored in the cache.

The way Streamlit keeps track of changes in these components is through hashing. Think of the cache as an in-memory key-value store, where the key is a hash of all of the above and the value is the actual output object passed by reference.

Finally, `@st.cache` supports arguments to configure the cache's behavior. You can find more information on those in our *API reference*.

Let's take a look at a few examples that illustrate how caching works in a Streamlit app.

15.1 Example 1: Basic usage

For starters, let's take a look at a sample app that has a function that performs an expensive, long-running computation. Without caching, this function is rerun each time the app is refreshed, leading to a poor user experience. Copy this code into a new app and try it out yourself:

```
import streamlit as st
import time

def expensive_computation(a, b):
    time.sleep(2)  #  This makes the function take 2s to run
    return a * b

a = 2
b = 21
res = expensive_computation(a, b)

st.write("Result:", res)
```

Try pressing **R** to rerun the app, and notice how long it takes for the result to show up. This is because `expensive_computation(a, b)` is being re-executed every time the app runs. This isn't a great experience.

Let's add the `@st.cache` decorator:

```
import streamlit as st
import time

@st.cache  #  Added this
def expensive_computation(a, b):
    time.sleep(2)  # This makes the function take 2s to run
    return a * b

a = 2
b = 21
res = expensive_computation(a, b)

st.write("Result:", res)
```

Now run the app again and you'll notice that it is much faster every time you press R to rerun. To understand what is happening, let's add an st.write inside the function:

```
import streamlit as st
import time

@st.cache(suppress_st_warning=True)  #  Changed this
def expensive_computation(a, b):
    #  Added this
    st.write("Cache miss: expensive_computation(", a, ",", b, ") ran")
    time.sleep(2)  # This makes the function take 2s to run
    return a * b

a = 2
b = 21
res = expensive_computation(a, b)

st.write("Result:", res)
```

Now when you rerun the app the text "Cache miss" appears on the first run, but not on any subsequent runs. That's because the cached function is only being executed once, and every time after that you're actually hitting the cache.

Note: You may have noticed that we've added the *suppress_st_warning* keyword to the *@st.cache* decorators. That's because the cached function above uses a Streamlit command itself (*st.write* in this case), and when Streamlit sees that, it shows a warning that your command will only execute when you get a cache hit. More often than not, when you see that warning it's because there's a bug in your code. However, in our case we're using the *st.write* command to demonstrate when the cache is being hit, so the behavior Streamlit is warning us about is exactly what we want. As a result, we are passing in *suppress_st_warning=True* to turn that warning off.

15.2 Example 2: When the function arguments change

Without stopping the previous app server, let's change one of the arguments to our cached function:

```
import streamlit as st
import time

@st.cache(suppress_st_warning=True)
def expensive_computation(a, b):
    st.write("Cache miss: expensive_computation(", a, ",", b, ") ran")
    time.sleep(2)  # This makes the function take 2s to run
    return a * b

a = 2
b = 210  #  Changed this
res = expensive_computation(a, b)

st.write("Result:", res)
```

Now the first time you rerun the app it's a cache miss. This is evidenced by the "Cache miss" text showing up and the app taking 2s to finish running. After that, if you press **R** to rerun, it's always a cache hit. That is, no such text shows up and the app is fast again.

This is because Streamlit notices whenever the arguments **a** and **b** change and determines whether the function should be re-executed and re-cached.

15.3 Example 3: When the function body changes

Without stopping and restarting your Streamlit server, let's remove the widget from our app and modify the function's code by adding `a + 1` to the return value.

```
import streamlit as st
import time

@st.cache(suppress_st_warning=True)
def expensive_computation(a, b):
    st.write("Cache miss: expensive_computation(", a, ",", b, ") ran")
    time.sleep(2)  # This makes the function take 2s to run
    return a * b + 1  #  Added a +1 at the end here

a = 2
b = 210
res = expensive_computation(a, b)

st.write("Result:", res)
```

The first run is a "Cache miss", but when you press **R** each subsequent run is a cache hit. This is because on first run, Streamlit detected that the function body changed, reran the function, and put the result in the cache.

Tip: If you change the function back the result will already be in the Streamlit cache from a previous run. Try it out!

15.4 Example 4: When an inner function changes

Let's make our cached function depend on another function internally:

```
import streamlit as st
import time

def inner_func(a, b):
    st.write("inner_func(", a, ",", b, ") ran")
    return a * b

@st.cache(suppress_st_warning=True)
def expensive_computation(a, b):
    st.write("Cache miss: expensive_computation(", a, ",", b, ") ran")
    time.sleep(2)  # This makes the function take 2s to run
    return inner_func(a, b) + 1

a = 2
b = 210
res = expensive_computation(a, b)

st.write("Result:", res)
```

What you see is the usual:

1. The first run results in a cache miss.
2. Every subsequent rerun results in a cache hit.

But now let's try modifying the `inner_func()`:

```
import streamlit as st
import time

def inner_func(a, b):
    st.write("inner_func(", a, ",", b, ") ran")
    return a ** b  #  Changed the * to ** here

@st.cache(suppress_st_warning=True)
def expensive_computation(a, b):
    st.write("Cache miss: expensive_computation(", a, ",", b, ") ran")
    time.sleep(2)  # This makes the function take 2s to run
    return inner_func(a, b) + 1

a = 2
b = 21
res = expensive_computation(a, b)

st.write("Result:", res)
```

Even though `inner_func()` is not annotated with `@st.cache`, when we edit its body we cause a "Cache miss" in the outer `expensive_computation()`.

That's because Streamlit always traverses your code and its dependencies to verify that the cached values are still valid. This means that while developing your app you can edit your code freely without worrying about the cache. Any change you make to your app, Streamlit should do the right thing!

Streamlit is also smart enough to only traverse dependencies that belong to your app, and skip over any dependency that comes from an installed Python library.

15.5 Example 5: Use caching to speed up your app across users

Going back to our original function, let's add a widget to control the value of `b`:

```
import streamlit as st
import time

@st.cache(suppress_st_warning=True)
def expensive_computation(a, b):
    st.write("Cache miss: expensive_computation(", a, ",", b, ") ran")
    time.sleep(2)  # This makes the function take 2s to run
    return a * b

a = 2
b = st.slider("Pick a number", 0, 10)  #  Changed this
res = expensive_computation(a, b)

st.write("Result:", res)
```

What you'll see:

- If you move the slider to a number Streamlit hasn't seen before, you'll have a cache miss again. And every subsequent rerun with the same number will be a cache hit, of course.
- If you move the slider back to a number Streamlit has seen before, the cache is hit and the app is fast as expected.

In computer science terms, what is happening here is that `@st.cache` is memoizing `expensive_computation(a, b)`.

But now let's go one step further! Try the following:

1. Move the slider to a number you haven't tried before, such as 9.
2. Pretend you're another user by opening another browser tab pointing to your Streamlit app (usually at http://localhost:8501)
3. In the new tab, move the slider to 9.

Notice how this is actually a cache hit! That is, you don't actually see the "Cache miss" text on the second tab even though that second user never moved the slider to 9 at any point prior to this.

This happens because the Streamlit cache is global to all users. So everyone contributes to everyone else's performance.

15.6 Example 6: Mutating cached values

As mentioned in the *overview* section, the Streamlit cache stores items by reference. This allows the Streamlit cache to support structures that aren't memory-managed by Python, such as TensorFlow objects. However, it can also lead to unexpected behavior — which is why Streamlit has a few checks to guide developers in the right direction. Let's look into those checks now.

Let's write an app that has a cached function which returns a mutable object, and then let's follow up by mutating that object:

```
import streamlit as st
import time

@st.cache(suppress_st_warning=True)
```

(continues on next page)

(continued from previous page)

```
def expensive_computation(a, b):
    st.write("Cache miss: expensive_computation(", a, ",", b, ") ran")
    time.sleep(2)  # This makes the function take 2s to run
    return {"output": a * b}  #  Mutable object

a = 2
b = 21
res = expensive_computation(a, b)

st.write("Result:", res)

res["output"] = "result was manually mutated"  #  Mutated cached value

st.write("Mutated result:", res)
```

When you run this app for the first time, you should see three messages on the screen:

- Cache miss (. . .)
- Result: {output: 42}
- Mutated result: {output: "result was manually mutated"}

No surprises here. But now notice what happens when you rerun you app (i.e. press **R**):

- Result: {output: "result was manually mutated"}
- Mutated result: {output: "result was manually mutated"}
- Cached object mutated. (. . .)

So what's up?

What's going on here is that Streamlit caches the output `res` by reference. When you mutated `res["output"]` outside the cached function you ended up inadvertently modifying the cache. This means every subsequent call to `expensive_computation(2, 21)` will return the wrong value!

Since this behavior is usually not what you'd expect, Streamlit tries to be helpful and show you a warning, along with some ideas about how to fix your code.

In this specific case, the fix is just to not mutate `res["output"]` outside the cached function. There was no good reason for us to do that anyway! Another solution would be to clone the result value with `res = deepcopy(expensive_computation(2, 21))`. Check out the section entitled *Fixing caching issues* for more information on these approaches and more.

15.7 Advanced caching

In *caching*, you learned about the Streamlit cache, which is accessed with the `@st.cache` decorator. In this article you'll see how Streamlit's caching functionality is implemented, so that you can use it to improve the performance of your Streamlit apps.

The cache is a key-value store, where the key is a hash of:

1. The input parameters that you called the function with
2. The value of any external variable used in the function
3. The body of the function
4. The body of any function used inside the cached function

And the value is a tuple of:

- The cached output
- A hash of the cached output (you'll see why soon)

For both the key and the output hash, Streamlit uses a specialized hash function that knows how to traverse code, hash special objects, and can have its *behavior customized by the user*.

For example, when the function `expensive_computation(a, b)`, decorated with `@st.cache`, is executed with `a=2` and `b=21`, Streamlit does the following:

1. Computes the cache key
2. If the key is found in the cache, then:
    - Extracts the previously-cached (output, output_hash) tuple.
    - Performs an **Output Mutation Check**, where a fresh hash of the output is computed and compared to the stored `output_hash`.
        - If the two hashes are different, shows a **Cached Object Mutated** warning. (Note: Setting `allow_output_mutation=True` disables this step).
3. If the input key is not found in the cache, then:
    - Executes the cached function (i.e. `output = expensive_computation(2, 21)`).
    - Calculates the `output_hash` from the function's `output`.
    - Stores `key` → `(output, output_hash)` in the cache.
4. Returns the output.

If an error is encountered an exception is raised. If the error occurs while hashing either the key or the output an `UnhashableTypeError` error is thrown. If you run into any issues, see *fixing caching issues*.

15.7.1 The `hash_funcs` parameter

As described above, Streamlit's caching functionality relies on hashing to calculate the key for cached objects, and to detect unexpected mutations in the cached result.

For added expressive power, Streamlit lets you override this hashing process using the `hash_funcs` argument. Suppose you define a type called `FileReference` which points to a file in the filesystem:

```
class FileReference:
    def __init__(self, filename):
        self.filename = filename

@st.cache
def func(file_reference):
    ...
```

By default, Streamlit hashes custom classes like `FileReference` by recursively navigating their structure. In this case, its hash is the hash of the filename property. As long as the file name doesn't change, the hash will remain constant.

However, what if you wanted to have the hasher check for changes to the file's modification time, not just its name? This is possible with `@st.cache`'s `hash_funcs` parameter:

```
class FileReference:
    def __init__(self, filename):
        self.filename = filename

def hash_file_reference(file_reference):
    filename = file_reference.filename
    return (filename, os.path.getmtime(filename))

@st.cache(hash_funcs={FileReference: hash_file_reference})
def func(file_reference):
    ...
```

Additionally, you can hash `FileReference` objects by the file's contents:

```
class FileReference:
    def __init__(self, filename):
        self.filename = filename

def hash_file_reference(file_reference):
    with open(file_reference.filename) as f:
      return f.read()

@st.cache(hash_funcs={FileReference: hash_file_reference})
def func(file_reference):
    ...
```

Note: Because Streamlit's hash function works recursively, you don't have to hash the contents inside *hash_file_reference* Instead, you can return a primitive type, in this case the contents of the file, and Streamlit's internal hasher will compute the actual hash from it.

15.7.2 Typical hash functions

While it's possible to write custom hash functions, let's take a look at some of the tools that Python provides out of the box. Here's a list of some hash functions and when it makes sense to use them.

Python's `id` function | *Example*

- Speed: Fast
- Use case: If you're hashing a singleton object, like an open database connection or a TensorFlow session. These are objects that will only be instantiated once, no matter how many times your script reruns.

`lambda _: None` | *Example*

- Speed: Fast
- Use case: If you want to turn off hashing of this type. This is useful if you know the object is not going to change.

Python's `hash()` function | *Example*

- Speed: Can be slow based the size of the object being cached
- Use case: If Python already knows how to hash this type correctly.

Custom hash function | *Example*

- Speed: N/a

- Use case: If you'd like to override how Streamlit hashes a particular type.

15.7.3 Example 1: Pass a database connection around

Suppose we want to open a database connection that can be reused across multiple runs of a Streamlit app. For this you can make use of the fact that cached objects are stored by reference to automatically initialize and reuse the connection:

```
@st.cache(allow_output_mutation=True)
def get_database_connection():
    return db.get_connection()
```

With just 3 lines of code, the database connection is created once and stored in the cache. Then, every subsequent time `get_database_conection` is called, the already-created connection object is reused automatically. In other words, it becomes a singleton.

Tip: Use the *allow_output_mutation=True* flag to suppress the immutability check. This prevents Streamlit from trying to hash the output connection, and also turns off Streamlit's mutation warning in the process.

What if you want to write a function that receives a database connection as input? For that, you'll use `hash_funcs`:

```
@st.cache(hash_funcs={DBConnection: id})
def get_users(connection):
    # Note: We assume that connection is of type DBConnection.
    return connection.execute_sql('SELECT * from Users')
```

Here, we use Python's built-in `id` function, because the connection object is coming from the Streamlit cache via the `get_database_conection` function. This means that the same connection instance is passed around every time, and therefore it always has the same id. However, if you happened to have a second connection object around that pointed to an entirely different database, it would still be safe to pass it to `get_users` because its id is guaranteed to be different than the first id.

These design patterns apply any time you have an object that points to an external resource, such as a database connection or Tensorflow session.

15.7.4 Example 2: Turn off hashing for a specific type

You can turn off hashing entirely for a particular type by giving it a custom hash function that returns a constant. One reason that you might do this is to avoid hashing large, slow-to-hash objects that you know are not going to change. For example:

```
@st.cache(hash_funcs={pd.DataFrame: lambda _: None})
def func(huge_constant_dataframe):
    ...
```

When Streamlit encounters an object of this type, it always converts the object into `None`, no matter which instance of `FooType` its looking at. This means all instances are hash to the same value, which effectively cancels out the hashing mechanism.

15.7.5 Example 3: Use Python's `hash()` function

Sometimes, you might want to use Python's default hashing instead of Streamlit's. For example, maybe you've encountered a type that Streamlit is unable to hash, but it's hashable with Python's built-in `hash()` function:

```
@st.cache(hash_funcs={FooType: hash})
def func(...):
    ...
```

CHAPTER

SIXTEEN

COOKBOOK

Now that you've mastered Streamlit's main concepts, let's take a look at some advanced functionality, like styling data, adjusting the order of elements in a report, and adding animations.

Note: Have something to add? Please let us know what's important to you! Ping us in the community forum.

16.1 Insert elements out of order

You can use the `st.empty` method as a placeholder, to "save" a slot in your app that you can use later.

```
st.text('This will appear first')
# Appends some text to the app.

my_slot1 = st.empty()
# Appends an empty slot to the app. We'll use this later.

my_slot2 = st.empty()
# Appends another empty slot.

st.text('This will appear last')
# Appends some more text to the app.

my_slot1.text('This will appear second')
# Replaces the first empty slot with a text string.

my_slot2.line_chart(np.random.randn(20, 2))
# Replaces the second empty slot with a chart.
```

16.2 Animate elements

Let's combine some of the things you've learned to create compelling animations in your app.

```
progress_bar = st.progress(0)
status_text = st.empty()
chart = st.line_chart(np.random.randn(10, 2))

for i in range(100):
    # Update progress bar.
```

(continues on next page)

(continued from previous page)

```
    progress_bar.progress(i + 1)

    new_rows = np.random.randn(10, 2)

    # Update status text.
    status_text.text(
        'The latest random number is: %s' % new_rows[-1, 1])

    # Append data to the chart.
    chart.add_rows(new_rows)

    # Pretend we're doing some computation that takes time.
    time.sleep(0.1)

status_text.text('Done!')
st.balloons()
```

16.3 Append data to a table or chart

In Streamlit, you can not only replace entire elements in your app, but also modify the data behind those elements. Here is how:

```
import numpy as np
import time

# Get some data.
data = np.random.randn(10, 2)

# Show the data as a chart.
chart = st.line_chart(data)

# Wait 1 second, so the change is clearer.
time.sleep(1)

# Grab some more data.
data2 = np.random.randn(10, 2)

# Append the new data to the existing chart.
chart.add_rows(data2)
```

16.4 Record a screencast

After you've built a Streamlit app, you may want to discuss some of it with co-workers over email or Slack, or share it with the world on Twitter. A great way to do that is with Streamlit's built-in screencast recorder. With it, you can record, narrate, stop, save, and share with a few clicks.

To start a screencast, locate the menu in the upper right corner of your app (), select **Record a screencast**, and follow the prompts. Before the recording starts, you'll see a countdown — this means it's showtime.

To stop your screencast, go back to the menu () and select **Stop recording** (or hit the **ESC** key). Follow the prompts to preview your recording and save it to disk. That's it, you're ready to share your Streamlit app.

CHAPTER

SEVENTEEN

EXTEND YOUR APP WITH COMPONENTS

17.1 Publish to PyPI

Publishing your Streamlit Component to PyPI makes it easily accessible to Python users around the world. This step is completely optional, so if you won't be releasing your component publicly, you can skip this section!

Note: For static Streamlit Components, publishing a Python package to PyPI follows the same steps as the core PyPI packaging instructions. A static Component likely contains only Python code, so once you have your setup.py file correct and generate your distribution files, you're ready to upload to PyPI.

Bi-directional Streamlit Components at minimum include both Python and JavaScript code, and as such, need a bit more preparation before they can be published on PyPI. The remainder of this page focuses on the bi-directional Component preparation process.

17.1.1 Prepare your Component

A bi-directional Streamlit Component varies slightly from a pure Python library in that it must contain pre-compiled frontend code. This is how base Streamlit works as well; when you `pip install streamlit`, you are getting a Python library where the HTML and frontend code contained within it have been compiled into static assets.

The component-template GitHub repo provides the folder structure necessary for PyPI publishing. But before you can publish, you'll need to do a bit of housekeeping:

1. Give your Component a name, if you haven't already
   - Rename the `template/my_component/` folder to `template/<component name>/`
   - Pass your component's name as the the first argument to `declare_component()`
2. Edit `MANIFEST.in`, change the path for recursive-include from `package/frontend/build *` to `<component name>/frontend/build *`
3. Edit `setup.py`, adding your component's name and other relevant info
4. Create a release build of your frontend code. This will add a new directory, `frontend/build/`, with your compiled frontend in it:

```
$ cd frontend
$ npm run build
```

5. Pass the build folder's path as the `path` parameter to `declare_component`. (If you're using the template Python file, you can set `_RELEASE = True` at the top of the file):

```
    import streamlit.components.v1 as components

    # Change this:
    # component = components.declare_component("my_component", url="http://
↪localhost:3001")

    # To this:
    parent_dir = os.path.dirname(os.path.abspath(__file__))
    build_dir = os.path.join(parent_dir, "frontend/build")
    component = components.declare_component("new_component_name", path=build_dir)
```

17.1.2 Build a Python wheel

Once you've changed the default `my_component` references, compiled the HTML and JavaScript code and set your new component name in `components.declare_component()`, you're ready to build a Python wheel:

1. Make sure you have the latest versions of setuptools, wheel, and twine
2. Create a wheel from the source code:

```
# Run this from your component's top-level directory; that is,
# the directory that contains `setup.py`
$ python setup.py sdist bdist_wheel
```

17.1.3 Upload your wheel to PyPI

With your wheel created, the final step is to upload to PyPI. The instructions here highlight how to upload to Test PyPI, so that you can learn the mechanics of the process without worrying about messing anything up. Uploading to PyPI follows the same basic procedure.

1. Create an account on Test PyPI if you don't already have one
   - Visit https://test.pypi.org/account/register/ and complete the steps
   - Visit https://test.pypi.org/manage/account/#api-tokens and create a new API token. Don't limit the token scope to a particular project, since you are creating a new project. Copy your token before closing the page, as you won't be able to retrieve it again.
2. Upload your wheel to Test PyPI. `twine` will prompt you for a username and password. For the username, use **token**. For the password, use your token value from the previous step, including the `pypi-` prefix:

```
python3 -m twine upload --repository testpypi dist/*
```

3. Install your newly-uploaded package in a new Python project to make sure it works:

```
python -m pip install --index-url https://test.pypi.org/simple/ --no-deps␣
↪example-pkg-YOUR-USERNAME-HERE
```

If all goes well, you're ready to upload your library to PyPI by following the instructions at https://packaging.python.org/tutorials/packaging-projects/#next-steps.

Congratulations, you've created a publicly-available Streamlit Component!

17.2 Promote your Component!

We'd love to help you share your Component with the Streamlit Community! To share it, please post on the Streamlit 'Show the Community!' Forum category with the title similar to "New Component: `<your component name>`, a new way to do X".

You can also Tweet at us @streamlit so that we can retweet your announcement for you.

CHAPTER

EIGHTEEN

API REFERENCE

Streamlit makes it easy for you to visualize, mutate, and share data. The API reference is organized by activity type, like displaying data or optimizing performance. Each section includes methods associated with the activity type, including examples.

Know what you're looking for? Use these links or the left nav to move through this API reference.

18.1 Magic commands

Magic commands are a feature in Streamlit that allows you to write markdown and data to your app with very few keypresses. Here's an example:

```
# Draw a title and some text to the app:
'''
# This is the document title

This is some _markdown_.
'''

df = pd.DataFrame({'col1': [1,2,3]})
df  # <-- Draw the dataframe

x = 10
'x', x  # <-- Draw the string 'x' and then the value of x
```

Any time Streamlit sees either a variable or literal value on its own line, it automatically writes that to your app using `st.write` (which you'll learn about later).

Also, magic is smart enough to ignore docstrings. That is, it ignores the strings at the top of files and functions.

If you prefer to call Streamlit commands more explicitly, you can always turn magic off in your `~/.streamlit/config.toml` with the following setting:

```
[runner]
magicEnabled = false
```

Important: Right now, Magic only works in the main Python app file, not in imported files. See GitHub issue #288 for a discussion of the issues.

18.2 Display text

Streamlit apps usually start with a call to `st.title` to set the app's title. After that, there are 2 heading levels you can use: `st.header` and `st.subheader`.

Pure text is entered with `st.text`, and Markdown with `st.markdown`.

We also offer a "swiss-army knife" command called `st.write`, which accepts multiple arguments, and multiple data types. And as described above, you can also use magic commands in place of `st.write`.

`streamlit.`**`text`**(*body*)

Write fixed-width and preformatted text.

Parameters **`body`** (*str*) – The string to display.

Example

```
>>> st.text('This is some text.')
```

streamlit.**markdown**(*body*, *unsafe_allow_html=False*)

Display string formatted as Markdown.

Parameters

- **body** (*str*) – The string to display as Github-flavored Markdown. Syntax information can be found at: https://github.github.com/gfm.

  This also supports:

  – Emoji shortcodes, such as *:+1:* and *:sunglasses:*. For a list of all supported codes, see https://raw.githubusercontent.com/omnidan/node-emoji/master/lib/emoji.json.

  – LaTeX expressions, by wrapping them in "$" or "$$" (the "$$" must be on their own lines). Supported LaTeX functions are listed at https://katex.org/docs/supported.html.

- **unsafe_allow_html** (*bool*) – By default, any HTML tags found in the body will be escaped and therefore treated as pure text. This behavior may be turned off by setting this argument to True.

  That said, we *strongly advise against it*. It is hard to write secure HTML, so by using this argument you may be compromising your users' security. For more information, see:

  https://github.com/streamlit/streamlit/issues/152

  Also note that `unsafe_allow_html` is a temporary measure and may be removed from Streamlit at any time.

  If you decide to turn on HTML anyway, we ask you to please tell us your exact use case here:

  https://discuss.streamlit.io/t/96

  This will help us come up with safe APIs that allow you to do what you want.

Example

```
>>> st.markdown('Streamlit is **_really_ cool**.')
```

streamlit.**latex**(*body*)

Display mathematical expressions formatted as LaTeX.

Supported LaTeX functions are listed at https://katex.org/docs/supported.html.

Parameters **body** (*str or SymPy expression*) – The string or SymPy expression to display as LaTeX. If str, it's a good idea to use raw Python strings since LaTeX uses backslashes a lot.

Example

```
>>> st.latex(r'''
...     a + ar + a r^2 + a r^3 + \cdots + a r^{n-1} =
...     \sum_{k=0}^{n-1} ar^k =
...     a \left(\frac{1-r^{n}}{1-r}\right)
...     ''')
```

streamlit.**write**(**args*, ***kwargs*)

Write arguments to the app.

This is the Swiss Army knife of Streamlit commands: it does different things depending on what you throw at it. Unlike other Streamlit commands, write() has some unique properties:

1. You can pass in multiple arguments, all of which will be written.
2. Its behavior depends on the input types as follows.
3. It returns None, so it's "slot" in the App cannot be reused.

Parameters

- ***args** (*any*) – One or many objects to print to the App.

  Arguments are handled as follows:

  - **write(string)** [Prints the formatted Markdown string, with] support for LaTeX expression and emoji shortcodes. See docs for st.markdown for more.
  - write(data_frame) : Displays the DataFrame as a table.
  - write(error) : Prints an exception specially.
  - write(func) : Displays information about a function.
  - write(module) : Displays information about the module.
  - write(dict) : Displays dict in an interactive widget.
  - write(obj) : The default is to print str(obj).
  - write(mpl_fig) : Displays a Matplotlib figure.
  - write(altair) : Displays an Altair chart.
  - write(keras) : Displays a Keras model.
  - write(graphviz) : Displays a Graphviz graph.
  - write(plotly_fig) : Displays a Plotly figure.
  - write(bokeh_fig) : Displays a Bokeh figure.
  - write(sympy_expr) : Prints SymPy expression using LaTeX.

- **unsafe_allow_html** (*bool*) – This is a keyword-only argument that defaults to False.

  By default, any HTML tags found in strings will be escaped and therefore treated as pure text. This behavior may be turned off by setting this argument to True.

  That said, *we strongly advise* against it*. It is hard to write secure HTML, so by using this argument you may be compromising your users' security. For more information, see:

  https://github.com/streamlit/streamlit/issues/152

Also note that `unsafe_allow_html` is a temporary measure and may be removed from Streamlit at any time.

If you decide to turn on HTML anyway, we ask you to please tell us your exact use case here: https://discuss.streamlit.io/t/96 .

This will help us come up with safe APIs that allow you to do what you want.

Example

Its basic use case is to draw Markdown-formatted text, whenever the input is a string:

```
>>> write('Hello, *World!* :sunglasses:')
```

As mentioned earlier, *st.write()* also accepts other data formats, such as numbers, data frames, styled data frames, and assorted objects:

```
>>> st.write(1234)
>>> st.write(pd.DataFrame({
...     'first column': [1, 2, 3, 4],
...     'second column': [10, 20, 30, 40],
... }))
```

Finally, you can pass in multiple arguments to do things like:

```
>>> st.write('1 + 1 = ', 2)
>>> st.write('Below is a DataFrame:', data_frame, 'Above is a dataframe.')
```

Oh, one more thing: *st.write* accepts chart objects too! For example:

```
>>> import pandas as pd
>>> import numpy as np
>>> import altair as alt
>>>
>>> df = pd.DataFrame(
...     np.random.randn(200, 3),
...     columns=['a', 'b', 'c'])
...
>>> c = alt.Chart(df).mark_circle().encode(
...     x='a', y='b', size='c', color='c', tooltip=['a', 'b', 'c'])
>>>
>>> st.write(c)
```

streamlit.**title**(*body*)

Display text in title formatting.

Each document should have a single *st.title()*, although this is not enforced.

Parameters **body** (*str*) – The text to display.

Example

```
>>> st.title('This is a title')
```

streamlit.**header**(*body*)
Display text in header formatting.

Parameters **body** (*str*) – The text to display.

Example

```
>>> st.header('This is a header')
```

streamlit.**subheader**(*body*)
Display text in subheader formatting.

Parameters **body** (*str*) – The text to display.

Example

```
>>> st.subheader('This is a subheader')
```

streamlit.**code**(*body*, *language='python'*)
Display a code block with optional syntax highlighting.

(This is a convenience wrapper around *st.markdown()*)

Parameters

- **body** (*str*) – The string to display as code.
- **language** (*str*) – The language that the code is written in, for syntax highlighting. If omitted, the code will be unstyled.

Example

```
>>> code = '''def hello():
...     print("Hello, Streamlit!")'''
>>> st.code(code, language='python')
```

18.3 Display data

When you're working with data, it is extremely valuable to visualize that data quickly, interactively, and from multiple different angles. That's what Streamlit is actually built and optimized for.

You can display data via *charts*, and you can display it in raw form. These are the Streamlit commands you can use to display raw data.

streamlit.**dataframe**(*data=None*, *width=None*, *height=None*)
Display a dataframe as an interactive table.

Parameters

- **data** (*pandas.DataFrame, pandas.Styler, numpy.ndarray, Iterable, dict,*) – or None The data to display.

  If 'data' is a pandas.Styler, it will be used to style its underyling DataFrame. Streamlit supports custom cell values and colors. (It does not support some of the more exotic pandas styling features, like bar charts, hovering, and captions.) Styler support is experimental!

- **width** (*int or None*) – Desired width of the UI element expressed in pixels. If None, a default width based on the page width is used.
- **height** (*int or None*) – Desired height of the UI element expressed in pixels. If None, a default height is used.

Examples

```
>>> df = pd.DataFrame(
...     np.random.randn(50, 20),
...     columns=('col %d' % i for i in range(20)))
...
>>> st.dataframe(df)  # Same as st.write(df)
```

```
>>> st.dataframe(df, 200, 100)
```

You can also pass a Pandas Styler object to change the style of the rendered DataFrame:

```
>>> df = pd.DataFrame(
...     np.random.randn(10, 20),
...     columns=('col %d' % i for i in range(20)))
...
>>> st.dataframe(df.style.highlight_max(axis=0))
```

streamlit.**table**(*data=None*)

Display a static table.

This differs from *st.dataframe* in that the table in this case is static: its entire contents are laid out directly on the page.

Parameters **data** (*pandas.DataFrame, pandas.Styler, numpy.ndarray, Iterable, dict,*) – or None The table data.

Example

```
>>> df = pd.DataFrame(
...     np.random.randn(10, 5),
...     columns=('col %d' % i for i in range(5)))
...
>>> st.table(df)
```

streamlit.**json**(*body*)

Display object or string as a pretty-printed JSON string.

Parameters **body** (*Object or str*) – The object to print as JSON. All referenced objects should be serializable to JSON as well. If object is a string, we assume it contains serialized JSON.

Example

```
>>> st.json({
...     'foo': 'bar',
...     'baz': 'boz',
...     'stuff': [
...         'stuff 1',
...         'stuff 2',
...         'stuff 3',
...         'stuff 5',
...     ],
... })
```

18.4 Display charts

Streamlit supports several different charting libraries, and our goal is to continually add support for more. Right now, the most basic library in our arsenal is Matplotlib. Then there are also interactive charting libraries like Vega Lite (2D charts) and deck.gl (maps and 3D charts). And finally we also provide a few chart types that are "native" to Streamlit, like `st.line_chart` and `st.area_chart`.

`streamlit.`**`line_chart`**(*data=None*, *width=0*, *height=0*, *use_container_width=True*)

Display a line chart.

This is syntax-sugar around st.altair_chart. The main difference is this command uses the data's own column and indices to figure out the chart's spec. As a result this is easier to use for many "just plot this" scenarios, while being less customizable.

If st.line_chart does not guess the data specification correctly, try specifying your desired chart using st.altair_chart.

Parameters

- **`data`** (*`pandas.DataFrame`, `pandas.Styler`, `numpy.ndarray`, `Iterable`, `dict`*) – or None Data to be plotted.
- **`width`** (*`int`*) – The chart width in pixels. If 0, selects the width automatically.
- **`height`** (*`int`*) – The chart width in pixels. If 0, selects the height automatically.
- **`use_container_width`** (*`bool`*) – If True, set the chart width to the column width. This takes precedence over the width argument.

Example

```
>>> chart_data = pd.DataFrame(
...     np.random.randn(20, 3),
...     columns=['a', 'b', 'c'])
...
>>> st.line_chart(chart_data)
```

`streamlit.`**`area_chart`**(*data=None*, *width=0*, *height=0*, *use_container_width=True*)

Display an area chart.

This is just syntax-sugar around st.altair_chart. The main difference is this command uses the data's own column and indices to figure out the chart's spec. As a result this is easier to use for many "just plot this" scenarios, while being less customizable.

If st.area_chart does not guess the data specification correctly, try specifying your desired chart using st.altair_chart.

Parameters

- **data** (*pandas.DataFrame, pandas.Styler, numpy.ndarray, Iterable, or dict*) – Data to be plotted.
- **width** (*int*) – The chart width in pixels. If 0, selects the width automatically.
- **height** (*int*) – The chart width in pixels. If 0, selects the height automatically.
- **use_container_width** (*bool*) – If True, set the chart width to the column width. This takes precedence over the width argument.

Example

```
>>> chart_data = pd.DataFrame(
...     np.random.randn(20, 3),
...     columns=['a', 'b', 'c'])
...
>>> st.area_chart(chart_data)
```

streamlit.**bar_chart**(*data=None*, *width=0*, *height=0*, *use_container_width=True*)

Display a bar chart.

This is just syntax-sugar around st.altair_chart. The main difference is this command uses the data's own column and indices to figure out the chart's spec. As a result this is easier to use for many "just plot this" scenarios, while being less customizable.

If st.bar_chart does not guess the data specification correctly, try specifying your desired chart using st.altair_chart.

Parameters

- **data** (*pandas.DataFrame, pandas.Styler, numpy.ndarray, Iterable, or dict*) – Data to be plotted.
- **width** (*int*) – The chart width in pixels. If 0, selects the width automatically.
- **height** (*int*) – The chart width in pixels. If 0, selects the height automatically.
- **use_container_width** (*bool*) – If True, set the chart width to the column width. This takes precedence over the width argument.

Example

```
>>> chart_data = pd.DataFrame(
...     np.random.randn(50, 3),
...     columns=["a", "b", "c"])
...
>>> st.bar_chart(chart_data)
```

streamlit.**pyplot**(*fig=None*, *clear_figure=None*, ***kwargs*)

Display a matplotlib.pyplot figure.

Parameters

- **fig** (*Matplotlib Figure*) – The figure to plot. When this argument isn't specified, this function will render the global figure (but this is deprecated, as described below)

- **clear_figure** (*bool*) – If True, the figure will be cleared after being rendered. If False, the figure will not be cleared after being rendered. If left unspecified, we pick a default based on the value of *fig*.
  - If *fig* is set, defaults to *False*.
  - If *fig* is not set, defaults to *True*. This simulates Jupyter's approach to matplotlib rendering.
- ****kwargs** (*any*) – Arguments to pass to Matplotlib's savefig function.

Example

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>>
>>> arr = np.random.normal(1, 1, size=100)
>>> fig, ax = plt.subplots()
>>> ax.hist(arr, bins=20)
>>>
>>> st.pyplot(fig)
```

Notes

Note: Deprecation warning. After December 1st, 2020, we will remove the ability to specify no arguments in *st.pyplot()*, as that requires the use of Matplotlib's global figure object, which is not thread-safe. So please always pass a figure object as shown in the example section above.

Matplotlib support several different types of "backends". If you're getting an error using Matplotlib with Streamlit, try setting your backend to "TkAgg":

```
echo "backend: TkAgg" >> ~/.matplotlib/matplotlibrc
```

For more information, see https://matplotlib.org/faq/usage_faq.html.

streamlit.**altair_chart**(*altair_chart*, *use_container_width=False*)

Display a chart using the Altair library.

Parameters

- **altair_chart** (*altair.vegalite.v2.api.Chart*) – The Altair chart object to display.
- **use_container_width** (*bool*) – If True, set the chart width to the column width. This takes precedence over Altair's native *width* value.

Example

```
>>> import pandas as pd
>>> import numpy as np
>>> import altair as alt
>>>
>>> df = pd.DataFrame(
...     np.random.randn(200, 3),
...     columns=['a', 'b', 'c'])
...
>>> c = alt.Chart(df).mark_circle().encode(
...     x='a', y='b', size='c', color='c', tooltip=['a', 'b', 'c'])
>>>
>>> st.altair_chart(c, use_container_width=True)
```

Examples of Altair charts can be found at https://altair-viz.github.io/gallery/.

streamlit.**vega_lite_chart**(*data=None*, *spec=None*, *use_container_width=False*, ***kwargs*)

Display a chart using the Vega-Lite library.

Parameters

- **data** (*pandas.DataFrame, pandas.Styler, numpy.ndarray, Iterable, dict,*) – or None Either the data to be plotted or a Vega-Lite spec containing the data (which more closely follows the Vega-Lite API).
- **spec** (*dict or None*) – The Vega-Lite spec for the chart. If the spec was already passed in the previous argument, this must be set to None. See https://vega.github.io/vega-lite/docs/ for more info.
- **use_container_width** (*bool*) – If True, set the chart width to the column width. This takes precedence over Vega-Lite's native *width* value.
- ****kwargs** (*any*) – Same as spec, but as keywords.

Example

```
>>> import pandas as pd
>>> import numpy as np
>>>
>>> df = pd.DataFrame(
...     np.random.randn(200, 3),
...     columns=['a', 'b', 'c'])
>>>
>>> st.vega_lite_chart(df, {
...     'mark': {'type': 'circle', 'tooltip': True},
...     'encoding': {
...         'x': {'field': 'a', 'type': 'quantitative'},
...         'y': {'field': 'b', 'type': 'quantitative'},
...         'size': {'field': 'c', 'type': 'quantitative'},
...         'color': {'field': 'c', 'type': 'quantitative'},
...     },
... })
```

Examples of Vega-Lite usage without Streamlit can be found at https://vega.github.io/vega-lite/examples/. Most of those can be easily translated to the syntax shown above.

streamlit.**plotly_chart**(*figure_or_data*, *use_container_width=False*, *sharing='streamlit'*, ***kwargs*)

Display an interactive Plotly chart.

Plotly is a charting library for Python. The arguments to this function closely follow the ones for Plotly's *plot()* function. You can find more about Plotly at https://plot.ly/python.

Parameters

- **figure_or_data** (*plotly.graph_objs.Figure, plotly.graph_objs.Data,*) – dict/list of plotly.graph_objs.Figure/Data

  See https://plot.ly/python/ for examples of graph descriptions.

- **use_container_width** (*bool*) – If True, set the chart width to the column width. This takes precedence over the figure's native *width* value.

- **sharing** (*{'streamlit', 'private', 'secret', 'public'}*) – Use 'streamlit' to insert the plot and all its dependencies directly in the Streamlit app using plotly's offline mode (default). Use any other sharing mode to send the chart to Plotly chart studio, which requires an account. See https://plotly.com/chart-studio/ for more information.

- ****kwargs** – Any argument accepted by Plotly's *plot()* function.

To show Plotly charts in Streamlit, call *st.plotly_chart* wherever you would call Plotly's *py.plot* or *py.iplot*.

Example

The example below comes straight from the examples at https://plot.ly/python:

```
>>> import streamlit as st
>>> import plotly.figure_factory as ff
>>> import numpy as np
>>>
>>> # Add histogram data
>>> x1 = np.random.randn(200) - 2
>>> x2 = np.random.randn(200)
>>> x3 = np.random.randn(200) + 2
>>>
>>> # Group data together
>>> hist_data = [x1, x2, x3]
>>>
>>> group_labels = ['Group 1', 'Group 2', 'Group 3']
>>>
>>> # Create distplot with custom bin_size
>>> fig = ff.create_distplot(
...         hist_data, group_labels, bin_size=[.1, .25, .5])
>>>
>>> # Plot!
>>> st.plotly_chart(fig, use_container_width=True)
```

streamlit.**bokeh_chart**(*figure*, *use_container_width=False*)

Display an interactive Bokeh chart.

Bokeh is a charting library for Python. The arguments to this function closely follow the ones for Bokeh's *show* function. You can find more about Bokeh at https://bokeh.pydata.org.

Parameters

- **figure** (*bokeh.plotting.figure.Figure*) – A Bokeh figure to plot.

- **use_container_width** (*bool*) – If True, set the chart width to the column width. This takes precedence over Bokeh's native *width* value.

- **show Bokeh charts in Streamlit, call st.bokeh_chart** (*To*) –
- **you would call Bokeh's show.** (*wherever*) –

Example

```
>>> import streamlit as st
>>> from bokeh.plotting import figure
>>>
>>> x = [1, 2, 3, 4, 5]
>>> y = [6, 7, 2, 4, 5]
>>>
>>> p = figure(
...     title='simple line example',
...     x_axis_label='x',
...     y_axis_label='y')
...
>>> p.line(x, y, legend='Trend', line_width=2)
>>>
>>> st.bokeh_chart(p, use_container_width=True)
```

streamlit.**pydeck_chart**(*pydeck_obj=None*, *use_container_width=False*)

Draw a chart using the PyDeck library.

This supports 3D maps, point clouds, and more! More info about PyDeck at https://deckgl.readthedocs.io/en/latest/.

These docs are also quite useful:

- DeckGL docs: https://github.com/uber/deck.gl/tree/master/docs
- DeckGL JSON docs: https://github.com/uber/deck.gl/tree/master/modules/json

When using this command, we advise all users to use a personal Mapbox token. This ensures the map tiles used in this chart are more robust. You can do this with the mapbox.token config option.

To get a token for yourself, create an account at https://mapbox.com. It's free! (for moderate usage levels) See https://docs.streamlit.io/en/latest/cli.html#view-all-config-options for more info on how to set config options.

Parameters **spec** (*pydeck.Deck or None*) – Object specifying the PyDeck chart to draw.

Example

Here's a chart using a HexagonLayer and a ScatterplotLayer on top of the light map style:

```
>>> df = pd.DataFrame(
...     np.random.randn(1000, 2) / [50, 50] + [37.76, -122.4],
...     columns=['lat', 'lon'])
>>>
>>> st.pydeck_chart(pdk.Deck(
...      map_style='mapbox://styles/mapbox/light-v9',
...      initial_view_state=pdk.ViewState(
...          latitude=37.76,
...          longitude=-122.4,
...          zoom=11,
...          pitch=50,
...      ),
...      layers=[
```

(continues on next page)

(continued from previous page)

```
...         pdk.Layer(
...            'HexagonLayer',
...            data=df,
...            get_position='[lon, lat]',
...            radius=200,
...            elevation_scale=4,
...            elevation_range=[0, 1000],
...            pickable=True,
...            extruded=True,
...         ),
...         pdk.Layer(
...             'ScatterplotLayer',
...             data=df,
...             get_position='[lon, lat]',
...             get_color='[200, 30, 0, 160]',
...             get_radius=200,
...         ),
...     ],
... ))
```

streamlit.**graphviz_chart**(*figure_or_dot*, *use_container_width=False*)

Display a graph using the dagre-d3 library.

Parameters

- **figure_or_dot** (*graphviz.dot.Graph, graphviz.dot.Digraph, str*) – The Graphlib graph object or dot string to display
- **use_container_width** (*bool*) – If True, set the chart width to the column width. This takes precedence over the figure's native *width* value.

Example

```
>>> import streamlit as st
>>> import graphviz as graphviz
>>>
>>> # Create a graphlib graph object
>>> graph = graphviz.Digraph()
>>> graph.edge('run', 'intr')
>>> graph.edge('intr', 'runbl')
>>> graph.edge('runbl', 'run')
>>> graph.edge('run', 'kernel')
>>> graph.edge('kernel', 'zombie')
>>> graph.edge('kernel', 'sleep')
>>> graph.edge('kernel', 'runmem')
>>> graph.edge('sleep', 'swap')
>>> graph.edge('swap', 'runswap')
>>> graph.edge('runswap', 'new')
>>> graph.edge('runswap', 'runmem')
>>> graph.edge('new', 'runmem')
>>> graph.edge('sleep', 'runmem')
>>>
>>> st.graphviz_chart(graph)
```

Or you can render the chart from the graph using GraphViz's Dot language:

```
>>> st.graphviz_chart('''
    digraph {
        run -> intr
        intr -> runbl
        runbl -> run
        run -> kernel
        kernel -> zombie
        kernel -> sleep
        kernel -> runmem
        sleep -> swap
        swap -> runswap
        runswap -> new
        runswap -> runmem
        new -> runmem
        sleep -> runmem
    }
''')
```

streamlit.**map**(*data=None*, *zoom=None*, *use_container_width=True*)

Display a map with points on it.

This is a wrapper around st.pydeck_chart to quickly create scatterplot charts on top of a map, with auto-centering and auto-zoom.

When using this command, we advise all users to use a personal Mapbox token. This ensures the map tiles used in this chart are more robust. You can do this with the mapbox.token config option.

To get a token for yourself, create an account at https://mapbox.com. It's free! (for moderate usage levels) See https://docs.streamlit.io/en/latest/cli.html#view-all-config-options for more info on how to set config options.

Parameters

- **data** (*pandas.DataFrame, pandas.Styler, numpy.ndarray, Iterable, dict,*) – or None The data to be plotted. Must have columns called 'lat', 'lon', 'latitude', or 'longitude'.
- **zoom** (*int*) – Zoom level as specified in https://wiki.openstreetmap.org/wiki/Zoom_levels

Example

```
>>> import pandas as pd
>>> import numpy as np
>>>
>>> df = pd.DataFrame(
...     np.random.randn(1000, 2) / [50, 50] + [37.76, -122.4],
...     columns=['lat', 'lon'])
>>>
>>> st.map(df)
```

18.5 Display media

It's easy to embed images, videos, and audio files directly into your Streamlit apps.

streamlit.**image**(*image*, *caption=None*, *width=None*, *use_column_width=False*, *clamp=False*, *channels='RGB'*, *output_format='auto'*, ***kwargs*)

Display an image or list of images.

Parameters

- **image** (*numpy.ndarray, [numpy.ndarray], BytesIO, str, or [str]*) – Monochrome image of shape (w,h) or (w,h,1) OR a color image of shape (w,h,3) OR an RGBA image of shape (w,h,4) OR a URL to fetch the image from OR an SVG XML string like <*svg xmlns*=... </*svg*> OR a list of one of the above, to display multiple images.
- **caption** (*str or list of str*) – Image caption. If displaying multiple images, caption should be a list of captions (one for each image).
- **width** (*int or None*) – Image width. None means use the image width. Should be set for SVG images, as they have no default image width.
- **use_column_width** (*bool*) – If True, set the image width to the column width. This takes precedence over the *width* parameter.
- **clamp** (*bool*) – Clamp image pixel values to a valid range ([0-255] per channel). This is only meaningful for byte array images; the parameter is ignored for image URLs. If this is not set, and an image has an out-of-range value, an error will be thrown.
- **channels** (*'RGB' or 'BGR'*) – If image is an nd.array, this parameter denotes the format used to represent color information. Defaults to 'RGB', meaning *image[:, :, 0]* is the red channel, *image[:, :, 1]* is green, and *image[:, :, 2]* is blue. For images coming from libraries like OpenCV you should set this to 'BGR', instead.
- **output_format** (*'JPEG', 'PNG', or 'auto'*) – This parameter specifies the format to use when transferring the image data. Photos should use the JPEG format for lossy compression while diagrams should use the PNG format for lossless compression. Defaults to 'auto' which identifies the compression type based on the type and format of the image argument.

Example

```
>>> from PIL import Image
>>> image = Image.open('sunrise.jpg')
>>>
>>> st.image(image, caption='Sunrise by the mountains',
...          use_column_width=True)
```

streamlit.**audio**(*data*, *format='audio/wav'*, *start_time=0*)

Display an audio player.

Parameters

- **data** (*str, bytes, BytesIO, numpy.ndarray, or file opened with*) – io.open(). Raw audio data, filename, or a URL pointing to the file to load. Numpy arrays and raw data formats must include all necessary file headers to match specified file format.
- **start_time** (*int*) – The time from which this element should start playing.

- **format** (*str*) – The mime type for the audio file. Defaults to 'audio/wav'. See https://tools.ietf.org/html/rfc4281 for more info.

Example

```
>>> audio_file = open('myaudio.ogg', 'rb')
>>> audio_bytes = audio_file.read()
>>>
>>> st.audio(audio_bytes, format='audio/ogg')
```

streamlit.**video**(*data*, *format='video/mp4'*, *start_time=0*)

Display a video player.

Parameters

- **data** (*str, bytes, BytesIO, numpy.ndarray, or file opened with*) – io.open(). Raw video data, filename, or URL pointing to a video to load. Includes support for YouTube URLs. Numpy arrays and raw data formats must include all necessary file headers to match specified file format.
- **format** (*str*) – The mime type for the video file. Defaults to 'video/mp4'. See https://tools.ietf.org/html/rfc4281 for more info.
- **start_time** (*int*) – The time from which this element should start playing.

Example

```
>>> video_file = open('myvideo.mp4', 'rb')
>>> video_bytes = video_file.read()
>>>
>>> st.video(video_bytes)
```

Note: Some videos may not display if they are encoded using MP4V (which is an export option in OpenCV), as this codec is not widely supported by browsers. Converting your video to H.264 will allow the video to be displayed in Streamlit. See this StackOverflow post or this Streamlit forum post for more information.

18.6 Display interactive widgets

With widgets, Streamlit allows you to bake interactivity directly into your apps with buttons, sliders, text inputs, and more.

streamlit.**button**(*label*, *key=None*)

Display a button widget.

Parameters

- **label** (*str*) – A short label explaining to the user what this button is for.
- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns If the button was clicked on the last run of the app.

Return type bool

Example

```
>>> if st.button('Say hello'):
...     st.write('Why hello there')
... else:
...     st.write('Goodbye')
```

streamlit.**checkbox**(*label*, *value=False*, *key=None*)

Display a checkbox widget.

Parameters

- **label** (*str*) – A short label explaining to the user what this checkbox is for.
- **value** (*bool*) – Preselect the checkbox when it first renders. This will be cast to bool internally.
- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns Whether or not the checkbox is checked.

Return type bool

Example

```
>>> agree = st.checkbox('I agree')
>>>
>>> if agree:
...     st.write('Great!')
```

streamlit.**radio**(*label*, *options*, *index=0*, *format_func=<class 'str'>*, *key=None*)

Display a radio button widget.

Parameters

- **label** (*str*) – A short label explaining to the user what this radio group is for.
- **options** (*list, tuple, numpy.ndarray, pandas.Series, or pandas.DataFrame*) – Labels for the radio options. This will be cast to str internally by default. For pandas.DataFrame, the first column is selected.
- **index** (*int*) – The index of the preselected option on first render.
- **format_func** (*function*) – Function to modify the display of radio options. It receives the raw option as an argument and should output the label to be shown for that option. This has no impact on the return value of the radio.
- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns The selected option.

Return type any

Example

```
>>> genre = st.radio(
...     "What's your favorite movie genre",
...     ('Comedy', 'Drama', 'Documentary'))
>>>
>>> if genre == 'Comedy':
...     st.write('You selected comedy.')
... else:
...     st.write("You didn't select comedy.")
```

streamlit.**selectbox**(*label*, *options*, *index=0*, *format_func=<class 'str'>*, *key=None*)

Display a select widget.

Parameters

- **label** (*str*) – A short label explaining to the user what this select widget is for.
- **options** (*list, tuple, numpy.ndarray, pandas.Series, or pandas.DataFrame*) – Labels for the select options. This will be cast to str internally by default. For pandas.DataFrame, the first column is selected.
- **index** (*int*) – The index of the preselected option on first render.
- **format_func** (*function*) – Function to modify the display of the labels. It receives the option as an argument and its output will be cast to str.
- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns The selected option

Return type any

Example

```
>>> option = st.selectbox(
...     'How would you like to be contacted?',
...     ('Email', 'Home phone', 'Mobile phone'))
>>>
>>> st.write('You selected:', option)
```

streamlit.**multiselect**(*label*, *options*, *default=None*, *format_func=<class 'str'>*, *key=None*)

Display a multiselect widget. The multiselect widget starts as empty.

Parameters

- **label** (*str*) – A short label explaining to the user what this select widget is for.
- **options** (*list, tuple, numpy.ndarray, pandas.Series, or pandas.DataFrame*) – Labels for the select options. This will be cast to str internally by default. For pandas.DataFrame, the first column is selected.
- **default** (*[str] or None*) – List of default values.
- **format_func** (*function*) – Function to modify the display of selectbox options. It receives the raw option as an argument and should output the label to be shown for that option. This has no impact on the return value of the selectbox.

- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns A list with the selected options

Return type [str]

Example

```
>>> options = st.multiselect(
...     'What are your favorite colors',
...     ['Green', 'Yellow', 'Red', 'Blue'],
...     ['Yellow', 'Red'])
>>>
>>> st.write('You selected:', options)
```

Note: User experience can be degraded for large lists of *options* (100+), as this widget is not designed to handle arbitrary text search efficiently. See this thread on the Streamlit community forum for more information and GitHub issue #1059 for updates on the issue.

streamlit.**slider**(*label*, *min_value=None*, *max_value=None*, *value=None*, *step=None*, *format=None*, *key=None*)

Display a slider widget.

This supports int, float, date, time, and datetime types.

This also allows you to render a range slider by passing a two-element tuple or list as the *value*.

The difference between *st.slider* and *st.select_slider* is that *slider* only accepts numerical or date/time data and takes a range as input, while *select_slider* accepts any datatype and takes an iterable set of options.

Parameters

- **label** (*str or None*) – A short label explaining to the user what this slider is for.
- **min_value** (*a supported type or None*) – The minimum permitted value. Defaults to 0 if the value is an int, 0.0 if a float, value - timedelta(days=14) if a date/datetime, time.min if a time
- **max_value** (*a supported type or None*) – The maximum permitted value. Defaults to 100 if the value is an int, 1.0 if a float, value + timedelta(days=14) if a date/datetime, time.max if a time
- **value** (*a supported type or a tuple/list of supported types or None*) – The value of the slider when it first renders. If a tuple/list of two values is passed here, then a range slider with those lower and upper bounds is rendered. For example, if set to *(1, 10)* the slider will have a selectable range between 1 and 10. Defaults to min_value.
- **step** (*int/float/timedelta or None*) – The stepping interval. Defaults to 1 if the value is an int, 0.01 if a float, timedelta(days=1) if a date/datetime, timedelta(minutes=15) if a time (or if max_value - min_value < 1 day)
- **format** (*str or None*) – A printf-style format string controlling how the interface should display numbers. This does not impact the return value. Formatter for int/float supports: %d %e %f %g %i Formatter for date/time/datetime uses Moment.js notation: https://momentjs.com/docs/#/displaying/format/

- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns The current value of the slider widget. The return type will match the data type of the value parameter.

Return type int/float/date/time/datetime or tuple of int/float/date/time/datetime

Examples

```
>>> age = st.slider('How old are you?', 0, 130, 25)
>>> st.write("I'm ", age, 'years old')
```

And here's an example of a range slider:

```
>>> values = st.slider(
...     'Select a range of values',
...     0.0, 100.0, (25.0, 75.0))
>>> st.write('Values:', values)
```

This is a range time slider:

```
>>> from datetime import time
>>> appointment = st.slider(
...     "Schedule your appointment:",
...     value=(time(11, 30), time(12, 45)))
>>> st.write("You're scheduled for:", appointment)
```

Finally, a datetime slider:

```
>>> from datetime import datetime
>>> start_time = st.slider(
...     "When do you start?",
...     value=datetime(2020, 1, 1, 9, 30),
...     format="MM/DD/YY - hh:mm")
>>> st.write("Start time:", start_time)
```

streamlit.**select_slider**(*label*, *options=[]*, *value=None*, *format_func=<class 'str'>*, *key=None*)

Display a slider widget to select items from a list.

This also allows you to render a range slider by passing a two-element tuple or list as the *value*.

The difference between *st.select_slider* and *st.slider* is that *select_slider* accepts any datatype and takes an iterable set of options, while *slider* only accepts numerical or date/time data and takes a range as input.

Parameters

- **label** (*str or None*) – A short label explaining to the user what this slider is for.
- **options** (*list, tuple, numpy.ndarray, pandas.Series, or pandas.DataFrame*) – Labels for the slider options. All options will be cast to str internally by default. For pandas.DataFrame, the first column is selected.
- **value** (*a supported type or a tuple/list of supported types or None*) – The value of the slider when it first renders. If a tuple/list of two values is passed here, then a range slider with those lower and upper bounds is rendered. For example, if set to *(1, 10)* the slider will have a selectable range between 1 and 10. Defaults to first option.

- **format_func** (*function*) – Function to modify the display of the labels from the options. argument. It receives the option as an argument and its output will be cast to str.
- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns The current value of the slider widget. The return type will match the data type of the value parameter.

Return type any value or tuple of any value

Examples

```
>>> color = st.select_slider(
...     'Select a color of the rainbow',
...     options=['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'])
>>> st.write('My favorite color is', color)
```

And here's an example of a range select slider:

```
>>> start_color, end_color = st.select_slider(
...     'Select a range of color wavelength',
...     options=['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet'],
...     value=('red', 'blue'))
>>> st.write('You selected wavelengths between', start_color, 'and', end_color)
```

streamlit.**text_input**(*label*, *value=''*, *max_chars=None*, *key=None*, *type='default'*)

Display a single-line text input widget.

Parameters

- **label** (*str*) – A short label explaining to the user what this input is for.
- **value** (*any*) – The text value of this widget when it first renders. This will be cast to str internally.
- **max_chars** (*int or None*) – Max number of characters allowed in text input.
- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.
- **type** (*str*) – The type of the text input. This can be either "default" (for a regular text input), or "password" (for a text input that masks the user's typed value). Defaults to "default".

Returns The current value of the text input widget.

Return type str

Example

```
>>> title = st.text_input('Movie title', 'Life of Brian')
>>> st.write('The current movie title is', title)
```

streamlit.**number_input**(*label*, *min_value=None*, *max_value=None*, *value=<streamlit.elements.utils.NoValue object>*, *step=None*, *format=None*, *key=None*)

Display a numeric input widget.

Parameters

- **label** (*str or None*) – A short label explaining to the user what this input is for.
- **min_value** (*int or float or None*) – The minimum permitted value. If None, there will be no minimum.
- **max_value** (*int or float or None*) – The maximum permitted value. If None, there will be no maximum.
- **value** (*int or float or None*) – The value of this widget when it first renders. Defaults to min_value, or 0.0 if min_value is None
- **step** (*int or float or None*) – The stepping interval. Defaults to 1 if the value is an int, 0.01 otherwise. If the value is not specified, the format parameter will be used.
- **format** (*str or None*) – A printf-style format string controlling how the interface should display numbers. Output must be purely numeric. This does not impact the return value. Valid formatters: %d %e %f %g %i
- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns The current value of the numeric input widget. The return type will match the data type of the value parameter.

Return type int or float

Example

```
>>> number = st.number_input('Insert a number')
>>> st.write('The current number is ', number)
```

streamlit.**text_area**(*label*, *value=''*, *height=None*, *max_chars=None*, *key=None*)

Display a multi-line text input widget.

Parameters

- **label** (*str*) – A short label explaining to the user what this input is for.
- **value** (*any*) – The text value of this widget when it first renders. This will be cast to str internally.
- **height** (*int or None*) – Desired height of the UI element expressed in pixels. If None, a default height is used.
- **max_chars** (*int or None*) – Maximum number of characters allowed in text area.

- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns The current value of the text input widget.

Return type str

Example

```
>>> txt = st.text_area('Text to analyze', '''
...     It was the best of times, it was the worst of times, it was
...     the age of wisdom, it was the age of foolishness, it was
...     the epoch of belief, it was the epoch of incredulity, it
...     was the season of Light, it was the season of Darkness, it
...     was the spring of hope, it was the winter of despair, (...)
...     ''')
>>> st.write('Sentiment:', run_sentiment_analysis(txt))
```

streamlit.**date_input**(*label*, *value=None*, *min_value=None*, *max_value=None*, *key=None*)

Display a date input widget.

Parameters

- **label** (*str*) – A short label explaining to the user what this date input is for.
- **value** (*datetime.date or datetime.datetime or list/tuple of datetime.date or datetime.datetime or None*) – The value of this widget when it first renders. If a list/tuple with 0 to 2 date/datetime values is provided, the datepicker will allow users to provide a range. Defaults to today as a single-date picker.
- **min_value** (*datetime.date or datetime.datetime*) – The minimum selectable date. Defaults to today-10y.
- **max_value** (*datetime.date or datetime.datetime*) – The maximum selectable date. Defaults to today+10y.
- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns The current value of the date input widget.

Return type datetime.date

Example

```
>>> d = st.date_input(
...     "When's your birthday",
...     datetime.date(2019, 7, 6))
>>> st.write('Your birthday is:', d)
```

streamlit.**time_input**(*label*, *value=None*, *key=None*)

Display a time input widget.

Parameters

- **label** (*str*) – A short label explaining to the user what this time input is for.

- **value** (*datetime.time/datetime.datetime*) – The value of this widget when it first renders. This will be cast to str internally. Defaults to the current time.
- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns The current value of the time input widget.

Return type datetime.time

Example

```
>>> t = st.time_input('Set an alarm for', datetime.time(8, 45))
>>> st.write('Alarm is set for', t)
```

streamlit.**file_uploader**(*label*, *type=None*, *accept_multiple_files=False*, *key=None*, ***kwargs*)

Display a file uploader widget. By default, uploaded files are limited to 200MB. You can configure this using the *server.maxUploadSize* config option.

Parameters

- **label** (*str or None*) – A short label explaining to the user what this file uploader is for.
- **type** (*str or list of str or None*) – Array of allowed extensions. ['png', 'jpg'] The default is None, which means all extensions are allowed.
- **accept_multiple_files** (*bool*) – If True, allows the user to upload multiple files at the same time, in which case the return value will be a list of files. Default: False
- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns

- If allow_multiple_files is False, returns either None or an UploadedFile object.
- If allow_multiple_files is True, returns a list with the uploaded files as UploadedFile objects. If no files were uploaded, returns an empty list.

The UploadedFile class is a subclass of BytesIO, and therefore it is "file-like". This means you can pass them anywhere where a file is expected.

Return type None or UploadedFile or list of UploadedFile

Examples

Insert a file uploader that accepts a single file at a time:

```
>>> uploaded_file = st.file_uploader("Choose a file")
>>> if uploaded_file is not None:
...     # To read file as bytes:
...     bytes_data = uploaded_file.read()
...     st.write(bytes_data)
>>>
...     # To convert to a string based IO:
...     stringio = StringIO(uploaded_file.decode("utf-8"))
```

(continues on next page)

(continued from previous page)

```
...     st.write(stringio)
>>>
...     # To read file as string:
...     string_data = stringio.read()
...     st.write(string_data)
>>>
...     # Can be used wherever a "file-like" object is accepted:
...     dataframe = pd.read_csv(uploaded_file)
...     st.write(dataframe)
```

Insert a file uploader that accepts multiple files at a time:

```
>>> uploaded_files = st.file_uploader("Choose a CSV file", accept_multiple_
↪files=True)
>>> for uploaded_file in uploaded_files:
...     bytes_data = uploaded_file.read()
...     st.write("filename:", uploaded_file.name)
...     st.write(bytes_data)
```

streamlit.**color_picker**(*label*, *value=None*, *key=None*)

Display a color picker widget.

Note: This is a beta feature. See https://docs.streamlit.io/en/latest/api.html#pre-release-features for more information.

Parameters

- **label** (*str*) – A short label explaining to the user what this input is for.
- **value** (*str or None*) – The hex value of this widget when it first renders. If None, defaults to black.
- **key** (*str*) – An optional string to use as the unique key for the widget. If this is omitted, a key will be generated for the widget based on its content. Multiple widgets of the same type may not share the same key.

Returns The selected color as a hex string.

Return type str

Example

```
>>> color = st.color_picker('Pick A Color', '#00f900')
>>> st.write('The current color is', color)
```

18.7 Control flow

By default, Streamlit apps execute the script entirely, but we allow some functionality to handle control flow in your applications.

streamlit.**stop**()

Stops execution immediately.

Streamlit will not run any statements after *st.stop()*. We recommend rendering a message to explain why the script has stopped. When run outside of Streamlit, this will raise an Exception.

Example

```
>>> name = st.text_input('Name')
>>> if not name:
>>>   st.warning('Please input a name.')
>>>   st.stop()
>>> st.success('Thank you for inputting a name.')
```

18.8 Add widgets to sidebar

Not only can you add interactivity to your report with widgets, you can organize them into a sidebar with `st.sidebar.[element_name]`. Each element that's passed to `st.sidebar` is pinned to the left, allowing users to focus on the content in your app. The only elements that aren't supported are `st.echo` and `st.spinner`.

Here's an example of how you'd add a selectbox to your sidebar.

```
import streamlit as st

add_selectbox = st.sidebar.selectbox(
    "How would you like to be contacted?",
    ("Email", "Home phone", "Mobile phone")
)
```

18.9 Lay out your app

In addition to the sidebar, you have a few other options for controlling how your app is laid out.

Note: These are beta features. See https://docs.streamlit.io/en/latest/api.html#pre-release-features for more information.

`streamlit.`**`beta_container`**`()`

Insert a multi-element container.

Inserts an invisible container into your app that can be used to hold multiple elements. This allows you to, for example, insert multiple elements into your app out of order.

To add elements to the returned container, you can use "with" notation (preferred) or just call methods directly on the returned object. See examples below.

Examples

Inserting elements using "with" notation:

```
>>> with st.beta_container():
...     st.write("This is inside the container")
...
...     # You can call any Streamlit command, including custom components:
...     st.bar_chart(np.random.randn(50, 3))
...
>>> st.write("This is outside the container")
```

Inserting elements out of order:

```
>>> container = st.beta_container()
>>> container.write("This is inside the container")
>>> st.write("This is outside the container")
>>>
>>> # Now insert some more in the container
>>> container.write("This is inside too")
```

streamlit.**beta_columns**(*spec*)

Insert containers laid out as side-by-side columns.

Inserts a number of multi-element containers laid out side-by-side and returns a list of container objects.

To add elements to the returned containers, you can use "with" notation (preferred) or just call methods directly on the returned object. See examples below.

Warning: Currently, you may not put columns inside another column.

Parameters **spec** (*int or list of numbers*) –

If an int Specifies the number of columns to insert, and all columns have equal width.

If a list of numbers Creates a column for each number, and each column's width is proportional to the number provided. Numbers can be ints or floats, but they must be positive.

For example, *st.beta_columns([3, 1, 2])* creates 3 columns where the first column is 3 times the width of the second, and the last column is 2 times that width.

Returns A list of container objects.

Return type list of containers

Examples

You can use *with* notation to insert any element into a column:

```
>>> col1, col2, col3 = st.beta_columns(3)
>>>
>>> with col1:
...     st.header("A cat")
...     st.image("https://static.streamlit.io/examples/cat.jpg", use_column_
↪width=True)
...
>>> with col2:
...     st.header("A dog")
...     st.image("https://static.streamlit.io/examples/dog.jpg", use_column_
↪width=True)
...
>>> with col3:
...     st.header("An owl")
...     st.image("https://static.streamlit.io/examples/owl.jpg", use_column_
↪width=True)
```

Or you can just call methods directly in the returned objects:

```
>>> col1, col2 = st.beta_columns([3, 1])
>>> data = np.random.randn(10, 1)
>>>
>>> col1.subheader("A wide column with a chart")
>>> col1.line_chart(data)
>>>
>>> col2.subheader("A narrow column with the data")
>>> col2.write(data)
```

streamlit.**beta_expander**(*label=None*, *expanded=False*)

Insert a multi-element container that can be expanded/collapsed.

Inserts a container into your app that can be used to hold multiple elements and can be expanded or collapsed by the user. When collapsed, all that is visible is the provided label.

To add elements to the returned container, you can use "with" notation (preferred) or just call methods directly on the returned object. See examples below.

Warning: Currently, you may not put expanders inside another expander.

Parameters

- **label** (*str*) – A string to use as the header for the expander.
- **expanded** (*bool*) – If True, initializes the expander in "expanded" state. Defaults to False (collapsed).

Examples

```
>>> st.line_chart({"data": [1, 5, 2, 6, 2, 1]})
>>>
>>> with st.beta_expander("See explanation"):
...     st.write("""
...         The chart above shows some numbers I picked for you.
...         I rolled actual dice for these, so they're *guaranteed* to
...         be random.
...     """)
...     st.image("https://static.streamlit.io/examples/dice.jpg")
```

18.10 Display code

Sometimes you want your Streamlit app to contain *both* your usual Streamlit graphic elements *and* the code that generated those elements. That's where `st.echo()` comes in.

streamlit.**echo**(*code_location='above'*)

Use in a *with* block to draw some code on the app, then execute it.

Parameters **code_location** (*"above" or "below"*) – Whether to show the echoed code before or after the results of the executed code block.

Example

```
>>> with st.echo():
>>>     st.write('This code will be printed')
```

Ok so let's say you have the following file, and you want to make its app a little bit more self-explanatory by making that middle section visible in the Streamlit app:

```
import streamlit as st

def get_user_name():
    return 'John'

# ------------------------------------------------
# Want people to see this part of the code...

def get_punctuation():
    return '!!!'

greeting = "Hi there, "
user_name = get_user_name()
punctuation = get_punctuation()

st.write(greeting, user_name, punctuation)

# ...up to here
# ------------------------------------------------

foo = 'bar'
st.write('Done!')
```

The file above creates a Streamlit app containing the words "Hi there, `John`", and then "Done!".

Now let's use `st.echo()` to make that middle section of the code visible in the app:

```
import streamlit as st

def get_user_name():
    return 'John'

with st.echo():
    # Everything inside this block will be both printed to the screen
    # and executed.

    def get_punctuation():
        return '!!!'

    greeting = "Hi there, "
    value = get_user_name()
    punctuation = get_punctuation()

    st.write(greeting, value, punctuation)

# And now we're back to _not_ printing to the screen
foo = 'bar'
st.write('Done!')
```

It's *that* simple!

Note: You can have multiple *st.echo()* blocks in the same file. Use it as often as you wish!

18.11 Display progress and status

Streamlit provides a few methods that allow you to add animation to your apps. These animations include progress bars, status messages (like warnings), and celebratory balloons.

`streamlit.`**`progress`**(*value*)

Display a progress bar.

Parameters **`value`** (*`int or float`*) – 0 <= value <= 100 for int

0.0 <= value <= 1.0 for float

Example

Here is an example of a progress bar increasing over time:

```
>>> import time
>>>
>>> my_bar = st.progress(0)
>>>
>>> for percent_complete in range(100):
...     time.sleep(0.1)
...     my_bar.progress(percent_complete + 1)
```

`streamlit.`**`spinner`**(*text='In progress...'*)

Temporarily displays a message while executing a block of code.

Parameters **`text`** (*`str`*) – A message to display while executing that block

Example

```
>>> with st.spinner('Wait for it...'):
>>>     time.sleep(5)
>>> st.success('Done!')
```

`streamlit.`**`balloons`**()

Draw celebratory balloons.

Example

```
>>> st.balloons()
```

. . . then watch your app and get ready for a celebration!

`streamlit.`**`error`**(*body*)

Display error message.

Parameters **`body`** (*`str`*) – The error text to display.

Example

```
>>> st.error('This is an error')
```

streamlit.**warning**(*body*)
Display warning message.

Parameters **body** (*str*) – The warning text to display.

Example

```
>>> st.warning('This is a warning')
```

streamlit.**info**(*body*)
Display an informational message.

Parameters **body** (*str*) – The info text to display.

Example

```
>>> st.info('This is a purely informational message')
```

streamlit.**success**(*body*)
Display a success message.

Parameters **body** (*str*) – The success text to display.

Example

```
>>> st.success('This is a success message!')
```

streamlit.**exception**(*exception*)
Display an exception.

Parameters **exception** (*Exception*) – The exception to display.

Example

```
>>> e = RuntimeError('This is an exception of type RuntimeError')
>>> st.exception(e)
```

18.12 Placeholders, help, and options

There are a handful of methods that allow you to create placeholders in your app, provide help using doc strings, and get and modify configuration options.

streamlit.**empty**()
Insert a single-element container.

Inserts a container into your app that can be used to hold a single element. This allows you to, for example, remove elements at any point, or replace several elements at once (using a child multi-element container).

To insert/replace/clear an element on the returned container, you can use "with" notation or just call methods directly on the returned object. See examples below.

Examples

Overwriting elements in-place using "with" notation:

```
>>> import time
>>>
>>> with st.empty():
...     for seconds in range(60):
...         st.write(f" {seconds} seconds have passed")
...         time.sleep(1)
...     st.write("✓ 1 minute over!")
```

Replacing several elements, then clearing them:

```
>>> placeholder = st.empty()
>>>
>>> # Replace the placeholder with some text:
>>> placeholder.text("Hello")
>>>
>>> # Replace the text with a chart:
>>> placeholder.line_chart({"data": [1, 5, 2, 6]})
>>>
>>> # Replace the chart with several elements:
>>> with placeholder.beta_container():
...     st.write("This is one element")
...     st.write("This is another")
...
>>> # Clear all those elements:
>>> placeholder.empty()
```

streamlit.**help**(*obj*)

Display object's doc string, nicely formatted.

Displays the doc string for this object.

Parameters **obj** (*Object*) – The object whose docstring should be displayed.

Example

Don't remember how to initialize a dataframe? Try this:

```
>>> st.help(pandas.DataFrame)
```

Want to quickly check what datatype is output by a certain function? Try:

```
>>> x = my_poorly_documented_function()
>>> st.help(x)
```

streamlit.**get_option**(*key*)

Return the current value of a given Streamlit config option.

Run *streamlit config show* in the terminal to see all available options.

Parameters **key** (*str*) – The config option key of the form "section.optionName". To see all available options, run *streamlit config show* on a terminal.

streamlit.**set_option**(*key*, *value*)

Set config option.

Currently, only the following config options can be set within the script itself:

- client.caching
- client.displayEnabled
- deprecation.*

Calling with any other options will raise StreamlitAPIException.

Run *streamlit config show* in the terminal to see all available options.

Parameters

- **key** (*str*) – The config option key of the form "section.optionName". To see all available options, run *streamlit config show* on a terminal.
- **value** – The new value to assign to this config option.

streamlit.**set_page_config**(*page_title=None*, *page_icon=None*, *layout='centered'*, *initial_sidebar_state='auto'*)

Configures the default settings of the page.

Note: This must be the first Streamlit command used in your app, and must only be set once.

Parameters

- **page_title** (*str or None*) – The page title, shown in the browser tab. If None, defaults to the filename of the script ("app.py" would show "app • Streamlit").
- **page_icon** (*Anything supported by st.image or str or None*) – The page favicon. Besides the types supported by *st.image* (like URLs or numpy arrays), you can pass in an emoji as a string ("") or a shortcode (":shark:"). Emoji icons are courtesy of Twemoji and loaded from MaxCDN.
- **layout** (*"centered" or "wide"*) – How the page content should be laid out. Defaults to "centered", which constrains the elements into a centered column of fixed width; "wide" uses the entire screen.
- **initial_sidebar_state** (*"auto" or "expanded" or "collapsed"*) – How the sidebar should start out. Defaults to "auto", which hides the sidebar on mobile-sized devices, and shows it otherwise. "expanded" shows the sidebar initially; "collapsed" hides it.

Example

```
>>> st.set_page_config(
...     page_title="Ex-stream-ly Cool App",
...     page_icon="",
...     layout="wide",
...     initial_sidebar_state="expanded",
... )
```

18.13 Mutate data

With Streamlit you can modify the data within an existing element (chart, table, dataframe).

DeltaGenerator.**add_rows**(*data=None*, ***kwargs*)

Concatenate a dataframe to the bottom of the current one.

Parameters

- **data** (*pandas.DataFrame, pandas.Styler, numpy.ndarray, Iterable, dict,*) –
- **None** (*or*) – Table to concat. Optional.
- ****kwargs** (*pandas.DataFrame, numpy.ndarray, Iterable, dict, or None*) – The named dataset to concat. Optional. You can only pass in 1 dataset (including the one in the data parameter).

Example

```
>>> df1 = pd.DataFrame(
...     np.random.randn(50, 20),
...     columns=('col %d' % i for i in range(20)))
...
>>> my_table = st.table(df1)
>>>
>>> df2 = pd.DataFrame(
...     np.random.randn(50, 20),
...     columns=('col %d' % i for i in range(20)))
...
>>> my_table.add_rows(df2)
>>> # Now the table shown in the Streamlit app contains the data for
>>> # df1 followed by the data for df2.
```

You can do the same thing with plots. For example, if you want to add more data to a line chart:

```
>>> # Assuming df1 and df2 from the example above still exist...
>>> my_chart = st.line_chart(df1)
>>> my_chart.add_rows(df2)
>>> # Now the chart shown in the Streamlit app contains the data for
>>> # df1 followed by the data for df2.
```

And for plots whose datasets are named, you can pass the data with a keyword argument where the key is the name:

```
>>> my_chart = st.vega_lite_chart({
...     'mark': 'line',
...     'encoding': {'x': 'a', 'y': 'b'},
...     'datasets': {
...       'some_fancy_name': df1,  # <-- named dataset
...      },
...     'data': {'name': 'some_fancy_name'},
... }),
>>> my_chart.add_rows(some_fancy_name=df2)  # <-- name used as keyword
```

18.14 Optimize performance

When you mark a function with Streamlit's cache annotation, it tells Streamlit that whenever the function is called it should check three things:

1. The name of the function
2. The actual code that makes up the body of the function
3. The input parameters that you called the function with

If this is the first time Streamlit has seen those three items, with those exact values, and in that exact combination, it runs the function and stores the result in a local cache.

Then, next time the function is called, if those three values have not changed Streamlit knows it can skip executing the function altogether. Instead, it just reads the output from the local cache and passes it on to the caller.

The main limitation is that Streamlit's cache feature doesn't know about changes that take place outside the body of the annotated function.

For more information about the Streamlit cache, its configuration parameters, and its limitations, see *Caching*.

`streamlit.`**`cache`**(*func=None*, *persist=False*, *allow_output_mutation=False*, *show_spinner=True*, *suppress_st_warning=False*, *hash_funcs=None*, *max_entries=None*, *ttl=None*)

Function decorator to memoize function executions.

Parameters

- **`func`** (*`callable`*) – The function to cache. Streamlit hashes the function and dependent code.
- **`persist`** (*`boolean`*) – Whether to persist the cache on disk.
- **`allow_output_mutation`** (*`boolean`*) – Streamlit normally shows a warning when return values are not mutated, as that can have unintended consequences. This is done by hashing the return value internally.

  If you know what you're doing and would like to override this warning, set this to True.
- **`show_spinner`** (*`boolean`*) – Enable the spinner. Default is True to show a spinner when there is a cache miss.
- **`suppress_st_warning`** (*`boolean`*) – Suppress warnings about calling Streamlit functions from within the cached function.
- **`hash_funcs`** (*`dict or None`*) – Mapping of types or fully qualified names to hash functions. This is used to override the behavior of the hasher inside Streamlit's caching mechanism: when the hasher encounters an object, it will first check to see if its type matches a key in this dict and, if so, will use the provided function to generate a hash for it. See below for an example of how this can be used.
- **`max_entries`** (*`int or None`*) – The maximum number of entries to keep in the cache, or None for an unbounded cache. (When a new entry is added to a full cache, the oldest cached entry will be removed.) The default is None.
- **`ttl`** (*`float or None`*) – The maximum number of seconds to keep an entry in the cache, or None if cache entries should not expire. The default is None.

Example

```
>>> @st.cache
... def fetch_and_clean_data(url):
...     # Fetch data from URL here, and then clean it up.
...     return data
...
>>> d1 = fetch_and_clean_data(DATA_URL_1)
>>> # Actually executes the function, since this is the first time it was
>>> # encountered.
>>>
>>> d2 = fetch_and_clean_data(DATA_URL_1)
>>> # Does not execute the function. Instead, returns its previously computed
>>> # value. This means that now the data in d1 is the same as in d2.
>>>
>>> d3 = fetch_and_clean_data(DATA_URL_2)
>>> # This is a different URL, so the function executes.
```

To set the *persist* parameter, use this command as follows:

```
>>> @st.cache(persist=True)
... def fetch_and_clean_data(url):
...     # Fetch data from URL here, and then clean it up.
...     return data
```

To disable hashing return values, set the *allow_output_mutation* parameter to *True*:

```
>>> @st.cache(allow_output_mutation=True)
... def fetch_and_clean_data(url):
...     # Fetch data from URL here, and then clean it up.
...     return data
```

To override the default hashing behavior, pass a custom hash function. You can do that by mapping a type (e.g. *MongoClient*) to a hash function (*id*) like this:

```
>>> @st.cache(hash_funcs={MongoClient: id})
... def connect_to_database(url):
...     return MongoClient(url)
```

Alternatively, you can map the type's fully-qualified name (e.g. *"pymongo.mongo_client.MongoClient"*) to the hash function instead:

```
>>> @st.cache(hash_funcs={"pymongo.mongo_client.MongoClient": id})
... def connect_to_database(url):
...     return MongoClient(url)
```

18.15 Pre-release features

At Streamlit, we like to move quick while keeping things stable. In our latest effort to move even faster without sacrificing stability, we're offering our bold and fearless users two ways to try out Streamlit's bleeding-edge features:

1. *Nightly releases*
2. *Beta and experimental features*

18.15.1 Nightly releases

At the end of each day (at night), our bots run automated tests against the latest Streamlit code and, if everything looks good, it publishes them as the `streamlit-nightly` package. This means the nightly build includes all our latest features, bug fixes, and other enhancements on the same day they land on our codebase.

How does this differ from official releases?

Official Streamlit releases go not only through both automated tests but also rigorous manual testing, while nightly releases only have automated tests. It's important to keep in mind that new features introduced in nightly releases often lack polish. In our official releases, we always make double-sure all new features are ready for prime time.

How do I use the nightly release?

All you need to do is install the `streamlit-nightly` package:

```
pip uninstall streamlit
pip install streamlit-nightly --upgrade
```

Warning: You should never have both *streamlit* and *streamlit-nightly* installed in the same environment!

Why should I use the nightly release?

Because you can't wait for official releases, and you want to help us find bugs early!

Why shouldn't I use the nightly release?

While our automated tests have high coverage, there's still a significant likelihood that there will be some bugs in the nightly code.

Can I choose which nightly release I want to install?

If you'd like to use a specific version, you can find the version number in our Release history. Specify the desired version using `pip` as usual: `pip install streamlit-nightly==x.yy.zz-123456`.

Can I compare changes between releases?

If you'd like to review the changes for a nightly release, you can use the comparison tool on GitHub.

18.15.2 Beta and Experimental Features

In addition to nightly releases, we also have two naming conventions for less stable Streamlit features: `st.beta_` and `st.experimental_`. These distinctions are prefixes we attach to our function names to make sure their status is clear to everyone.

Here's a quick rundown of what you get from each naming convention:

- **st**: this is where our core features like `st.write` and `st.dataframe` live. If we ever make backward-incompatible changes to these, they will take place gradually and with months of announcements and warnings.
- **beta_**: this is where all new features land before they becoming part of Streamlit core. This gives you a chance to try the next big thing we're cooking up weeks or months before we're ready to stabilize its API.
- **experimental_**: this is where we'll put features that may or may not ever make it into Streamlit core. We don't know whether these features have a future, but we want you to have access to everything we're trying, and work with us to figure them out.

The main difference between `beta_` and `experimental_` is that beta features are expected to make it into Streamlit core at some point soon, while experimental features may never make it.

Beta

Features with the `beta_` naming convention are all scheduled to become part of Streamlit core. While in beta, a feature's API and behaviors may not be stable, and it's possible they could change in ways that aren't backward-compatible.

The lifecycle of a beta feature

1. A feature is added with the `beta_` prefix.
2. The feature's API stabilizes and the feature is *cloned* without the `beta_` prefix, so it exists as both `st` and `beta_`. At this point, users will see a warning when using the version of the feature with the `beta_` prefix – but the feature will still work.
3. At some point, the code of the `beta_`-prefixed feature is *removed*, but there will still be a stub of the function prefixed with `beta_` that shows an error with appropriate instructions.
4. Finally, at a later date the `beta_` version is removed.

Experimental

Features with the `experimental_` naming convention are things that we're still working on or trying to understand. If these features are successful, at some point they'll become part of Streamlit core, by moving to the `beta_` naming convention and then to Streamlit core. If unsuccessful, these features are removed without much notice.

> **Warning:** Experimental features and their APIs may change or be removed at any time.

The lifecycle of an experimental feature

1. A feature is added with the `experimental_` prefix.
2. The feature is potentially tweaked over time, with possible API/behavior breakages.
3. At some point, we either promote the feature to `beta_` or remove it from `experimental_`. Either way, we leave a stub in `experimental_` that shows an error with instructions.

CHAPTER

NINETEEN

STREAMLIT CONFIGURATION

19.1 Command-line interface

When you install Streamlit, a command-line (CLI) tool gets installed as well. The purpose of this tool is to run Streamlit apps, change Streamlit configuration options, and help you diagnose and fix issues.

To see all of the supported commands:

```
streamlit --help
```

19.1.1 Run Streamlit apps

```
streamlit run your_script.py [-- script args]
```

Runs your app. At any time you can stop the server with **Ctrl+c**.

Note: When passing your script some custom arguments, **they must be passed after two dashes**. Otherwise the arguments get interpreted as arguments to Streamlit itself.

To see the Streamlit 'Hello, World!' example app, run `streamlit hello`.

19.1.2 View documentation

```
streamlit docs
```

Opens the Streamlit documentation (i.e. this website) in a web browser.

19.1.3 Clear cache

```
streamlit cache clear
```

Clears persisted files from the on-disk Streamlit cache, if present.

19.2 Set configuration options

Streamlit provides four different ways to set configuration options:

1. In a **global config file** at `~/.streamlit/config.toml` for macOS/Linux or `%userprofile%/.streamlit/config.toml` for Windows:

```
[server]
port = 80
```

2. In a **per-project config file** at `$CWD/.streamlit/config.toml`, where `$CWD` is the folder you're running Streamlit from.

3. Through `STREAMLIT_*` **environment variables**, such as:

```
export STREAMLIT_SERVER_PORT=80
```

4. As **flags on the command line** when running `streamlit run`. These allow you to do things like change the port the app is served from, disable run-on-save, and more:

```
streamlit run your_script.py --server.port 80
```

19.3 View all configuration options

```
streamlit config show
```

Shows all config options available for Streamlit, including their current values:

```
# last updated 2020-12-03

[global]

# By default, Streamlit checks if the Python watchdog module is available and, if not,
↪ prints a warning asking for you to install it. The watchdog module is not required,
↪ but highly recommended. It improves Streamlit's ability to detect changes to files␣
↪in your filesystem.
# If you'd like to turn off this warning, set this to True.
# Default: false
disableWatchdogWarning = false

# If True, will show a warning when you run a Streamlit-enabled script via "python my_
↪script.py".
# Default: true
showWarningOnDirectExecution = true

# Level of logging: 'error', 'warning', 'info', or 'debug'.
# Default: 'info'
#
# DEPRECATED.
# global.logLevel has been replaced with logger.level
# This option will be removed on or after 2020-11-30.
#
#logLevel =

[logger]
```

(continues on next page)

(continued from previous page)

```
# Level of logging: 'error', 'warning', 'info', or 'debug'.
# Default: 'info'
level = "debug"

# String format for logging messages. If logger.datetimeFormat is set, logger␣
↪messages will default to `%(asctime)s.%(msecs)03d %(message)s`. See [Python's␣
↪documentation](https://docs.python.org/2.6/library/logging.html#formatter-objects)␣
↪for available attributes.
# Default: None
messageFormat = "%(asctime)s %(levelname) -7s %(name)s: %(message)s"

[client]

# Whether to enable st.cache.
# Default: true
caching = true

# If false, makes your Streamlit script not draw to a Streamlit app.
# Default: true
displayEnabled = true

[runner]

# Allows you to type a variable or string by itself in a single line of Python code␣
↪to write it to the app.
# Default: true
magicEnabled = true

# Install a Python tracer to allow you to stop or pause your script at any point and␣
↪introspect it. As a side-effect, this slows down your script's execution.
# Default: false
installTracer = false

# Sets the MPLBACKEND environment variable to Agg inside Streamlit to prevent Python␣
↪crashing.
# Default: true
fixMatplotlib = true

[server]

# List of folders that should not be watched for changes. This impacts both "Run on␣
↪Save" and @st.cache.
# Relative paths will be taken as relative to the current working directory.
# Example: ['/home/user1/env', 'relative/path/to/folder']
# Default: []
folderWatchBlacklist = []

# Change the type of file watcher used by Streamlit, or turn it off completely.
# Allowed values: * "auto" : Streamlit will attempt to use the watchdog module, and␣
↪falls back to polling if watchdog is not available. * "watchdog" : Force Streamlit␣
↪to use the watchdog module. * "poll" : Force Streamlit to always use polling. *
↪"none" : Streamlit will not watch files.
# Default: "auto"
fileWatcherType = "auto"

# Symmetric key used to produce signed cookies. If deploying on multiple replicas,␣
↪this should be set to the same value across all replicas to ensure they all share␣
↪the same secret.
```

(continues on next page)

(continued from previous page)

```
# Default: randomly generated secret key.
cookieSecret = "909a66338a19aad71fc0381e2b126f95ec2db3d19094e3c4ad51f866d3bda991"

# If false, will attempt to open a browser window on start.
# Default: false unless (1) we are on a Linux box where DISPLAY is unset, or (2)␣
↪server.liveSave is set.
headless = false

# Automatically rerun script when the file is modified on disk.
# Default: false
runOnSave = false

# The address where the server will listen for client and browser connections. Use␣
↪this if you want to bind the server to a specific address. If set, the server will␣
↪only be accessible from this address, and not from any aliases (like localhost).
# Default: (unset)
#address =

# The port where the server will listen for browser connections.
# Default: 8501
port = 8501

# The base path for the URL where Streamlit should be served from.
# Default: ""
baseUrlPath = ""

# Enables support for Cross-Origin Request Sharing (CORS) protection, for added␣
↪security.
# Due to conflicts between CORS and XSRF, if `server.enableXsrfProtection` is on and␣
↪`server.enableCORS` is off at the same time, we will prioritize `server.
↪enableXsrfProtection`.
# Default: true
enableCORS = true

# Enables support for Cross-Site Request Forgery (XSRF) protection, for added␣
↪security.
# Due to conflicts between CORS and XSRF, if `server.enableXsrfProtection` is on and␣
↪`server.enableCORS` is off at the same time, we will prioritize `server.
↪enableXsrfProtection`.
# Default: true
enableXsrfProtection = true

# Max size, in megabytes, for files uploaded with the file_uploader.
# Default: 200
maxUploadSize = 200

# Enables support for websocket compression.
# Default: true
enableWebsocketCompression = true

[browser]

# Internet address where users should point their browsers in order to connect to the␣
↪app. Can be IP address or DNS name and path.
# This is used to: - Set the correct URL for CORS and XSRF protection purposes. -␣
↪Show the URL on the terminal - Open the browser - Tell the browser where to connect␣
↪to the server when in liveSave mode.
```

(continues on next page)

(continued from previous page)

```
# Default: 'localhost'
serverAddress = "localhost"

# Whether to send usage statistics to Streamlit.
# Default: true
gatherUsageStats = true

# Port where users should point their browsers in order to connect to the app.
# This is used to: - Set the correct URL for CORS and XSRF protection purposes. -␣
↪Show the URL on the terminal - Open the browser - Tell the browser where to connect␣
↪to the server when in liveSave mode.
# Default: whatever value is set in server.port.
serverPort = 8501

[mapbox]

# Configure Streamlit to use a custom Mapbox token for elements like st.pydeck_chart␣
↪and st.map. To get a token for yourself, create an account at https://mapbox.com. It
↪'s free (for moderate usage levels)!
# Default: ""
token = ""

[deprecation]

# Set to false to disable the deprecation warning for the file uploader encoding.
# Default: "True"
showfileUploaderEncoding = "True"

# Set to false to disable the deprecation warning for the image format parameter.
# Default: "True"
showImageFormat = "True"

# Set to false to disable the deprecation warning for using the global pyplot␣
↪instance.
# Default: "True"
showPyplotGlobalUse = "True"

[s3]

# Name of the AWS S3 bucket to save apps.
# Default: (unset)
#bucket =

# URL root for external view of Streamlit apps.
# Default: (unset)
#url =

# Access key to write to the S3 bucket.
# Leave unset if you want to use an AWS profile.
# Default: (unset)
#accessKeyId =

# Secret access key to write to the S3 bucket.
# Leave unset if you want to use an AWS profile.
# Default: (unset)
#secretAccessKey =
```

(continues on next page)

(continued from previous page)

```
# The "subdirectory" within the S3 bucket where to save apps.
# S3 calls paths "keys" which is why the keyPrefix is like a subdirectory. Use "" to␣
↪mean the root directory.
# Default: ""
keyPrefix = ""

# AWS region where the bucket is located, e.g. "us-west-2".
# Default: (unset)
#region =

# AWS credentials profile to use.
# Leave unset to use your default profile.
# Default: (unset)
#profile =
```

CHAPTER

TWENTY

COMPONENTS API REFERENCE

The first step in developing a Streamlit Component is deciding whether to create a static component (i.e. rendered once, controlled by Python) or to create a bi-directional component that can communicate from Python to JavaScript and back.

20.1 Create a static component

If your goal in creating a Streamlit Component is solely to display HTML code or render a chart from a Python visualization library, Streamlit provides two methods that greatly simplify the process: `components.html()` and `components.iframe()`.

If you are unsure whether you need bi-directional communication, **start here first**!

20.1.1 Render an HTML string

While `st.text`, `st.markdown` and `st.write` make it easy to write text to a Streamlit app, sometimes you'd rather implement a custom piece of HTML. Similarly, while Streamlit natively supports many charting libraries, you may want to implement a specific HTML/JavaScript template for a new charting library. `components.html` works by giving you the ability to embed an iframe inside of a Streamlit app that contains your desired output.

`streamlit.components.v1.`**`html`**(*html*, *width=None*, *height=None*, *scrolling=False*)

Display an HTML string in an iframe.

Parameters

- **`html`** (*`str`*) – The HTML string to embed in the iframe.
- **`width`** (*`int`*) – The width of the frame in CSS pixels. Defaults to the report's default element width.
- **`height`** (*`int`*) – The height of the frame in CSS pixels. Defaults to 150.
- **`scrolling`** (*`bool`*) – If True, show a scrollbar when the content is larger than the iframe. Otherwise, do not show a scrollbar. Defaults to False.

Example

```
import streamlit as st
import streamlit.components.v1 as components

# bootstrap 4 collapse example
components.html(
    """
```

(continues on next page)

(continued from previous page)

```
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/
↪bootstrap.min.css" integrity="sha384-
↪Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin=
↪"anonymous">
    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="sha384-
↪KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN" crossorigin=
↪"anonymous"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"␣
↪integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmYl"␣
↪crossorigin="anonymous"></script>
    <div id="accordion">
      <div class="card">
        <div class="card-header" id="headingOne">
          <h5 class="mb-0">
            <button class="btn btn-link" data-toggle="collapse" data-target="
↪#collapseOne" aria-expanded="true" aria-controls="collapseOne">
            Collapsible Group Item #1
            </button>
          </h5>
        </div>
        <div id="collapseOne" class="collapse show" aria-labelledby="headingOne" data-
↪parent="#accordion">
          <div class="card-body">
            Collapsible Group Item #1 content
          </div>
        </div>
      </div>
      <div class="card">
        <div class="card-header" id="headingTwo">
          <h5 class="mb-0">
            <button class="btn btn-link collapsed" data-toggle="collapse" data-target=
↪"#collapseTwo" aria-expanded="false" aria-controls="collapseTwo">
            Collapsible Group Item #2
            </button>
          </h5>
        </div>
        <div id="collapseTwo" class="collapse" aria-labelledby="headingTwo" data-
↪parent="#accordion">
          <div class="card-body">
            Collapsible Group Item #2 content
          </div>
        </div>
      </div>
    </div>
    """,
    height=600,
)
```

20.1.2 Render an iframe URL

`components.iframe` is similar in features to `components.html`, with the difference being that `components.iframe` takes a URL as its input. This is used for situations where you want to include an entire page within a Streamlit app.

`streamlit.components.v1.`**`iframe`**(*src*, *width=None*, *height=None*, *scrolling=False*)

Load a remote URL in an iframe.

Parameters

- **`src`** (*str*) – The URL of the page to embed.
- **`width`** (*int*) – The width of the frame in CSS pixels. Defaults to the report's default element width.
- **`height`** (*int*) – The height of the frame in CSS pixels. Defaults to 150.
- **`scrolling`** (*bool*) – If True, show a scrollbar when the content is larger than the iframe. Otherwise, do not show a scrollbar. Defaults to False.

Example

```
import streamlit as st
import streamlit.components.v1 as components

# embed streamlit docs in a streamlit app
components.iframe("https://docs.streamlit.io/en/latest")
```

20.2 Create a bi-directional component

A bi-directional Streamlit Component has two parts:

1. A **frontend**, which is built out of HTML and any other web tech you like (JavaScript, React, Vue, etc.), and gets rendered in Streamlit apps via an iframe tag.
2. A **Python API**, which Streamlit apps use to instantiate and talk to that frontend

To make the process of creating bi-directional Streamlit Components easier, we've created a React template and a TypeScript-only template in the Streamlit Component-template GitHub repo. We also provide some example Components in the same repo.

20.2.1 Development Environment Setup

To build a Streamlit Component, you need the following installed in your development environment:

- Python 3.6+
- Streamlit 0.63+
- nodejs
- npm or yarn

Clone the component-template GitHub repo, then decide whether you want to use the React.js ("template") or plain TypeScript ("template-reactless") template.

1. Initialize and build the component template frontend from the terminal:

```
# React template
$ template/my_component/frontend
$ npm install    # Initialize the project and install npm dependencies
$ npm run start  # Start the Webpack dev server

# or

# TypeScript-only template
$ template-reactless/my_component/frontend
$ npm install    # Initialize the project and install npm dependencies
$ npm run start  # Start the Webpack dev server
```

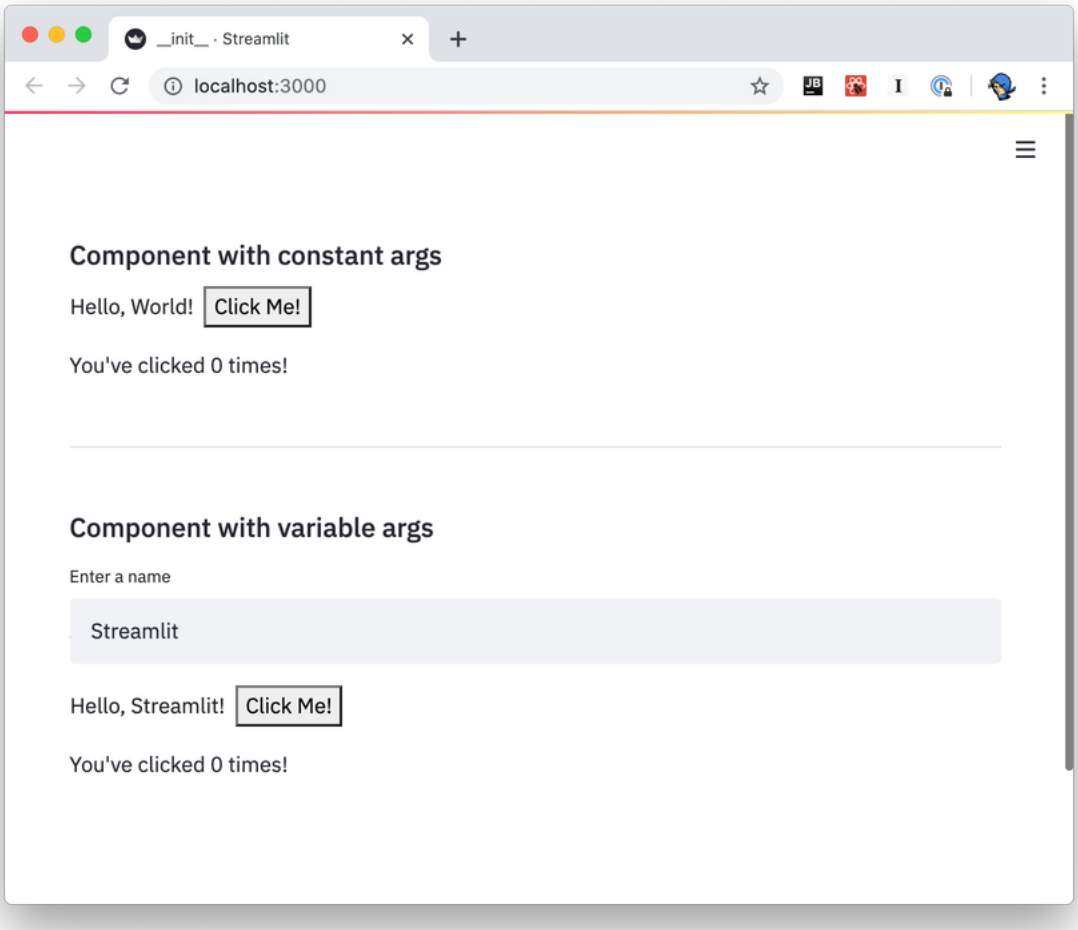
2. *From a separate terminal*, run the Streamlit app (Python) that declares and uses the component:

```
# React template
$ cd template
$ . venv/bin/activate # or similar to activate the venv/conda environment where␣
↪Streamlit is installed
$ streamlit run my_component/__init__.py # run the example

# or

# TypeScript-only template
$ cd template-reactless
$ . venv/bin/activate # or similar to activate the venv/conda environment where␣
↪Streamlit is installed
$ streamlit run my_component/__init__.py # run the example
```

After running the steps above, you should see a Streamlit app in your browser that looks like this:



The example app from the template shows how bi-directional communication is implemented. The Streamlit Component displays a button (`Python → JavaScript`), and the end-user can click the button. Each time the button is clicked, the JavaScript front-end increments the counter value and passes it back to Python (`JavaScript → Python`), which is then displayed by Streamlit (`Python → JavaScript`).

20.2.2 Frontend

Because each Streamlit Component is its own webpage that gets rendered into an `iframe`, you can use just about any web tech you'd like to create that web page. We provide two templates to get started with in the Streamlit Components-template GitHub repo; one of those templates uses React and the other does not.

Note: Even if you're not already familiar with React, you may still want to check out the React-based template. It handles most of the boilerplate required to send and receive data from Streamlit, and you can learn the bits of React you need as you go.

If you'd rather not use React, please read this section anyway! It explains the fundamentals of Streamlit ←→ Component communication.

React

The React-based template is in `template/my_component/frontend/src/MyComponent.tsx`.

- `MyComponent.render()` is called automatically when the component needs to be re-rendered (just like in any React app)
- Arguments passed from the Python script are available via the `this.props.args` dictionary:

```
# Send arguments in Python:
result = my_component(greeting="Hello", name="Streamlit")
```

```
// Receive arguments in frontend:
let greeting = this.props.args["greeting"]; // name = "Hello"
let name = this.props.args["name"]; // greeting = "Streamlit"
```

- Use `Streamlit.setComponentValue()` to return data from the component to the Python script:

```
// Set value in frontend:
Streamlit.setComponentValue(3.14);
```

```
# Access value in Python:
result = my_component(greeting="Hello", name="Streamlit")
st.write("result = ", result) # result = 3.14
```

When you call `Streamlit.setComponentValue(new_value)`, that new value is sent to Streamlit, which then *re-executes the Python script from top to bottom*. When the script is re-executed, the call to `my_component(...)` will return the new value.

From a *code flow* perspective, it appears that you're transmitting data synchronously with the frontend: Python sends the arguments to JavaScript, and JavaScript returns a value to Python, all in a single function call! But in reality this is all happening *asynchronously*, and it's the re-execution of the Python script that achieves the sleight of hand.

- Use `Streamlit.setFrameHeight()` to control the height of your component. By default, the React template calls this automatically (see `StreamlitComponentBase.componentDidUpdate()`). You can override this behavior if you need more control.
- There's a tiny bit of magic in the last line of the file: `export default withStreamlitConnection(MyComponent)` - this does some handshaking with Streamlit, and sets up the mechanisms for bi-directional data communication.

Typescript-only

The Typescript-only template is in `template-reactless/my_component/frontend/src/MyComponent.tsx`.

This template has much more code than its React sibling, in that all the mechanics of handshaking, setting up event listeners, and updating the component's frame height are done manually. The React version of the template handles most of these details automatically.

- Towards the bottom of the source file, the template calls `Streamlit.setComponentReady()` to tell Streamlit it's ready to start receiving data. (You'll generally want to do this after creating and loading everything that the Component relies on.)
- It subscribes to `Streamlit.RENDER_EVENT` to be notified of when to redraw. (This event won't be fired until `setComponentReady` is called)

- Within its `onRender` event handler, it accesses the arguments passed in the Python script via `event.detail.args`
- It sends data back to the Python script in the same way that the React template does - clicking on the "Click Me!" button calls `Streamlit.setComponentValue()`
- It informs Streamlit when its height may have changed via `Streamlit.setFrameHeight()`

Other frontend details

- Because you're hosting your component from a dev server (via `npm run start`), any changes you make should be automatically reflected in the Streamlit app when you save.
- If you want to add more packages to your component, run `npm add` to add them from within your component's `frontend/` directory.

```
npm add baseui
```

- To build a static version of your component, run `npm run build`. See *Prepare your Component* for more information

20.2.3 Python API

`components.declare_component()` is all that's required to create your Component's Python API:

```
import streamlit.components.v1 as components
my_component = components.declare_component(
  "my_component",
  url="http://localhost:3001"
)
```

You can then use the returned `my_component` function to send and receive data with your frontend code:

```
# Send data to the frontend using named arguments.
return_value = my_component(name="Blackbeard", ship="Queen Anne's Revenge")

# `my_component`'s return value is the data returned from the frontend.
st.write("Value = ", return_value)
```

While the above is all you need to define from the Python side to have a working Component, we recommend creating a "wrapper" function with named arguments and default values, input validation and so on. This will make it easier for end-users to understand what data values your function accepts and allows for defining helpful docstrings.

Please see this example from the Components-template for an example of creating a wrapper function.

20.2.4 Data serialization

Python → Frontend

You send data from Python to the frontend by passing keyword args to your Component's invoke function (that is, the function returned from `declare_component`). You can send the following types of data from Python to the frontend:

- Any JSON-serializable data
- `numpy.array`

- `pandas.DataFrame`

Any JSON-serializable data gets serialized to a JSON string, and deserialized to its JavaScript equivalent. `numpy.array` and `pandas.DataFrame` get serialized using Apache Arrow and are deserialized as instances of `ArrowTable`, which is a custom type that wraps Arrow structures and provides a convenient API on top of them.

Check out the CustomDataframe and SelectableDataTable Component example code for more context on how to use `ArrowTable`.

Frontend → Python

You send data from the frontend to Python via the `Streamlit.setComponentValue()` API (which is part of the template code). Unlike arg-passing from Python → frontend, **this API takes a single value**. If you want to return multiple values, you'll need to wrap them in an `Array` or `Object`.

Custom Components can send JSON-serializable data from the frontend to Python, as well as Apache Arrow `ArrowTables` to represent dataframes.

CHAPTER

TWENTYONE

TROUBLESHOOTING

These articles are designed to get you "unstuck" when something goes wrong. If you have questions, feature requests, or want to report an issue, please use our community forum or GitHub issues.

21.1 Sanity checks

If you're having problems running your Streamlit app, here are a few things to try out.

21.1.1 Check #0: Are you using a Streamlit-supported version of Python?

Streamlit will maintain backwards-compatibility with earlier Python versions as practical, guaranteeing compatibility with *at least* the last three minor versions of Python 3.

As new versions of Python are released, we will try to be compatible with the new version as soon as possible, though frequently we are at the mercy of other Python packages to support these new versions as well.

Streamlit currently supports versions 3.6, 3.7 and 3.8 of Python. Python 3.9 support is currently on-hold as we wait for pyarrow to support Python 3.9.

21.1.2 Check #1: Is Streamlit running?

On a Mac or Linux machine, type this on the terminal:

```
ps -Al | grep streamlit
```

If you don't see `streamlit run` in the output (or `streamlit hello`, if that's the command you ran) then the Streamlit server is not running. So re-run your command and see if the bug goes away.

21.1.3 Check #2: Is this an already-fixed Streamlit bug?

We try to fix bugs quickly, so many times a problem will go away when you upgrade Streamlit. So the first thing to try when having an issue is upgrading to the latest version of Streamlit:

```
pip install --upgrade streamlit
streamlit version
```

. . . and then verify that the version number printed is `0.73.1`.

Try reproducing the issue now. If not fixed, keep reading on.

21.1.4 Check #3: Are you running the correct Streamlit binary?

Let's check whether your Python environment is set up correctly. Edit the Streamlit script where you're experiencing your issue, **comment everything out, and add these lines instead:**

```
import streamlit as st
st.write(st.__version__)
```

. . . then call `streamlit run` on your script and make sure it says the same version as above. If not the same version, check out these instructions for some sure-fire ways to set up your environment.

21.1.5 Check #4: Is your browser caching your app too aggressively?

There are two easy ways to check this:

1. Load your app in a browser then press `Ctrl-Shift-R` or `-Shift-R` to do a hard refresh (Chrome/Firefox).
2. As a test, run Streamlit on another port. This way the browser starts the page with a brand new cache. For that, pass the `--server.port` argument to Streamlit on the command line:

   ```
   streamlit run my_app.py --server.port=9876
   ```

21.1.6 Check #5: Is this a Streamlit regression?

If you've upgraded to the latest version of Streamlit and things aren't working, you can downgrade at any time using this command:

```
pip install --upgrade streamlit==0.50
```

. . . where `0.50` is the version you'd like to downgrade to. See *Changelog* for a complete list of Streamlit versions.

21.1.7 Check #6 [Windows]: Is Python added to your PATH?

When installed by downloading from python.org, Python is not automatically added to the Windows system PATH. Because of this, you may get error messages like the following:

Command Prompt:

```
C:\Users\streamlit> streamlit hello
'streamlit' is not recognized as an internal or external command,
operable program or batch file.
```

PowerShell:

```
PS C:\Users\streamlit> streamlit hello
streamlit : The term 'streamlit' is not recognized as the name of a cmdlet, function,␣
↪script file, or operable program. Check the spelling of the name, or if a path was␣
↪included, verify that
the path is correct and try again.
At line:1 char:1
+ streamlit hello
+ ~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (streamlit:String) [],␣
↪CommandNotFoundException
    + FullyQualifiedErrorId : CommandNotFoundException
```

To resolve this issue, add Python to the Windows system PATH.

After adding Python to your Windows PATH, you should then be able to follow the instructions in our Get Started section.

21.1.8 Check #7 [Windows]: Do you need Build Tools for Visual Studio installed?

Starting with version 0.63 (July 2020), Streamlit added pyarrow as an install dependency as part of the Streamlit Components feature release. Occasionally, when trying to install Streamlit from PyPI, you may see errors such as the following:

```
Using cached pyarrow-1.0.1.tar.gz (1.3 MB)
  Installing build dependencies ... error
  ERROR: Command errored out with exit status 1:
   command: 'c:\users\streamlit\appdata\local\programs\python\python38-32\python.exe'
↪'c:\users\streamlit\appdata\local\programs\python\python38-32\lib\site-packages\pip
↪' install --ignore-installed --no-user --prefix 'C:\Users\streamlit\AppData\Local\
↪Temp\pip-build-env-s7owjrle\overlay' --no-warn-script-location --no-binary :none: --
↪only-binary :none: -i https://pypi.org/simple -- 'cython >= 0.29' 'numpy==1.14.5;␣
↪python_version<'"'"'3.7'"'"'' 'numpy==1.16.0; python_version>='"'"'3.7'"'"'␣
↪setuptools setuptools_scm wheel
       cwd: None

  Complete output (319 lines):

      Running setup.py install for numpy: finished with status 'error'
      ERROR: Command errored out with exit status 1:
       command: 'c:\users\streamlit\appdata\local\programs\python\python38-32\python.
↪exe' -u -c 'import sys, setuptools, tokenize; sys.argv[0] = '"'"'C:\\Users\\
↪streamlit\\AppData\\Local\\Temp\\pip-install-0jwfwx_u\\numpy\\setup.py'"'"'; __file_
↪_='"'"'C:\\Users\\streamlit\\AppData\\Local\\Temp\\pip-install-0jwfwx_u\\numpy\\
↪setup.py'"'"';f=getattr(tokenize, '"'"'open'"'"', open)(__file__);code=f.read().
↪replace('"'"'\r\n'"'"', '"'"'\n'"'"');f.close();exec(compile(code, __file__, '"'"'
↪'exec'"'"'))' install --record 'C:\Users\streamlit\AppData\Local\Temp\pip-record-
↪eys4l2gc\install-record.txt' --single-version-externally-managed --prefix 'C:\Users\
↪streamlit\AppData\Local\Temp\pip-build-env-s7owjrle\overlay' --compile --install-
↪headers 'C:\Users\streamlit\AppData\Local\Temp\pip-build-env-s7owjrle\overlay\
↪Include\numpy'
           cwd: C:\Users\streamlit\AppData\Local\Temp\pip-install-0jwfwx_u\numpy\
      Complete output (298 lines):

      blas_opt_info:
      blas_mkl_info:
      No module named 'numpy.distutils._msvccompiler' in numpy.distutils; trying from␣
↪distutils
      customize MSVCCompiler
        libraries mkl_rt not found in ['c:\\users\\streamlit\\appdata\\local\\
↪programs\\python\\python38-32\\lib', 'C:\\', 'c:\\users\\streamlit\\appdata\\local\\
↪programs\\python\\python38-32\\libs']
        NOT AVAILABLE

      blis_info:
      No module named 'numpy.distutils._msvccompiler' in numpy.distutils; trying from␣
↪distutils
      customize MSVCCompiler
        libraries blis not found in ['c:\\users\\streamlit\\appdata\\local\\programs\\
↪python\\python38-32\\lib', 'C:\\', 'c:\\users\\streamlit\\appdata\\local\\programs\\
↪python\\python38-32\\libs']
```

(continues on next page)

(continued from previous page)

```
        NOT AVAILABLE

      # <truncated for brevity> #

      c:\users\streamlit\appdata\local\programs\python\python38-32\lib\distutils\dist.
↪py:274: UserWarning: Unknown distribution option: 'define_macros'
        warnings.warn(msg)
      running install
      running build
      running config_cc
      unifing config_cc, config, build_clib, build_ext, build commands --compiler␣
↪options
      running config_fc
      unifing config_fc, config, build_clib, build_ext, build commands --fcompiler␣
↪options
      running build_src
      build_src
      building py_modules sources
      creating build
      creating build\src.win32-3.8
      creating build\src.win32-3.8\numpy
      creating build\src.win32-3.8\numpy\distutils
      building library "npymath" sources
      No module named 'numpy.distutils._msvccompiler' in numpy.distutils; trying from␣
↪distutils
      error: Microsoft Visual C++ 14.0 is required. Get it with "Build Tools for␣
↪Visual Studio": https://visualstudio.microsoft.com/downloads/
      ----------------------------------------
  ERROR: Command errored out with exit status 1: 'c:\users\streamlit\appdata\local\
↪programs\python\python38-32\python.exe' -u -c 'import sys, setuptools, tokenize;␣
↪sys.argv[0] = '"'"'C:\\Users\\streamlit\\AppData\\Local\\Temp\\pip-install-0jwfwx_u\
↪\numpy\\setup.py'"'"'; __file__='"'"'C:\\Users\\streamlit\\AppData\\Local\\Temp\\
↪pip-install-0jwfwx_u\\numpy\\setup.py'"'"';f=getattr(tokenize, '"'"'open'"'"',␣
↪open)(__file__);code=f.read().replace('"'"'\r\n'"'"', '"'"'\n'"'"');f.close();
↪exec(compile(code, __file__, '"'"'exec'"'"'))' install --record 'C:\Users\streamlit\
↪AppData\Local\Temp\pip-record-eys4l2gc\install-record.txt' --single-version-
↪externally-managed --prefix 'C:\Users\streamlit\AppData\Local\Temp\pip-build-env-
↪s7owjrle\overlay' --compile --install-headers 'C:\Users\streamlit\AppData\Local\
↪Temp\pip-build-env-s7owjrle\overlay\Include\numpy' Check the logs for full command␣
↪output.
  ----------------------------------------
```

This error indicates that Python is trying to compile certain libraries during install, but it cannot find the proper compilers on your system, as reflected by the line `error: Microsoft Visual C++ 14.0 is required. Get it with "Build Tools for Visual Studio"`.

Installing Build Tools for Visual Studio should resolve this issue.

21.2 App is not loading when running remotely

Below are a few common errors that occur when users spin up their own solution to host a Streamlit app remotely.

To learn about a deceptively simple way to host Streamlit apps that avoids all the issues below, check out Streamlit for Teams.

21.2.1 Symptom #1: The app never loads

When you enter the app's URL in a browser and all you see is a **blank page, a "Page not found" error, a "Conection refused" error**, or anything like that, first check that Streamlit is actually running on the remote server. On a Linux server you can SSH into it and then run:

```
ps -Al | grep streamlit
```

If you see Streamlit running, the most likely culprit is the Streamlit port not being exposed. The fix depends on your exact setup. Below are three example fixes:

- **Try port 80:** Some hosts expose port 80 by default. To set Streamlit to use that port, start Streamlit with the `--server.port` option:

  ```
  streamlit run my_app.py --server.port=80
  ```

- **AWS EC2 server**: First, click on your instance in the AWS Console. Then scroll down and click on *Security Groups* → *Inbound* → *Edit*. Next, add a *Custom TCP* rule that allows the *Port Range* `8501` with *Source* `0.0.0.0/0`.
- **Other types of server**: Check the firewall settings.

If that still doesn't solve the problem, try running a simple HTTP server instead of Streamlit, and seeing if *that* works correctly. If it does, then you know the problem lies somewhere in your Streamlit app or configuration (in which case you should ask for help in our forums!) If not, then it's definitely unrelated to Streamlit.

How to start a simple HTTP server:

```
python -m http.server [port]
```

21.2.2 Symptom #2: The app says "Please wait. . . " forever

If when you try to load your app in a browser you see a blue box in the center of the page with the text "Please wait. . . ", the underlying cause is likely one of the following:

- Misconfigured CORS protection.
- Server is stripping headers from the Websocket connection, thereby breaking compression.

To diagnose the issue, try temporarily disabling CORS protection by running Streamlit with the `--server.enableCORS` flag set to `false`:

```
streamlit run my_app.py --server.enableCORS=false
```

If this fixes your issue, **you should re-enable CORS protection** and then set `browser.serverPort` and `browser.serverAddress` to the URL and port of your Streamlit app.

If the issue persists, try disabling websocket compression by running Streamlit with the `--server.enableWebsocketCompression` flag set to `false`

```
streamlit run my_app.py --server.enableWebsocketCompression=false
```

If this fixes your issue, your server setup is likely stripping the `Sec-WebSocket-Extensions` HTTP header that is used to negotiate Websocket compression.

Compression is not required for Streamlit to work, but it's strongly recommended as it improves performance. If you'd like to turn it back on, you'll need to find which part of your infrastructure is stripping the `Sec-WebSocket-Extensions` HTTP header and change that behavior.

21.2.3 Symptom #3: Unable to upload files when running in multiple replicas

If the file uploader widget returns an error with status code 403, this is probably due to a misconfiguration in your app's XSRF protection logic.

To diagnose the issue, try temporarily disabling XSRF protection by running Streamlit with the `--server.enableXsrfProtection` flag set to `false`:

```
streamlit run my_app.py --server.enableXsrfProtection=false
```

If this fixes your issue, **you should re-enable XSRF protection** and then configure your app to use the same secret across every replica by setting the `server.cookieSecret` config option to the same hard-to-guess string everywhere.

21.3 How to clean install Streamlit

If you run into any issues while installing Streamlit or if you want to do a clean install, don't worry, we've got you covered. In this guide, we'll show you how to clean install Streamlit for Windows, macOS, and Linux.

21.3.1 Install Streamlit on Windows

Streamlit's officially-supported environment manager on Windows is Anaconda Navigator.

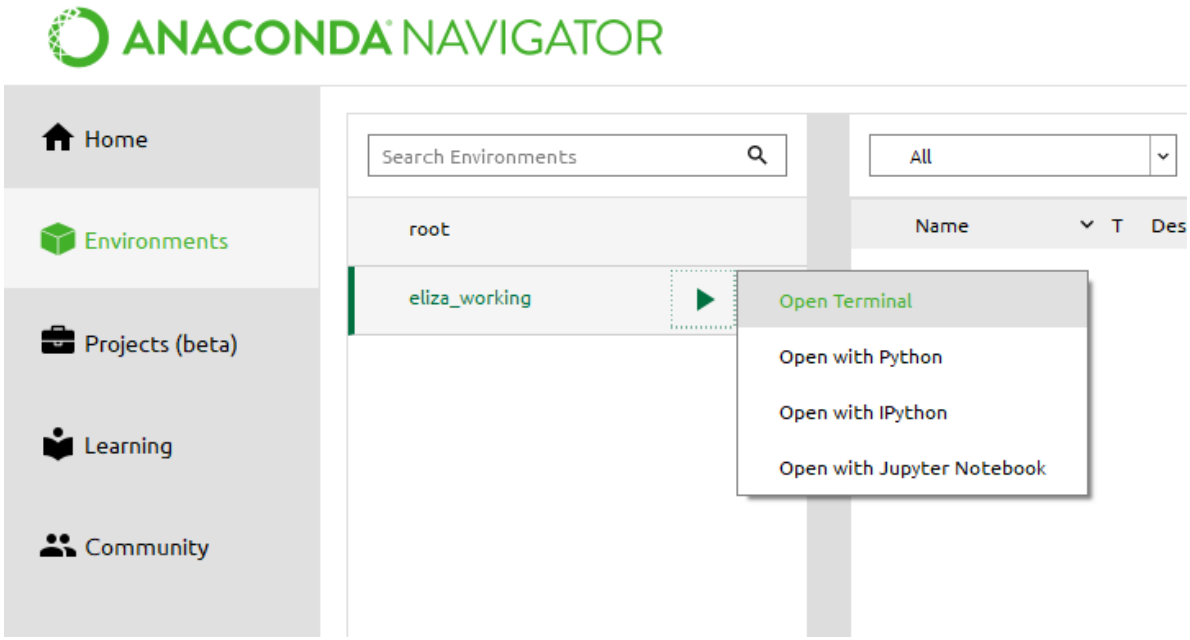
Install Anaconda

If you don't have Anaconda install yet, follow the steps provided on the Anaconda installation page.

Create a new environment with Streamlit

Next you'll need to set up your environment.

1. Follow the steps provided by Anaconda to set up and manage your environment using the Anaconda Navigator.
2. Select the "" icon next to your new environment. Then select "Open terminal":



3. In the terminal that appears, type:

```
pip install streamlit
```

4. Test that the installation worked:

```
streamlit hello
```

Streamlit's Hello app should appear in a new tab in your web browser!

Use your new environment

1. In Anaconda Navigator, open a terminal in your environment (see step 2 above).
2. In the terminal that appears, use Streamlit as usual:

```
streamlit run myfile.py
```

21.3.2 Install Streamlit on macOS/Linux

Streamlit's officially-supported environment manager for macOS and Linux is Pipenv. See instructions on how to install and use it below.

Install Pipenv

1. Install `pip`.

   On a macOS:

   ```
   sudo easy_install pip
   ```

   On Ubuntu with Python 3:

   ```
   sudo apt-get install python3-pip
   ```

   For other Linux distributions, see How to install PIP for Python.

2. Install `pipenv`.

   ```
   pip3 install pipenv
   ```

Create a new environment with Streamlit

1. Navigate to your project folder:

   ```
   cd myproject
   ```

2. Create a new Pipenv environment in that folder and activate that environment:

   ```
   pipenv shell
   ```

   When you run the command above, a file called `Pipfile` will appear in `myprojects/`. This file is where your Pipenv environment and its dependencies are declared.

3. Install Streamlit in your environment:

   ```
   pip install streamlit
   ```

   Or if you want to create an easily-reproducible environment, replace `pip` with `pipenv` every time you install something:

   ```
   pipenv install streamlit
   ```

4. Test that the installation worked:

   ```
   streamlit hello
   ```

   Streamlit's Hello app should appear in a new tab in your web browser!

Use your new environment

1. Any time you want to use the new environment, you first need to go to your project folder (where the `Pipenv` file lives) and run:

```
pipenv shell
```

2. Now you can use Python and Streamlit as usual:

```
streamlit run myfile.py
```

3. When you're done using this environment, just type `exit` or press `ctrl-D` to return to your normall shell.

21.4 Caching issues

While developing an app, if you see an error or warning that stems from a cached function, it's probably related to the hashing procedure described in the *Improve app performance*. In this article, we'll provide solutions to common issues encountered when using caching. If you have an issue that's not covered in this article, please let us know in the community forum.

21.4.1 How to debug a cached function that isn't executing

If you believe your cached function isn't executing even though its inputs are a "Cache miss", you can debug using `st.write` statements inside and outside of your function like this:

```
@st.cache
def my_cached_func(a, b):
    st.write("Cache miss: my_cached_func(", a, ", ", b, ") ran")
    ...

st.write("Calling my_cached_func(", a, ", ", b, ")")
my_cached_func(2, 21)
```

21.4.2 How to fix an `UnhashableTypeError`

Streamlit raises this error whenever it encounters a type it doesn't know how to hash. This could be either when hashing the inputs to generate the cache key or when hashing the output to verify whether it changed. To address it, you'll need to help Streamlit understand how to hash that type by using the `hash_funcs` argument:

```
@st.cache(hash_funcs={FooType: hash_foo_type})
def my_cached_func(a, b):
    ...
```

Here, `FooType` is the type Streamlit was unable to hash, and `hash_foo_type` is a function that can be used to properly hash `FooType` objects.

For example, if you'd like to make Streamlit ignore a specific type of object when hashing, you can pass a constant function to `hash_funcs`, like this:

```
@st.cache(hash_funcs={FooType: lambda _: None})
def my_cached_func(a, b):
    ...
```

For more information, see Improve app performance.

21.4.3 How to fix the Cached Object Mutated warning

By default Streamlit expects its cached values to be treated as immutable – that cached objects remain constant. You received this warning if your code modified a cached object (see Example 5 in Caching). When this happens, you have a few options:

1. If you don't understand why you're seeing this error, it's very likely that you didn't mean to mutate the cached value in the first place. So you should either:

- **Preferred:** rewrite your code to remove that mutation
- Clone the output of the cached function before mutating it. For example:

```
import copy
cloned_output = copy.deepcopy(my_cached_function(...))
```

1. If you wanted to allow the cached object to mutate, you can disable this check by setting `allow_output_mutation=True` like this:

```
@st.cache(allow_output_mutation=True)
def my_cached_func(...):
    ...
```

For examples, see *Advanced caching*.

Note: If your function returns multiple objects and you only want to allow a subset of them to mutate between runs, you can do that with the *hash_funcs* option.

---

2. If Streamlit is incorrectly hashing the cached object, you can override this by using `hash_funcs`. For example, if your function returns an object of type `FooType` then you could write:

```
@st.cache(hash_funcs={FooType: hash_func_for_foo_type})
def my_cached_func(...):
    ...
```

For more information, see Improve app performance.

By the way, the scenario above is fairly unlikely — unless `FooType` does something particularly tricky internally. This is the case with some `SpaCY` objects, which can automatically mutate behind the scenes for better performance, while keeping their semantics constant. That means Streamlit will correctly detect a mutation in the object's internal structure, even though semantically that mutation makes no difference.

21.4.4 If all else fails

If the proposed fixes above don't work for you, or if you have an idea on how to further improve `@st.cache` – let us know by asking questions in the community forum, filing a bug, or submitting a feature request. We love hearing back from the community!

CHAPTER

TWENTYTWO

STREAMLIT FREQUENTLY ASKED QUESTIONS

Here are some frequently asked questions about Streamlit and Streamlit Components. If you feel something important is missing that everyone needs to know, please open an issue or submit a pull request and we'll be happy to review it!

22.1 Using Streamlit

1. **How can I make `st.pydeck_chart` use custom Mapbox styles?**

   If you are supplying a Mapbox token, but the resulting `pydeck_chart` doesn't show your custom Mapbox styles, please check that you are adding the Mapbox token to the Streamlit `config.toml` configuration file. Streamlit DOES NOT read Mapbox tokens from inside of a PyDeck specification (i.e. from inside of the Streamlit app). Please see this forum thread for more information.

22.2 Manually deploying Streamlit

1. **How do I deploy Streamlit on a domain so it appears to run on a regular port (i.e. port 80)?**

   - You should use a **reverse proxy** to forward requests from a webserver like Apache or Nginx to the port where your Streamlit app is running. You can accomplish this in several different ways. The simplest way is to forward all requests sent to your domain so that your Streamlit app appears as the content of your website.
   - Another approach is to configure your webserver to forward requests to designated subfolders (e.g. *http://awesomestuff.net/streamlitapp*) to different Streamlit apps on the same domain, as in this example config for Nginx submitted by a Streamlit community member.

2. **How can I deploy multiple Streamlit apps on different subdomains?**

   Like running your Streamlit app on more common ports such as 80, subdomains are handled by a web server like Apache or Nginx:

   - Set up a web server on a machine with a public IP address, then use a DNS server to point all desired subdomains to your webserver's IP address
   - Configure your web server to route requests for each subdomain to the different ports that your Streamlit apps are running on

   For example, let's say you had two Streamlit apps called `Calvin` and `Hobbes`. App `Calvin` is running on port **8501**. You set up app `Hobbes` to run on port **8502**. Your webserver would then be set up to "listen" for requests on subdomains `calvin.somedomain.com` and `hobbes.subdomain.com`, and route requests to port **8501** and **8502**, respectively.

Check out these two tutorials for Apache2 and Nginx that deal with setting up a webserver to redirect subdomains to different ports:

- Apache2 subdomains
- NGinx subdomains

3. **How do I deploy Streamlit on Heroku, AWS, Google Cloud, etc...?**

   Here are some user-submitted tutorials for different cloud services:

   - How to Deploy Streamlit to a Free Amazon EC2 instance, by Rahul Agarwal
   - Host Streamlit on Heroku, by Maarten Grootendorst
   - Host Streamlit on Azure, by Richard Peterson
   - Host Streamlit on 21YunBox, by Toby Lei

4. **Does Streamlit support the WSGI Protocol? (aka Can I deploy Streamlit with gunicorn?)**

   Streamlit does not support the WSGI protocol at this time, so deploying Streamlit with (for example) gunicorn is not currently possible. Check out this thread regarding deploying Streamlit in a gunicorn-like manner to see how other users have accomplished this.

22.3 Streamlit Components

Below are some selected questions we've received about Streamlit Components. If you don't find your question here, take a look on the Streamlit community forum via the Components tag.

1. **How do Streamlit Components differ from functionality provided in the base Streamlit package?**

   - Streamlit Components are wrapped up in an iframe, which gives you the ability to do whatever you want (within the iframe) using any web technology you like.
   - There is a strict message protocol between Components and Streamlit, which makes possible for Components to act as widgets. As Streamlit Components are wrapped in iframe, they cannot modify their parent's DOM (a.k.a the Streamlit report), which ensures that Streamlit is always secure even with user-written components.

2. **What types of things aren't possible with Streamlit Components?**

   Because each Streamlit Component gets mounted into its own sandboxed iframe, this implies a few limitations on what is possible with Components:

   - **Can't communicate with other Components**: Components can't contain (or otherwise communicate with) other components, so Components cannot be used to build something like `grid_layout`
   - **Can't modify CSS**: A Component can't modify the CSS that the rest of the Streamlit app uses, so you can't create something like `dark_mode`
   - **Can't add/remove elements**: A Component can't add or remove other elements of a Streamlit app, so you couldn't make something like `remove_streamlit_hamburger_menu`

3. **How do I build a Component that can be displayed in the sidebar?**

   Currently, it is not possible to create a component in the sidebar, but we're hoping to release that functionality in a future release.

4. **My Component seems to be blinking/stuttering. . . how do I fix that?**

   Currently, no automatic debouncing of Component updates is performed within Streamlit. The Component creator themselves can decide to rate-limit the updates they send back to Streamlit.

CHAPTER

TWENTYTHREE

CHANGELOG

This page lists highlights, bug fixes, and known issues for official Streamlit releases. If you're looking for information about nightly releases, beta features, or experimental features, see Try pre-release features.

Tip: To upgrade to the latest version of Streamlit, run:

```
$ pip install --upgrade streamlit
```

23.1 Version 0.73.0

Release date: December 17, 2020

Notable Changes

- Streamlit can now be installed on Python 3.9. Streamlit components are not yet compatible with Python 3.9 and must use version 3.8 or earlier.
- Streamlit Components now allows same origin, enabling features provided by the browser such as a webcam component.
- Fix Streamlit Share deploy experience for users running on Git versions 2.7.0 or earlier.
- Handle unexpected closing of uploaded files for `st.file_uploader`.

23.2 Version 0.72.0

Release date: December 2, 2020

Notable Changes

- Establish a framework for theming and migrate existing components.
- Improve the sidebar experience for mobile devices.
- Update `st.file_uploader` to reduce reruns.

23.3 Version 0.71.0

Release date: November 11, 2020

Notable Changes

- Updated `st.file_uploader` to automatically reset buffer on app reruns.
- Optimize the default rendering of charts and reduce issues with the initial render.

23.4 Version 0.70.0

Release date: October 28, 2020

Notable Changes

- `st.set_page_config` and `st.color_picker` have now been moved into the Streamlit namespace. These will be removed from beta January 28th, 2021. Learn more about our beta process here.
- Improve display of bar charts for discrete values.

23.5 Version 0.69.0

Release date: October 15, 2020

Highlights:

- Introducing Streamlit sharing, the best way to deploy, manage, and share your public Streamlit apps - for free. Read more about it on our blog post or sign up here!
- Added `st.experimental_rerun` to programatically re-run your app. Thanks SimonBiggs!

Notable Changes

- Better support across browsers for start and stop times for st.video.
- Bug fix for intermittently failing media files
- Bug fix for custom components compatibility with Safari. Make sure to upgrade to the latest streamlit-component-lib.

23.6 Version 0.68.0

Release date: October 8, 2020

Highlights:

- Introducing new layout options for Streamlit! Move aside, vertical layout. Make a little space for... horizontal layout! Check out our blog post.
- File uploader redesigned with new functionality for multiple files uploads and better support for working with uploaded files. This may cause breaking changes. Please see the new api in our documentation

Notable Changes

- `st.balloon` has gotten a facelift with nicer balloons and smoother animations.

- Breaking Change: Following the deprecation of `st.deck_gl_chart` in January 2020, we have now removed the API completely. Please use `st.pydeck_chart` instead.
- Breaking Change: Following the deprecation of `width` and `height` for `st.altair_chart`, `st.graphviz_chart`, `st.plotly_chart`, and `st.vega_lite_chart` in January 2020, we have now removed the args completely. Please set the width and height in the respective charting library.

23.7 Version 0.67.0

Release date: September 16, 2020

Highlights:

- Streamlit Components can now return bytes to your Streamlit App. To create a component that returns bytes, make sure to upgrade to the latest streamlit-component-lib.

Notable Changes

- Deprecation warning: Beginning December 1st, 2020 `st.pyplot()` will require a figure to be provided. To disable the deprecation warning, please set `deprecation.showPyplotGlobalUse` to `False`
- `st.multiselect` and `st.select` are now lightning fast when working with large datasets. Thanks masa3141!

23.8 Version 0.66.0

Release date: September 1, 2020

Highlights:

- `st.write` is now available for use in the sidebar!
- A slider for distinct or non-numerical values is now available with `st.select_slider`.
- Streamlit Components can now return dataframes to your Streamlit App. Check out our SelectableDataTable example.
- The Streamlit Components library used in our Streamlit Component template is now available as a npm package (streamlit-component-lib) to simplify future upgrades to the latest version. Existing components do not need to migrate.

Notable Changes

- Support StringDtype from pandas version 1.0.0
- Support for running Streamlit on Unix sockets

23.9 Version 0.65.0

Release date: August 12, 2020

Highlights:

- Ability to set page title, favicon, sidebar state, and wide mode via st.beta_set_page_config(). See our documentation for details.
- Add stateful behaviors through the use of query parameters with st.experimental_set_query_params and st.experimental_get_query_params. Thanks @zhaoooyue!
- Improved pandas dataframe support for st.radio, st.selectbox, and st.multiselect.
- Break out of your Streamlit app with st.stop.
- Inline SVG support for st.image.

Callouts:

- Deprecation Warning: The st.image parameter format has been renamed to output_format.

23.10 Version 0.64.0

Release date: July 23, 2020

Highlights:

- Default matplotlib to display charts with a tight layout. To disable this, set `bbox_inches` to `None`, inches as a string, or a `Bbox`
- Deprecation warning for automatic encoding on `st.file_uploader`
- If `gatherUserStats` is `False`, do not even load the Segment library. Thanks @tanmaylaud!

23.11 Version 0.63.0

Release date: July 13, 2020

Highlights:

- **Support for Streamlit Components!!!** See documentation for more info.
- Support for datetimes in `st.slider`. And, of course, just like any other value you use in `st.slider`, you can also pass in two-element lists to get a datetime range slider.

23.12 Version 0.62.0

Release date: June 21, 2020

Highlights:

- Ability to turn websocket compression on/off via the config option `server.enableWebsocketCompression`. This is useful if your server strips HTTP headers and you do not have access to change that behavior.

- Out-of-the-box support for CSRF protection using the Cookie-to-header token technique. This means that if you're serving your Streamlit app from multiple replicas you'll need to configure them to to use the same cookie secret with the `server.cookieSecret` config option. To turn XSRF protection off, set `server.enableXsrfProtection=false`.

Notable bug fixes:

- Added a grace period to the image cache expiration logic in order to fix multiple related bugs where images sent with `st.image` or `st.pyplot` were sometimes missing.

23.13 Version 0.61.0

Release date: June 2, 2020

Highlights:

- Support for date ranges in `st.date_picker`. See docs for more info, but the TLDR is: just pass a list/tuple as the default date and it will be interpreted as a range.
- You can now choose whether `st.echo` prints the code above or below the output of the echoed block. To learn more, refer to the `code_location` argument in the docs.
- Improved `@st.cache` support for Keras models and Tensorflow `saved_models`.

23.14 Version 0.60.0

Release date: May 18, 2020

Highlights:

- Ability to set the height of an `st.text_area` with the `height` argument (expressed in pixels). See docs for more.
- Ability to set the maximimum number of characters allowed in `st.text_area` or `st.text_input`. Check out the `max_chars` argument in the docs.
- Better DeckGL support for the H3 geospatial indexing system. So now you can use things like `H3HexagonLayer` in `st.pydeck_chart`.
- Improved `@st.cache` support for PyTorch TensorBase and Model.

23.15 Version 0.59.0

Release date: May 05, 2020

Highlights:

- New color-picker widget! Use it with `st.beta_color_picker()`
- Introducing `st.beta_*` and `st.experimental_*` function prefixes, for faster Streamlit feature releases. See docs for more info.
- Improved `@st.cache` support for SQL Alchemy objects, CompiledFFI, PyTorch Tensors, and `builtins.mappingproxy`.

23.16 Version 0.58.0

Release date: April 22, 2020

Highlights:

- Made `st.selectbox` filtering case-insensitive.
- Better support for Tensorflow sessions in `@st.cache`.
- Changed behavior of `st.pyplot` to auto-clear the figure only when using the global Matplotlib figure (i.e. only when calling `st.pyplot()` rather than `st.pyplot(fig)`).

23.17 Version 0.57.0

Release date: March 26, 2020

Highlights:

- Ability to set expiration options for `@st.cache`'ed functions by setting the `max_entries` and `ttl` arguments. See docs.
- Improved the machinery behind `st.file_uploader`, so it's much more performant now! Also increased the default upload limit to 200MB (configurable via `server.max_upload_size`).
- The `server.address` config option now *binds* the server to that address for added security.
- Even more details added to error messages for `@st.cache` for easier debugging.

23.18 Version 0.56.0

Release date: February 15, 2020

Highlights:

- Improved error messages for st.cache. The errors now also point to the new caching docs we just released. Read more here!

Breaking changes:

- As announced last month, **Streamlit no longer supports Python 2.** To use Streamlit you'll need Python 3.5 or above.

23.19 Version 0.55.0

Release date: February 4, 2020

Highlights:

- **Ability to record screencasts directly from Streamlit!** This allows you to easily record and share explanations about your models, analyses, data, etc. Just click then "Record a screencast". Give it a try!

23.20 Version 0.54.0

Release date: January 29, 2020

Highlights:

- Support for password fields! Just pass `type="password"` to `st.text_input()`.

Notable fixes:

- Numerous st.cache improvements, including better support for complex objects.
- Fixed cross-talk in sidebar between multiple users.

Breaking changes:

- If you're using the SessionState hack Gist, you should re-download it! Depending on which hack you're using, here are some links to save you some time:
  - SessionState.py
  - st_state_patch.py

23.21 Version 0.53.0

Release date: January 14, 2020

Highlights:

- Support for all DeckGL features! Just use Pydeck instead of `st.deck_gl_chart`. To do that, simply pass a PyDeck object to `st.pydeck_chart`, `st.write`, or magic.

  *Note that as a **preview release** things may change in the near future. Looking forward to hearing input from the community before we stabilize the API!*

  The goals is for this to replace `st.deck_gl_chart`, since it is does everything the old API did *and much more!*

- Better handling of Streamlit upgrades while developing. We now auto-reload the browser tab if the app it is displaying uses a newer version of Streamlit than the one the tab is running.
- New favicon, with our new logo!

Notable fixes:

- Magic now works correctly in Python 3.8. It no longer causes docstrings to render in your app.

Breaking changes:

- Updated how we calculate the default width and height of all chart types. We now leave chart sizing up to your charting library itself, so please refer to the library's documentation.

  As a result, the `width` and `height` arguments have been deprecated from most chart commands, and `use_container_width` has been introduced everywhere to allow you to make charts fill as much horizontal space as possible (this used to be the default).

23.22 Version 0.52.0

Release date: December 20, 2019

Highlights:

- Preview release of the file uploader widget. To try it out just call `st.file_uploader`!

  *Note that as a **preview release** things may change in the near future. Looking forward to hearing input from the community before we stabilize the API!*

- Support for emoji codes in `st.write` and `st.markdown`! Try it out with `st.write("Hello :wave:")`.

Breaking changes:

- `st.pyplot` now clears figures by default, since that's what you want 99% of the time. This allows you to create two or more Matplotlib charts without having to call `pyplot.clf` every time. If you want to turn this behavior off, use `st.pyplot(clear_figure=False)`
- `st.cache` no longer checks for input mutations. This is the first change of our ongoing effort to simplify the caching system and prepare Streamlit for the launch of other caching primitives like Session State!

23.23 Version 0.51.0

Release date: November 30, 2019

Highlights:

- You can now tweak the behavior of the file watcher with the config option `server.fileWatcherType`. Use it to switch between:
  - `auto` (default) : Streamlit will attempt to use the watchdog module, and falls back to polling if watchdog is not available.
  - `watchdog` : Force Streamlit to use the watchdog module.
  - `poll` : Force Streamlit to always use polling.
  - `none` : Streamlit will not watch files.

Notable bug fixes:

- Fix the "keyPrefix" option in static report sharing #724
- Add support for getColorX and getTargetColorX to DeckGL Chart #718
- Fixing Tornado on Windows + Python 3.8 #682
- Fall back on webbrowser if xdg-open is not installed on Linux #701
- Fixing number input spin buttons for Firefox #683
- Fixing CTRL+ENTER on Windows #699
- Do not automatically create credential file when in headless mode #467

23.24 Version 0.50.1

Release date: November 10, 2019

Highlights:

- SymPy support and ability to draw mathematical expressions using LaTeX! See `st.latex`, `st.markdown`, and `st.write`.
- You can now set config options using environment variables. For example, `export STREAMLIT_SERVER_PORT=9876`.
- Ability to call `streamlit run` directly with Github and Gist URLs. No need to grab the "raw" URL first!
- Cleaner exception stack traces. We now remove all Streamlit-specific code from stack traces originating from the user's app.

23.25 Version 0.49.0

Release date: October 23, 2019

Highlights:

- New input widget for entering numbers with the keyboard: `st.number_input()`
- Audio/video improvements: ability to load from a URL, to embed YouTube videos, and to set the start position.
- You can now (once again) share static snapshots of your apps to S3! See the S3 section of `streamlit config show` to set it up. Then share from top-right menu.
- Use `server.baseUrlPath` config option to set Streamlit's URL to something like `http://domain.com/customPath`.

Notable bug fixes:

- Fixes numerous Windows bugs, including Issues #339 and #401.

23.26 Version 0.48.0

Release date: October 12, 2019

Highlights:

- Ability to set config options as command line flags or in a local config file.
- You can now maximize charts and images!
- Streamlit is now much faster when writing data in quick succession to your app.
- Ability to blacklist folder globs from "run on save" and `@st.cache` hashing.
- Improved handling of widget state when Python file is modified.
- Improved HTML support in `st.write` and `st.markdown`. HTML is still unsafe, though!

Notable bug fixes:

- Fixes `@st.cache` bug related to having your Python environment on current working directory. Issue #242
- Fixes loading of root url `/` on Windows. Issue #244

23.27 Version 0.47.0

Release date: October 1, 2019

Highlights:

- New hello.py showing off 4 glorious Streamlit apps. Try it out!
- Streamlit now automatically selects an unused port when 8501 is already in use.
- Sidebar support is now out of beta! Just start any command with `st.sidebar.` instead of `st.`
- Performance improvements: we added a cache to our websocket layer so we no longer re-send data to the browser when it hasn't changed between runs
- Our "native" charts `st.line_chart`, `st.area_chart` and `st.bar_chart` now use Altair behind the scenes
- Improved widgets: custom st.slider labels; default values in multiselect
- The filesystem watcher now ignores hidden folders and virtual environments
- Plus lots of polish around caching and widget state management

Breaking change:

- We have temporarily disabled support for sharing static "snapshots" of Streamlit apps. Now that we're no longer in a limited-access beta, we need to make sure sharing is well thought through and abides by laws like the DMCA. But we're working on a solution!

23.28 Version 0.46.0

Release date: September 19, 2019

Highlights:

- Magic commands! Use `st.write` without typing `st.write`. See https://docs.streamlit.io/en/latest/api.html#magic-commands
- New `st.multiselect` widget.
- Fixed numerous install issues so now you can use `pip install streamlit` even in Conda! We've therefore deactivated our Conda repo.
- Multiple bug fixes and additional polish in preparation for our launch!

Breaking change:

- HTML tags are now blacklisted in `st.write`/`st.markdown` by default. More information and a temporary work-around at: https://github.com/streamlit/streamlit/issues/152

23.29 Version 0.45.0

Release date: August 28, 2019

Highlights:

- Experimental support for *sidebar*! Let us know if you want to be a beta tester.
- Completely redesigned `st.cache`! Much more performant, has a cleaner API, support for caching functions called by `@st.cached` functions, user-friendly error messages, and much more!
- Lightning fast `st.image`, ability to choose between JPEG and PNG compression, and between RGB and BGR (for OpenCV).
- Smarter API for `st.slider`, `st.selectbox`, and `st.radio`.
- Automatically fixes the Matplotlib backend – no need to edit .matplotlibrc

23.30 Version 0.44.0

Release date: July 28, 2019

Highlights:

- Lightning-fast reconnect when you do a ctrl-c/rerun on your Streamlit code
- Useful error messages when the connection fails
- Fixed multiple bugs and improved polish of our newly-released interactive widgets

23.31 Version 0.43.0

Release date: July 9, 2019

Highlights:

- Support for interactive widgets!

23.32 Version 0.42.0

Release date: July 1, 2019

Highlights:

- Ability to save Vega-Lite and Altair charts to SVG or PNG
- We now cache JS files in your browser for faster loading
- Improvements to error-handling inside Streamlit apps

23.33 Version 0.41.0

Release date: June 24, 2019

Highlights:

- Greatly improved our support for named datasets in Vega-Lite and Altair
- Added ability to ignore certain folders when watching for file changes. See the `server.folderWatchBlacklist` config option.
- More robust against syntax errors on the user's script and imported modules

23.34 Version 0.40.0

Release date: June 10, 2019

Highlights:

- Streamlit is more than 10x faster. Just save and watch your analyses update instantly.
- We changed how you run Streamlit apps: `$ streamlit run your_script.py [script args]`
- Unlike the previous versions of Streamlit, `streamlit run [script] [script args]` creates a server (now you don't need to worry if the proxy is up). To kill the server, all you need to do is hit **Ctrl+c**.

Why is this so much faster?

Now, Streamlit keeps a single Python session running until you kill the server. This means that Streamlit can re-run your code without kicking off a new process; imported libraries are cached to memory. An added bonus is that `st.cache` now caches to memory instead of to disk.

What happens if I run Streamlit the old way?

If you run `$ python your_script.py` the script will execute from top to bottom, but won't produce a Streamlit app.

What are the limitations of the new architecture?

- To switch Streamlit apps, first you have to kill the Streamlit server with **Ctrl-c**. Then, you can use `streamlit run` to generate the next app.
- Streamlit only works when used inside Python files, not interactively from the Python REPL.

What else do I need to know?

- The strings we print to the command line when **liveSave** is on have been cleaned up. You may need to adjust any RegEx that depends on those.
- A number of config options have been renamed:

Old config	New config
proxy.isRemote	server.headless
proxy.liveSave	server.liveSave
proxy.runOnSave, proxy.watchFileSystem	server.runOnSave
proxy.enableCORS	server.enableCORS
proxy.port	server.port
browser.proxyAddress	browser.serverAddress
browser.proxyPort	browser.serverPort
client.waitForProxySecs	n/a
client.throttleSecs	n/a
client.tryToOutliveProxy	n/a
client.proxyAddress	n/a
client.proxyPort	n/a
proxy.autoCloseDelaySecs	n/a
proxy.reportExpirationSecs	n/a

What if something breaks?

If the new Streamlit isn't working, please let us know by Slack or email. You can downgrade at any time with these commands:

```
$ pip install --upgrade streamlit==0.37
```

```
$ conda install streamlit=0.37
```

What's next?

Thank you for staying with us on this journey! This version of Streamlit lays the foundation for interactive widgets, a new feature of Streamlit we're really excited to share with you in the next few months.

23.35 Version 0.36.0

Release date: May 03, 2019

Highlights

- `st.progress()` now also accepts floats from 0.0–1.0
- Improved rendering of long headers in DataFrames
- Shared apps now default to HTTPS

23.36 Version 0.35.0

Release date: April 26, 2019

Highlights

- Bokeh support! Check out docs for `st.bokeh_chart`
- Improved the size and load time of saved apps
- Implemented better error-catching throughout the codebase

CHAPTER

TWENTYFOUR

WELCOME TO STREAMLIT

Streamlit is an open-source Python library that makes it easy to create and share beautiful, custom web apps for machine learning and data science. In just a few minutes you can build and deploy powerful data apps - so let's get started!

1. Make sure that you have Python 3.6 - Python 3.8 installed.
2. Install Streamlit using PIP and run the 'hello world' app:

```
pip install streamlit
streamlit hello
```

3. That's it! In the next few seconds the sample app will open in a new tab in your default browser.

Still with us? Great! Now make your own app in just 3 more steps:

1. Open a new Python file, import Streamlit, and write some code
2. Run the file with:

   ```
   streamlit run [filename]
   ```

3. When you're ready, click 'Deploy' from the Streamlit menu to *share your app with the world*!

Now that you're set up, let's dive into more of how Streamlit works and how to build great apps.

24.1 How to use our docs

The docs are broken up into 5 sections that will help you get the most out of Streamlit.

- **Tutorials**: include our *Get Started* guide and a few step-by-step examples to building different types of apps in Streamlit.
- **Topic guides**: give you background on how different parts of Streamlit work. Make sure to check out the sections on *Creating an app* and *Deploying an app*, and for you advanced users who want to level up your apps, be sure to read up on *Caching* and *Components*.
- **Cookbook**: provides short code snippets that you can copy in for specific use cases.
- **Reference guides**: are the bread and butter of how our *APIs* and *configuration files* work and will give you short, actionable explanations of specific functions and features.
- **Support**: gives you more options for when you're stuck or want to talk about an idea. Check out our discussion forum as well as a number of *troubleshooting guides*.

24.2 Join the community

Streamlit is more than just a way to make data apps, it's also a community of creators that share their apps and ideas and help each other make their work better. Please come join us on the community forum. We love to hear your questions, ideas, and help you work through your bugs — stop by today!

INDEX

T

V

W