|  |
| --- |
| 비플레이스 |
| Streamlit Hands On |
| PART FIVE: DESIGNING AN APPLICATION WITH STREAMLIT |

|  |
| --- |
| 김무경  2023-10-26 |

내용

[1. 원본 3](#_Toc149249374)

[2. Introduction to Streamlit 3](#_Toc149249375)

[1.1. Creating Our First App 4](#_Toc149249376)

[1.1.1. Options for Application Development in Python 4](#_Toc149249377)

[1.1.2. Installing Streamlit 5](#_Toc149249378)

[1.1.3. Creating a Home Page 6](#_Toc149249379)

[1.2. Displaying Data in Streamlit 8](#_Toc149249380)

[1.2.1. Displaying Text to Users 9](#_Toc149249381)

[1.2.2. Displaying Python Data Structures 14](#_Toc149249382)

[1.2.3. Displaying Tabular Data 17](#_Toc149249383)

[1.2.4. Displaying Multimedia in Streamlit 21](#_Toc149249384)

[1.3. Streamlit Input Widgets 23](#_Toc149249385)

[1.3.1. Text Input Widgets 24](#_Toc149249386)

[1.3.2. Numerical Input Widgets 26](#_Toc149249387)

[1.3.3. Date and Time Input Widgets 28](#_Toc149249388)

[1.3.4. Boolean Input Widgets 30](#_Toc149249389)

[1.3.5. Selection Widgets 31](#_Toc149249390)

[3. Advanced Streamlit Features 37](#_Toc149249391)

[3.1. Data Visualization 37](#_Toc149249392)

[3.1.1. Metrics 38](#_Toc149249393)

[3.1.2. Plotting Basic Graphs with Streamlit 40](#_Toc149249394)

[3.1.3. Map Charts 45](#_Toc149249395)

[3.2. Layout Design 45](#_Toc149249396)

[3.2.1. Layout Widgets 46](#_Toc149249397)

[3.3. Streamlit Cache and Session States 52](#_Toc149249398)

[3.3.1. Caching Data with st.cache 53](#_Toc149249399)

[3.4. Storing Data with st.session\_state 53](#_Toc149249400)

[3.5. Custom HTML 55](#_Toc149249401)

[3.6. Multi-Page Applications 56](#_Toc149249402)

[4. Building a Database Query Application 57](#_Toc149249403)

[4.1. Building a Database Query Application 57](#_Toc149249404)

[4.1.1. Importing the Libraries 61](#_Toc149249405)

[4.1.2. Caching Data 61](#_Toc149249406)

[4.1.3. Creating our App Layout 63](#_Toc149249407)

[4.1.4. Using User Inputs to Produce a New DataFrame 65](#_Toc149249408)

[4.2. Deploying an App in the Cloud with Streamlit Share 68](#_Toc149249409)

[4.2.1. Create a GitHub Account 68](#_Toc149249410)

[4.2.2. Upload Application to GitHub 68](#_Toc149249411)

[4.2.3. Connect Streamlit Share to your GitHub 73](#_Toc149249412)

[4.2.4. Create a New App 74](#_Toc149249413)

[4.2.5. Set Custom Subdomain 75](#_Toc149249414)

# 원본

<https://python-textbook.pythonhumanities.com/05_streamlit/05_01_02_streamlit.html>

<https://github.com/FelipeAdeT/PythonforHumanities/tree/master>

# Introduction to Streamlit

In this part of the textbook, we will learn how to build a custom application in Python with Streamlit and host that application in the cloud.

In Chapter 1, we cover the basic of Streamlit, its utility, the key terminology, and how to display data on a page for a user. Here, readers will gain an understanding of the main widgets Streamlit offers, how to use them and why their useful.

In Chapter 2, we dive into more advanced features of Streamlit to produce better looking and more complex applications by creating data visualizations, controlling the layout of our application, caching data, leveraging custom HTML, and how to design multi-page apps.

In Chapter 3, readers will build upon this knowledge and learn how to develop a custom application and deploy it on Streamlit Share. We will design a database querying application that will be based around Pandas.

## Creating Our First App

### Options for Application Development in Python

In Python, there are several options available to those interested in designing applications. Selecting the right one depends on a few different factors, from customization, deployment (where the application will sit), performance, and speed with which it can be designed. Five years ago, if one wanted to design an application in Python, one needed to use the library Tkinter. This would have a Python-based application that could run locally. The key issue with Tkinter is that it has a steep learning curve, it can only run locally (unless you are willing to package a massive and slow .exe for a simple application), and its aesthetics leaves something to be desired.

커스터 마이징 정도, 배포 위치, 성능, 설계 신속성 등에 따라 다양한 옵션이 존재한다.

If one wanted to deploy an application on the web, there were two options available to researchers until a few years ago: flask and Django. Django has a steep learning curve but allows users to design and build entire websites that are entirely Python based. It also requires a detailed knowledge of servers in order to get an app running in the cloud effectively. Flask a simplified version of Django that does not require knowledge of server side development for deployment.

Django, Flask

Today, there are two options that allow researchers to make applications in minutes. The first is Bokeh. Bokeh allows one to create a Python server based application or write custom JavaScript functions that allow for an application to be compiled and distributed as a single HTML file. Bokeh excels at interactive applications where a user can manipulate a graph that simultaneously changes the output of a table, for example.

For this textbook, we will learn how to design applications with Streamlit. I have chosen Streamlit for this textbook for a few reasons. First, Streamlit is relatively easy to learn. You can have your first application up and running in minutes. Second, it is well-maintained. This means that new features are added regularly. The maintainers of the library listen to their users and consistently provide features that they need. Third, Streamlit was just purchased by Snowflake which means it should continue to exist far into the future. This means that this section of the textbook will not likely go out of date quickly. Fourth, spaCy has pre-built Streamlit components, meaning you can create a spaCy-based application in seconds, not hours. Fifth, Streamlit has a large community and an active forum and Discord channel. This means that if you need to do something in Streamlit, there is likely a tutorial available; if you encounter a bug, there is likely a solution on the forum; if a solution is not on the forum, someone will help you if you ask.

배우기 쉽고, 관리가 잘되고, 프로젝트의 지속성, 커뮤니티

All of these reasons make Streamlit the logical choice for a textbook designed for those with limited coding experience.

코딩에 경험이 적은 사람에게 적합하다.

### Installing Streamlit

In order to begin designing applications with Streamlit, you first need to install it. Like all other Python libraries, you can do so by using pip. Once you execute the code below, you will install Streamlit locally.

pip install streamlit

Remember, if you are installing Streamlit via a Jupyter Notebook, you will need to add a leading !

!pip install streamlit

The entire Streamlit application used for teaching purposes in this book can be found in the main repository under streamlit\_application. To view the application while reading the book, you can run the following command:

streamlit run Home.py

If you have installed Streamlit correctly, you should see a Streamlit application in your browser. In this chapter, we will learn about all the features covered in the demonstration application. In the final section of this chapter, we will learn how to build a basic application involving real-world humanities data.

### Creating a Home Page

To get started with Streamlit, you only need one Python (.py) file. This will be the main page of your application. For our purposes, we will call this Home.py. Once you have created the Python file, you should import Streamlit. You can do so with the following command on the first line of the Python file.

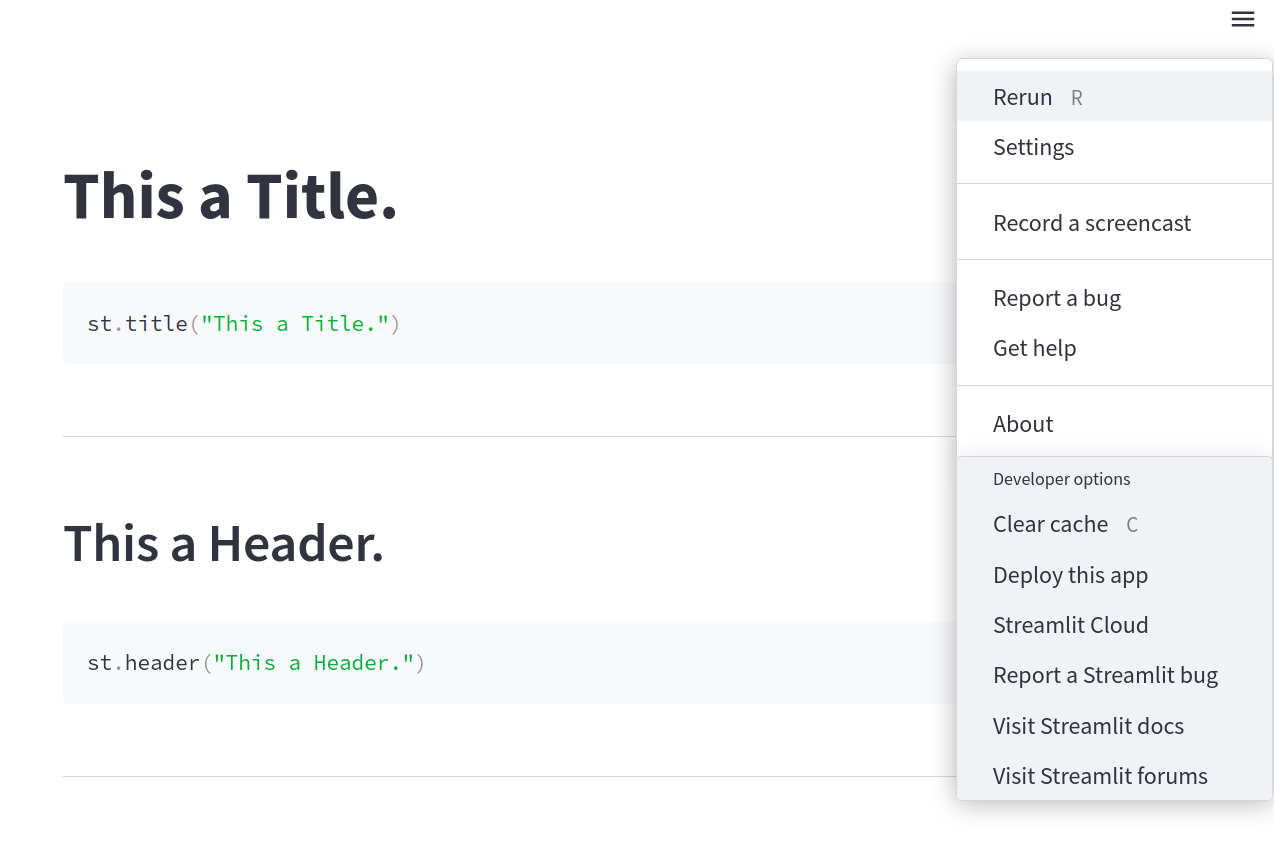
import streamlit as st

Note that we are importing Streamlit specifically as st. This adheres to the official Streamlit documentation. One should stick to this convention as nearly all Streamlit users follow this and it is expected. It will, therefore, make your code easier to understand for others and it will also make it easier for you to debug issues that surface as your code will conform to the expected standards.

Once you have created your file, you can run the following command in your command line:

streamlit run Home.py --server.port=8502

This will create your application on a local server (on your local computer) and populate that server in your browser. Streamlit functions by continuing to run the Python script in the background. This means that as you develop your application, you can see those developments in real time. In the top-right corner, you will see a hamburger icon (three horizontal lines). If you click this button, you will see several options. One of these is Rerun. This will let you rerun your application in real-time.



import streamlit as st

st.title('This a Title')

st.code('''

st.title('This a Title')

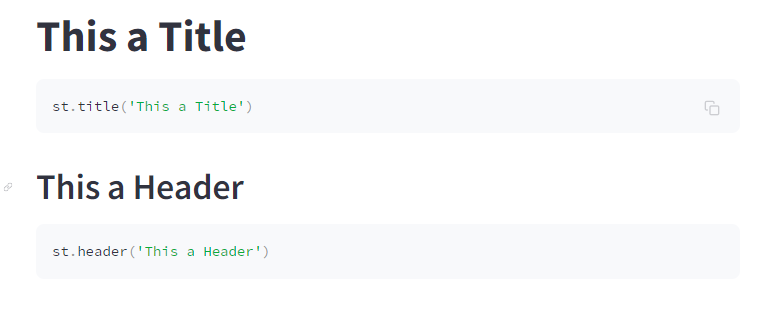
''')

st.header('This a Header')

st.code('''

st.header('This a Header')

''')



## Displaying Data in Streamlit

Streamlit offers numerous ways to display different types of data. In this section, we will be looking at a few of those methods from raw text, to structured markdown, to Pandas DataFrames, and even images.

* Displaying Text to Users
  + st.title
  + st.header
  + st.subheader
  + st.write
  + st.caption
  + st.markdown
* Displaying Python Data Structures
  + Data Structures with st.write()
  + Data Structures with st.json()
* Displaying Tabular Data
  + Tabular Data with st.write()
  + Tabular Data with st.dataframe()
  + Tabular Data with st.table()
  + Tabular Data with st.markdown()
* Displaying Multimedia in Streamlit
  + Images
  + Audio
  + Video

### Displaying Text to Users

Once you have imported streamlit, you can use the Streamlit library to create an application with just a few lines of code. To follow along in the repository, you can view the 01-Displaying Data.py file or the Displaying Data page of the application.

Streamlit offers a number of ways to display information to the users of your application. One of the most common things we must convey to a user is text. This can be a title, it can explain basic information about the application, or it can be to display results from some prompt. Streamlit offers several ways to display information to users.

#### st.title

Every application needs a name. If you want the title of your application to stand out on the page, you can use st.title() to output a title on your application page. This will take one argument, a string which will be your application title. It is entirely optional to have a title. To add the title, you would place the following line into your Python file.

st.title("This a Title.")

Once you have added this to your Python file, you can hit refresh in your Streamlit application, and you should see a title page now appear at the top of your page.

#### st.header

Now that we have a title, we can add some extra layers to our application, rather like a traditional HTML website. We can add headers, for example, with st.header(). Again, this will take a single argument, the text that you wish to display as a header. You can add a header to your application by adding the following line to the same Python file and hitting refresh:

st.header("This a Header.")

Notice that our header appears below our title. This is because Streamlit reads the Python file top-to-bottom as it reruns in the background. If you want an item to appear higher in the application, you must place it earlier in your Python file. Additionally, you can use containers which we will meet later in this chapter.

파이썬 파일에 위치하는 순서대로 화면에 출력이된다.

#### st.subheader

Just like st.header, st.subheader adds a subheading to your application to allow you to have even greater nested structure. If you add the following line to your Python file and hit refresh, you will see a subheading appear.

st.subheader("This a Subheader.")

#### st.write

The most common way to display text to a user is with st.write(). With this command, we can pass a single argument, some sort of data that we want to display. Let’s use this command and display the string This is text. To do this, we would add the following line to our Python file:

st.write("This is text.")

Streamlit’s st.write is quite powerful. As we will see below, it can display data structures, such as lists and dictionaries, as well as entire dataframes, automatically.

#### st.caption

In Streamlit, it may be necessary to caption something. As we will see, images can be captioned separately. To caption something in your application, you will use st.caption(). Again, this will take a single argument, the string that we wish to display as a caption. We can add a caption to our application by adding the following line to our Python file:

st.caption("This a Caption.")

#### st.markdown

Finally, we have st.markdown. Markdown is a type of language that allows you to structure text quickly. Markdown is easy-to-read for both humans and machines and is the standard language used for README pages. Markdown files end with a .md extension.

Streamlit allows users to leverage the power of markdown inside a Streamlit application. This means that we can read in a markdown file stored in the same directory as our application and automatically display that file’s contents in our application. This is really useful when you have pages or sections on a page that require longer section of text or, perhaps, things like lists. These types of long strings can often crowd a Python file. In our demonstration Streamlit application, for example, we can see this in action with the following lines added to our Python file.

with open("./contents/README.md", "r") as f:

markdown\_text = f.read()

st.markdown(markdown\_text, unsafe\_allow\_html =True)

As we will learn later in this book, st.markdown can also be used to display HTML which makes it even more powerful.

HTML 사용을 허용하려면, unsafe\_allow\_html =True 로 설정해줘야 한다.

README.md 파일은 배포된 자료에 contents 폴더 하위에 있음.

#### 전체 코드 및 결과

import streamlit as st

st.title("This a Title.")

st.code('''

st.title("This a Title.")

''')

st.header("This a Header.")

st.code('''

st.header("This a Header.")

''')

st.subheader("This a Subheader.")

st.code('''

st.subheader("This a Subheader.")

''')

st.write("This is text.")

st.code('''

st.write("This is text.")

''')

st.caption("This a Caption.")

st.code('''

st.caption("This a Caption.")

''')

with open("./contents/README.md", "r") as f:

markdown\_text = f.read()

st.markdown(markdown\_text, unsafe\_allow\_html =True)

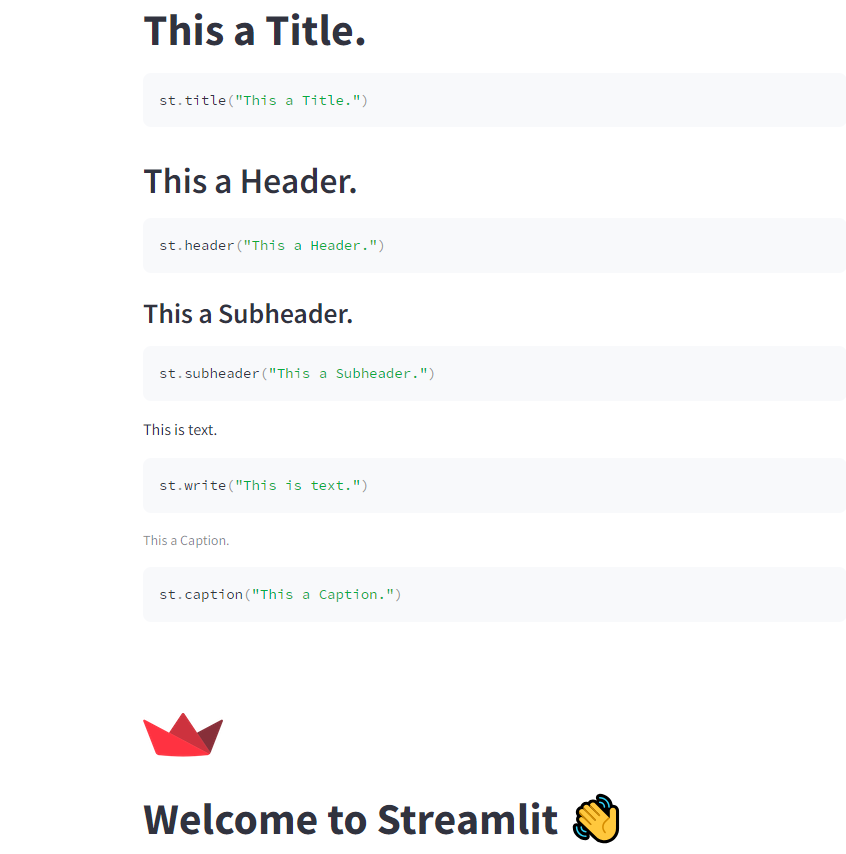
st.code('''

with open("./contents/README.md", "r") as f:

markdown\_text = f.read()

st.markdown(markdown\_text, unsafe\_allow\_html =True)

''')



### Displaying Python Data Structures

It is often necessary to display raw data inside an application either during the development phase of the application or in production (when users will engage with the app). Streamlit has two different ways to display data with the same results: st.write() or st.json(). On the surface, they both do the same thing, but st.json allows you to set an expanded keyword argument to True or False. This allows you to control if the data appears in its entirety in the application or as a expandable option within the application.

#### Data Structures with st.write()

Let’s first test this out with st.write(). We can use the following two lines to create a dictionary called names and then st.write() to display that data.

import streamlit as st

names = {"people": ["Tom", "Mary", "Fred", "Stephanie"]}

st.write(names)

st.code('''

names = {"people": ["Tom", "Mary", "Fred", "Stephanie"]}

st.write(names)

''')

The output will look like this in your application:



#### Data Structures with st.json()

For our second option, we can use st.json, but note that we are able to specify expanded here as a keyword argument.

import streamlit as st

names = {"people": ["Tom", "Mary", "Fred", "Stephanie"]}

st.json(names, expanded=False)

st.code('''

names = {"people": ["Tom", "Mary", "Fred", "Stephanie"]}

st.json(names, expanded=False)

''')

names = {"people": ["Tom", "Mary", "Fred", "Stephanie"]}

st.json(names, expanded=True)

st.code('''

names = {"people": ["Tom", "Mary", "Fred", "Stephanie"]}

st.json(names, expanded=False)

''')

The output will look like this in your application:



### Displaying Tabular Data

There are four ways to display tabular data within Streamlit.

* st.write() (defaults to st.dataframe())
* st.dataframe()
* st.table()
* st.markdown()

While on the surface, these may all seem to display the same data, understanding what each does is important so that you can have your application do precisely what you wish.

#### Tabular Data with st.write()

If you are trying to test an application quickly and just want to display tabular data without any extra customization, then st.write() is perfectly suitable. If Streamlit detects a Pandas DataFrame as the object that is being passed to st.write(), it will automatically output that data via st.dataframe().

st.write(df)

#### Tabular Data with st.dataframe()

If you want to have more control over how your tabular data is displayed in your application, you may want to use st.dataframe instead. By using st.dataframe, you can control the width and height of the displayed dataframe.

st.dataframe(df, height=200)

The output will look like this in your application:



With both st.write() and st.dataframe, users will be given a Streamlit dataframe display. This means that they can highlight certain parts of the dataframe, expand cells to read longer text, and sort the data. In other words, the dataframe is an entirely interactive display widget.

#### Tabular Data with st.table()

One of the downsides the st.dataframe display is that the interactivity comes at the cost of aesthetics. If you are working with humanities data, you may have a lot of text in your tables. That text can be difficult for viewers to read in the standard st.dataframe output. In these situations, st.table may be more appropriate.

미관상 보기가 좋다.

st.table(df)

#### Tabular Data with st.markdown()

A key limitation of both the st.table() and st.dataframe is that they do not offer a way to display images. With markdown, we can easily display images within our tables. This, however, comes at the cost of not being able to sort the output. We will learn how to do this later when we work with custom HTML in our Streamlit application. For now, understand that you can convert a Pandas DataFrame to markdown by using the to\_markdown() method.

이미지 출력 지원, 데이터프레임을 마크다운 코드로 변환(to\_markdown())

st.markdown(df.to\_markdown())

전체 코드 및 결과

import streamlit as st

import pandas as pd

# 예제 데이터 생성

data = {

'이름': ['홍길동', '김철수', '이영희', '박영수'],

'나이': [30, 25, 28, 32],

'성별': ['남성', '남성', '여성', '남성']

}

# 데이터 프레임 생성

df = pd.DataFrame(data)

# 데이터 프레임 출력

st.write(df)

st.dataframe(df, height=200)

st.table(df)

st.markdown(df.to\_markdown())

st.code('''

st.write(df)

st.dataframe(df, height=200)

st.table(df)

st.markdown(df.to\_markdown())

''')



### Displaying Multimedia in Streamlit

Streamlit also offers the ability to easily add multimedia into your application. For all three types of media (images, audio, and video), Streamlit allows you to place the media in the app in four ways:

* file path
* from a url
* from a NumPy Array
* from bytes

Each of these has its own uses. If the media you have is available locally within the app (such as logos and design elements), from file path usually makes the most sense. If your data sits on a server on the Web, then url is the right choice. These will be images that do not sit within the code of your application or in the local directory. When you have a user input media data into the app (via file upload), you will want to load the data via NumPy Array or Bytes. We will see these last two in action during the final chapter of this section when we apply Streamlit to develop real digital humanities applications.

#### Images

If we are working with images, we can load a local image with the following line:

st.image(path\_to\_image)

#### Audio

For audio, we would use the following line

st.audio(path\_to\_image)

#### Video

For video we would use the following line:

st.video(path\_to\_image)

## Streamlit Input Widgets

Often when you are designing your application, you will need a way to allow the user to interact with the app. When this occurs, you want a way to do something that that user input. There are many ways that we can allow a user to interact with our data in Streamlit. In this section, we will cover five of the main categories:

* text input
* numerical input
* date input
* boolean input
* selection input

Cheatsheet for Widgets in Streamlit

|  |  |  |
| --- | --- | --- |
| **Widget** | **Return** | **Description** |
| text\_input | string | A small text region |
| text\_area | string | A large text region |
| number\_input | integer or floa | A typed number input |
| slider | integer or floa | A slider for number input |
| date\_input | timeseries | A calendar for selecting a date |
| time\_input | timeseries | A dropdown menu for selecting a time |
| checkbox | Boolean | A checkbox for marking something as true or false |
| button | Boolean | A button for triggering an event |
| radio | string | A selection for a single option |
| selectbox | string | A selection for a single option (dropdown menu) |
| multiselect | list | A selection for multiple options (dropdown menu) |

* Text Input Widgets
  + st.text\_input()
  + st.text\_area()
* Numerical Input Widgets
  + st.number\_input()
  + st.slider()
* Date and Time Input Widgets
  + st.date\_input()
  + st.time\_input()
* Boolean Input Widgets
  + st.checkbox()
  + st.button()
* Selection Widgets
  + st.radio()
  + st.selectbox()
  + st.multiselect()

### Text Input Widgets

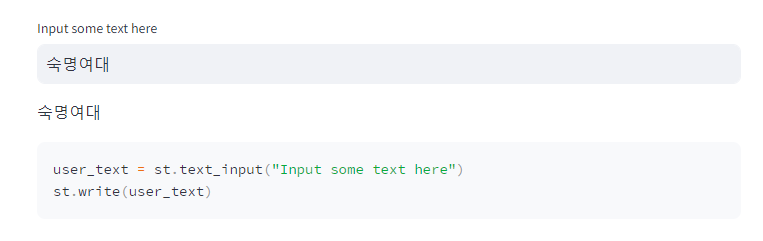
Streamlit offers two ways to allow users to input textual data into an application. You can either use st.text\_input() or st.text\_area(). Both essentially do precisely the same thing, that is, return a string from the user; but each should be used in specific situations. Streamlit’s st.text\_input() is designed for shorter text (such as names, queries, etc.), while st.text\_area() should be used for longer string input data, such as text that can be processed via a spaCy pipeline.

#### st.text\_input()

사용자로부터 짧은 데이터를 입력 받을 때 사용

user\_text = st.text\_input("Input some text here")

st.write(user\_text)



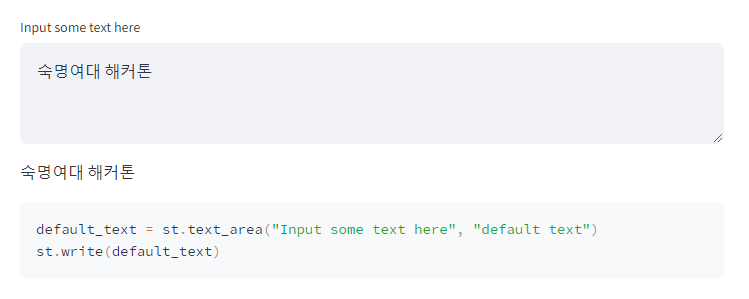
#### st.text\_area()

To create a text area style input, you can use the precise same code, but replace text\_input with text\_area. Both of these classes also let us pass an additional argument for some default text as the second parameter. We can add some default text that will prepopulate the text field with a predetermined string.

default\_text = st.text\_area("Input some text here", "default text")

st.write(default\_text)

첫 번째 파라미터는 라벨, 두 번째는 기본값



### Numerical Input Widgets

While you could let a user input numerical data inside of st.text\_input()or st.text\_area(), this really would be inappropriate. Both of these input options return strings. This means that if you gave the user the ability to input numerical data, you would have to convert it to either an integer or float. In addition to this, the text input widgets do not offer any special keyword arguments that you can pass to the widget that are specific to numerical data, such as minimum value and maximum value.

It is far better in these scenarios to use one of two options for numerical input.

#### st.number\_input()

The first option is st.number\_input(). This feature lets a user input a numerical data that they can tick up or down via a minus or plus sign in the widget. The widget will return an integer or float, depending on how you structure the widget.

Users can also manually type in a specific number. Another feature of this widget is the ability to specify a minimum value (with the min\_value argument) and maximum value (with the max\_value argument). You can also give the user a default value by setting the value argument to a specific number. Finally, you can even provide a step argument which will step up every n-numbers, so a step of 2 would increase the value by two each time the user clicks the plus symbol in the widget.

user\_number = st.number\_input("Input Number",

min\_value=1,

max\_value=10,

value=5,

step=1)

st.write(user\_number)



#### st.slider()

Another way to let a user input data is via the st.slider() widget. This widget will also return an integer or a float, depending on if your values are in decimal form.

slider\_number = st.slider("Select your Number",

min\_value=1,

max\_value=10,

value=5,

step=1)

st.write(slider\_number)



### Date and Time Input Widgets

Working with dates and times is essential in a lot of applications and Streamlit has two widgets for receiving time-series data. Both will require the use of the built-in library datetime, so be sure to import this if you intend to work time time-series data in your application.

#### st.date\_input()

The first widget is st.date\_input(), this allows you to receive a date object which will allow you to structure robust logic, such as finding all data that fall between a start date and end date. When creating the widget, you can simply use the defaults, but if you expect all your data to fall between two dates, it may be wise to set minimum and maximum values. You can do this via the datetime library date method. You should ensure that all values in the st.date\_input() widget conform to the following format:

user\_date = st.date\_input("Select your Date",

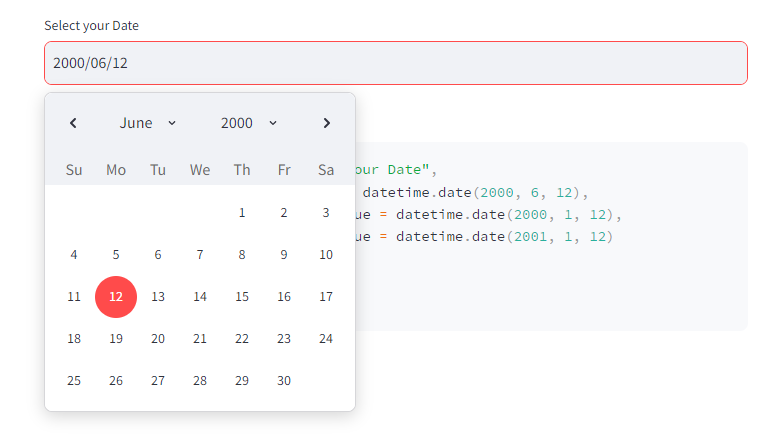
value = datetime.date(2000, 6, 12),

min\_value = datetime.date(2000, 1, 12),

max\_value = datetime.date(2001, 1, 12)

)

st.write(user\_date)



#### st.time\_input()

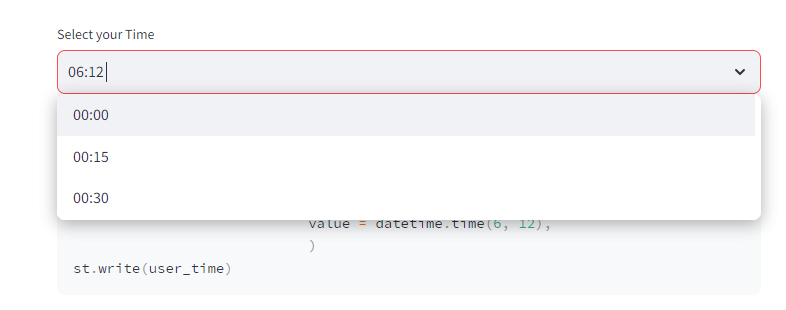
You can do precisely the same thing with time via the st.time\_input() widget. With time, however, we cannot set min or max values. Also, unlike the st.date\_input(), the st.time\_input() will use the time method from the datetime library.

user\_time = st.time\_input("Select your Time",

value = datetime.time(6, 12),

)

st.write(user\_time)



### Boolean Input Widgets

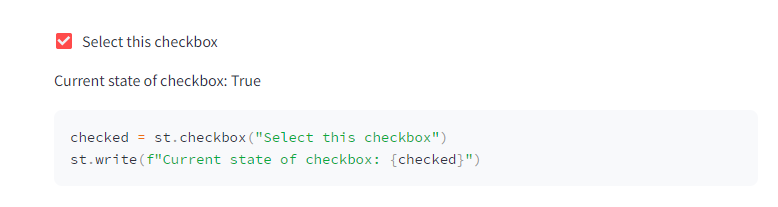
Another important feature for controlling the logic of your application is understanding Boolean (True or False) values from a user input. In Streamlit, we have two ways of using Boolean inputs to control the logic and they both function a bit differently.

#### st.checkbox()

The first is st.checkbox(). This allows us to create a checkbox. Its state can be constantly changed. So a box can be checked or unchecked. As a user changes the state of the checkbox, the Boolean output from the widget will change. We can create a checkbox widget and write out the results with the following lines of code.

checked = st.checkbox("Select this checkbox")

st.write(f"Current state of checkbox: {checked}")

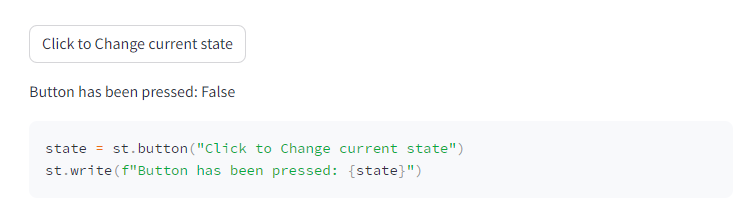


#### st.button()

Unlike the st.checkbox() widget, the st.button() widget will have a continuous state. This means that once the button is clicked, its Boolean output value will forever change unless you specifically change it in your script. In other words, the button’s state at the start of the application is False, but once a user clicks the button, that state will be True continuously. This is important because it means that the click of a button can trigger a one-time event, such as draw a map or run a machine learning model.

state = st.button("Click to Change current state")

st.write(f"Button has been pressed: {state}")



### Selection Widgets

The final collection of important widgets in Streamlit are the selection widgets. These allow you to give users a set of options to choose from. There are three types of selection widgets.

#### st.radio()

The first is st.radio(). This widget allows you to give the user the ability to second one item from a list of options. Only one option can be selected by the user.

options = ["Red", "Blue", "Yellow"]

radio\_selection = st.radio("Select Color", options)

st.write(f"Color selected is {radio\_selection}")



The st.radio() widget will return a string of the selected option.

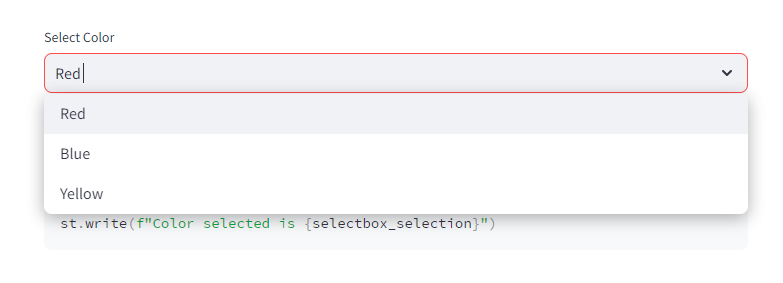
#### st.selectbox()

The same logic holds true for the st.selectbox() widget. The only difference between this and the st.radio() widget is the aesthetic way the options are presented. A selectbox is often more appropriate if you are presenting a user with a larger selection of options, which would be clunky with radio buttons. This will return a string of the selected item.

options = ["Red", "Blue", "Yellow"]

selectbox\_selection = st.selectbox("Select Color", options)

st.write(f"Color selected is {selectbox\_selection}")



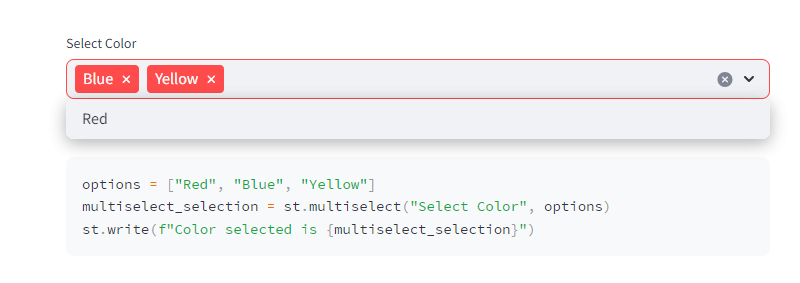
#### st.multiselect()

The final selection widget is the st.multiselect() widget which allows a user to select multiple items from a selectbox. This will return a list of the selected items in the options.

options = ["Red", "Blue", "Yellow"]

multiselect\_selection = st.multiselect("Select Color", options)

st.write(f"Color selected is {multiselect\_selection}")



#### 전체 코드 및 결과

import streamlit as st

import datetime

user\_text = st.text\_input("Input some text here")

st.write(user\_text)

default\_text = st.text\_area("Input some text here", "default text")

st.write(default\_text)

user\_number = st.number\_input("Input Number",

min\_value=1,

max\_value=10,

value=5,

step=1)

st.write(user\_number)

slider\_number = st.slider("Select your Number",

min\_value=1,

max\_value=10,

value=5,

step=1)

st.write(slider\_number)

user\_date = st.date\_input("Select your Date",

value = datetime.date(2000, 6, 12),

min\_value = datetime.date(2000, 1, 12),

max\_value = datetime.date(2001, 1, 12)

)

st.write(user\_date)

user\_time = st.time\_input("Select your Time",

value = datetime.time(6, 12),

)

st.write(user\_time)

checked = st.checkbox("Select this checkbox")

st.write(f"Current state of checkbox: {checked}")

state = st.button("Click to Change current state")

st.write(f"Button has been pressed: {state}")

options = ["Red", "Blue", "Yellow"]

radio\_selection = st.radio("Select Color", options)

st.write(f"Color selected is {radio\_selection}")

options = ["Red", "Blue", "Yellow"]

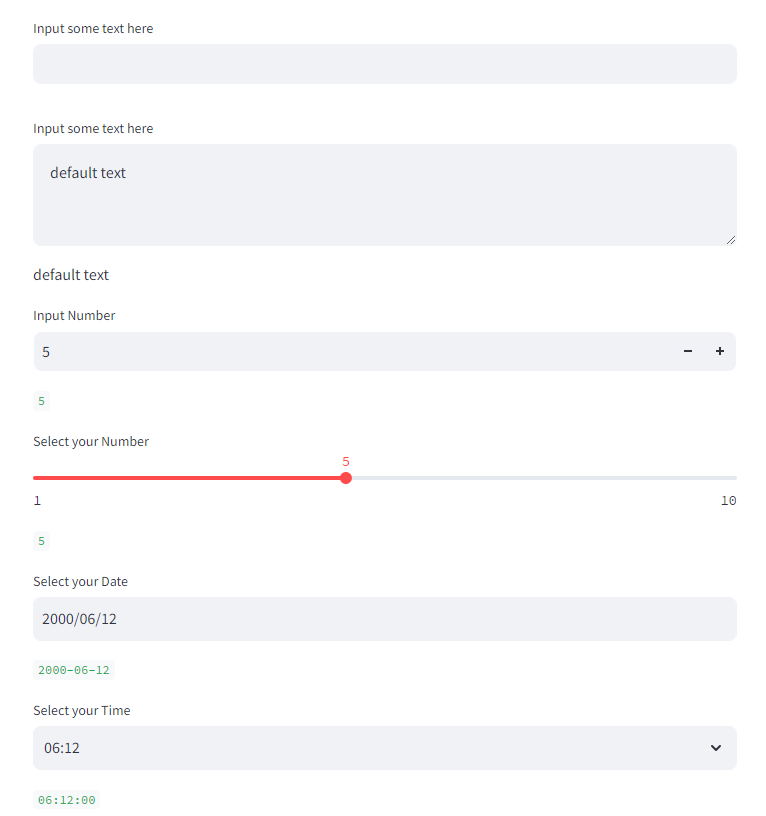
selectbox\_selection = st.selectbox("Select Color", options)

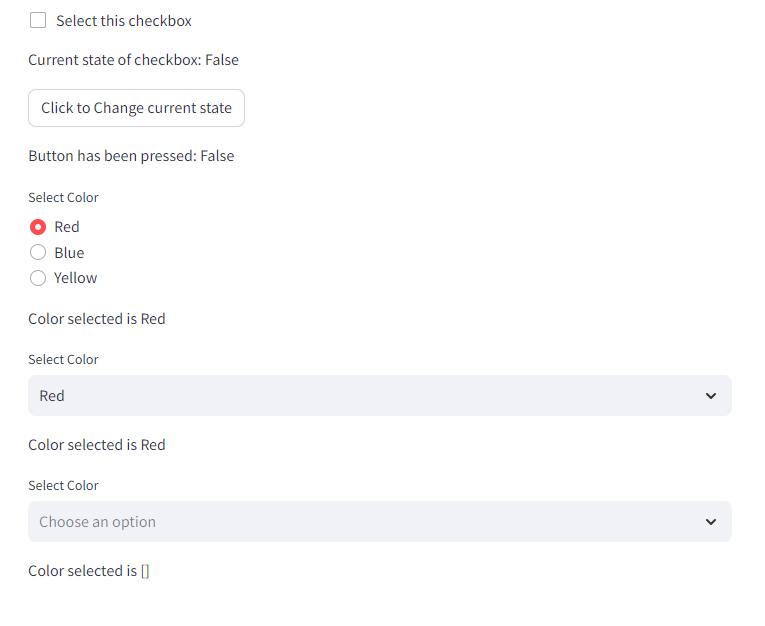
st.write(f"Color selected is {selectbox\_selection}")

options = ["Red", "Blue", "Yellow"]

multiselect\_selection = st.multiselect("Select Color", options)

st.write(f"Color selected is {multiselect\_selection}")





# Advanced Streamlit Features

## Data Visualization

When working with large quantities of data, it can often be difficult to present that data in a way that makes sense to non-experts. In these situations, we often rely on some form of chart to represent our data. Creating quality graphs in Python requires a lot of practice, but quality charts can be produced with Matplotlib, Altair, PyDeck, PyVis, Plotly, and Bokeh. Each has their own strengths. In this section, we will not go into these libraries, rather we will focus on how to present these different types of graphs in Streamlit.

In Streamlit, we can leverage these libraries to produce visually appealing charts in just a few lines of code. We will focus on three types of graphs: Basic Plot Graphs, Map Graphs, and Network Graphs.

* Metrics
* Plotting Basic Graphs with Streamlit
  + Line Charts with st.line\_chart()
  + Bar Charts with st.bar\_chart()
  + Area Charts with st.area\_chart()
* Map Charts
  + Creating Maps with st.map()
  + Third-Party Maps - An Example with PyDeck

### Metrics

Before we address plots, we should spend a brief moment and think about how we display raw numerical quantitative data. We could display the length of a dataframe or the word count of some text st.write(). Again, while this would be quick to do, it would not allow you to display other important information, such as how that number has changed from a previous state. Nor would it allow you to easily represent the change in a positive or negative direction without complex JavaScript and HTML. In these scenarios, we would want to use st.metric().

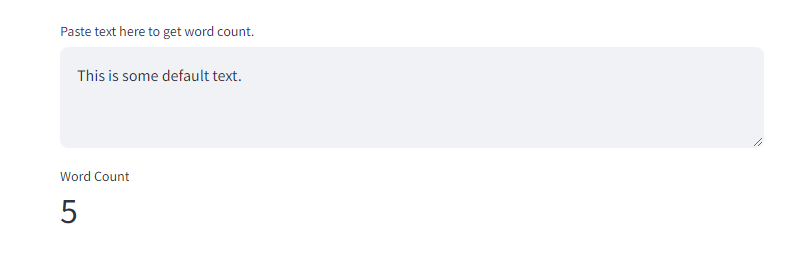
값의 변화 정보를 표시 할 수 있다.

By default, the metric will display a numerical output of some sort. This number could also be a string representation of a number, e.g. temperature. Let’s say, we wanted to create a simple application where a user could copy-and-paste some text into a st.text\_area() field. The app would split up the words at every white space and then provide the user with the total word count.

text = st.text\_area("Paste text here to get word count.", "This is some default text.")

word\_count = len(text.split())

st.metric("Word Count", word\_count)



Since we are using the st.metric() widget, however, we can also pass in a keyword argument that displays the degree to which the metric changed from the previous state. To do this, we will need to leverage the Streamlit Session State, which we will meet later in this chapter. This allows us to store a variable across different runs of the application. For now, we can ignore this bit of the code below and focus on the third argument that we passed to metric, change. This will display a change feature in the widget that will show the up or down trend of the change in green and red color, respectively.

if "prev\_word\_count" not in st.session\_state:

st.session\_state["prev\_word\_count"] = 5

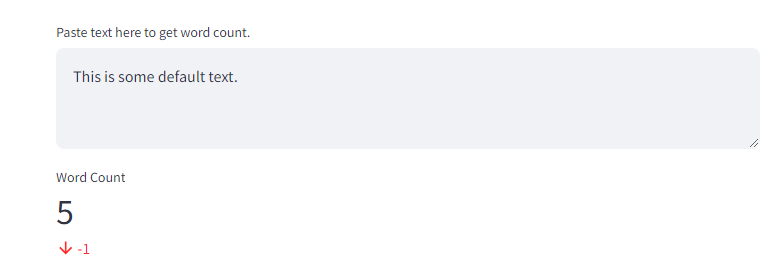
text = st.text\_area("Paste text here to get word count.", "This is some default text.")

word\_count = len(text.split())

change = word\_count-st.session\_state.prev\_word\_count

st.metric("Word Count", word\_count, change)

st.session\_state.prev\_word\_count = word\_count



Metric is a useful feature that allows us to create apps that display numerical data in easy-to-understand ways. But in other situations, a single qualitative number may not be appropriate. Here is where charts come in handy.

### Plotting Basic Graphs with Streamlit

We can plot basic graphs in Streamlit by passing a Pandas dataframe to different chart widgets in Streamlit. The first basic plot we can create is a line chart which we can create with the Streamlit widget st.line\_chart(). We will be working with the Titanic dataset here that we first met in Part Two of this textbook. To prepare the data for visualization, we need to modify it a bit and group everything by the specific value that we want to plot. In our case, we want to visualize the number of survivors for different age groups on the Titanic. We can prepare our dataframe with the code below.

import streamlit as st

import pandas as pd

# 데이터 준비

df = pd.read\_csv("data/titanic.csv")

df = df[["Age", "Survived"]]

chart\_df = df.groupby(["Age"]).sum()

chart\_df["Age"] = chart\_df.index

st.dataframe(df)

st.dataframe(chart\_df)



#### Line Charts with st.line\_chart()

Once we have created our new chart\_df, we can pass it to st.line\_chart(). Here, we will pass the entire dataframe as the first argument and specify our x axis and y axis on the graph. In our case, we want to view the Age column on the x axis and the Survived column on the y axis.

import streamlit as st

import pandas as pd

# 데이터 준비

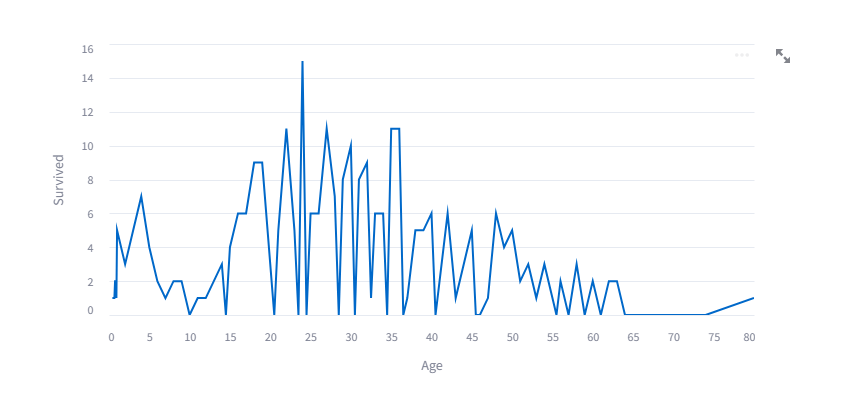
df = pd.read\_csv("data/titanic.csv")

df = df[["Age", "Survived"]]

chart\_df = df.groupby(["Age"]).sum()

chart\_df["Age"] = chart\_df.index

st.line\_chart(chart\_df, x="Age", y=["Survived"])



#### Bar Charts with st.bar\_chart()

Likewise, we can present this same data as a bar\_chart with the widget st.bar\_chart(). This will take the same arguments as above.

import streamlit as st

import pandas as pd

# 데이터 준비

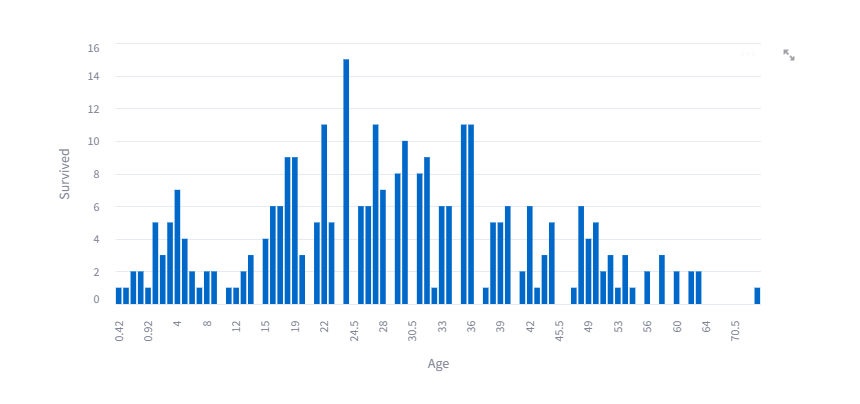
df = pd.read\_csv("data/titanic.csv")

df = df[["Age", "Survived"]]

chart\_df = df.groupby(["Age"]).sum()

chart\_df["Age"] = chart\_df.index

st.bar\_chart(chart\_df, x="Age", y=["Survived"])



#### Area Charts with st.area\_chart()

And finally we can also use the same arguments to create an area chart with the st.area\_char() widget.

import streamlit as st

import pandas as pd

# 데이터 준비

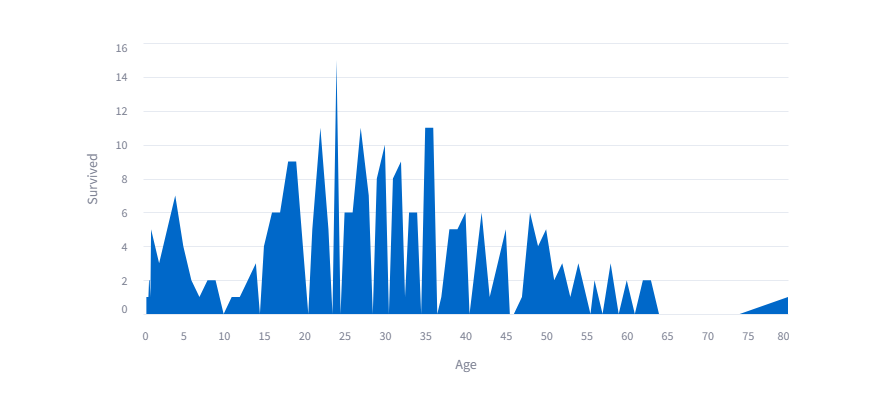
df = pd.read\_csv("data/titanic.csv")

df = df[["Age", "Survived"]]

chart\_df = df.groupby(["Age"]).sum()

chart\_df["Age"] = chart\_df.index

st.area\_chart(chart\_df, x="Age", y=["Survived"])



### Map Charts

위치 정보가 있는 데이터를 이용해서 지도 위 시각화가 가능하다.

#### Creating Maps with st.map()

#### Third-Party Maps - An Example with PyDeck

## Layout Design

* Layout Widgets
  + Sidebar
  + Columns
  + Expander
  + Container
  + Tabs
  + Empty

### Layout Widgets

Controlling the layout of an application is essential from both an aesthetic and programmatic perspective. In Streamlit, we have a number of ways to control our app’s layout. Streamlit offers six different ways of controling the layout of your application through widgets. Further customization is also possible by controling the page configuration of your application as well as custom HTML. Here, we will focus on the first two; we will treat custom HTML later in this chapter.

미적인 관점과 프로그램 관점에서 레이아웃은 필수이다.

#### Sidebar

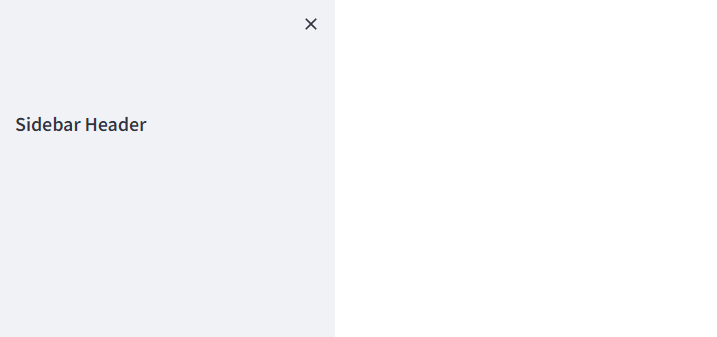
The sidebar is one of the more useful layout features in Streamlit. If you are creating a multi-page app, you will have a sidebar by default. If, however, you are not using a multi-page app, you simply need to do something within the st.sidebar() widget and the sidebar will automatically appear for you. We can access the sidebar widget at any place in our Python file by writing st.sidebar.[widget]()

If we were to use the following code in our Python file, for example, we would immediately have a sidebar with a header entityled Sidebar Header.

import streamlit as st

import pandas as pd

st.sidebar.header("Sidebar Header")



Notice on the left-hand side of the application, we have our header appear. We can populate any widget we desire into the sidebar by calling st.sidebar and then following this up with .[widget].

#### Columns

We can also control the horizontal layout of our application with Streamlit’s st.columns() widget. This will take a single mandatory argument, an integer that corresponds to the number of columns you wish to create. This will return a list of column widgets that you can write to. As with the sidebar, to write to the column widget, you will use its variable name proceeded by .[widget (write, header, etc.)]

import streamlit as st

import pandas as pd

st.sidebar.header("Sidebar Header")

st.header("Columns")

cols = st.columns(2)

cols[0].write("Column 1")

cols[1].write("Column 2")



#### Expander

In some cases, we will want to allow a user to view a large amount of data, but this will detract from the application and put some of the main features very low on the main page of the app. In these, cases we want a user to be able to expand certain aspects of the application when they wish and compress them when they are finished using them. Streamlit comes built with this feature in the st.expander() widget.

Once we create an expander in our application, we can then insert any other Streamlit we wish inside of it, just like columns and the sidebar. In the code below, we are creating a simple expander and then writing into it.

import streamlit as st

import pandas as pd

st.sidebar.header("Sidebar Header")

st.header("Columns")

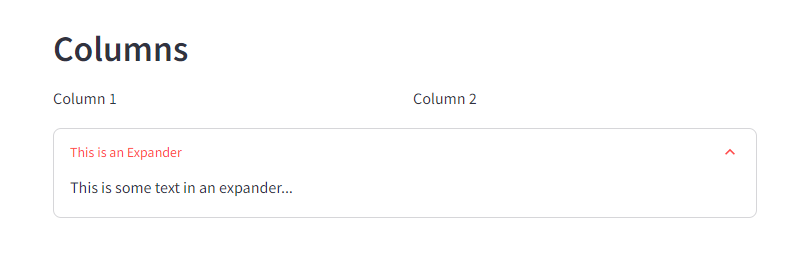
cols = st.columns(2)

cols[0].write("Column 1")

cols[1].write("Column 2")

expander = st.expander("This is an Expander")

expander.write("This is some text in an expander...")



Notice how we see a new widget on our page. In the top-right corner, you can press the carrot and the expander will reveal the contents.

#### Container

We also have the ability to create containers in our application with the st.container() widget. Remember, Streamlit widgets are populated in the order that Streamlit reads your Python file. This means that if your user performs an action and you want to display that data later in your file, then that data will populate at the bottom of your application. This is not ideal. In these scenarios, you need a way to populate a result higher in your application’s page. You can do this with the container. The container will sit in that precise position and it can then be populated at a later time.

프로그래밍적으로 화면의 출력 순서를 이 위젯을 이용해서 변경이 가능하다.

#### Tabs

In Streamlit, we can also make tabs so that a single page can host several different pages within it. This is useful if you are working with multiple datasets or collections of items that need to be displayed individually, but there is not enough space in your app to display all data sequentially. We create tabs in Streamlit via the st.tabs() widget. This will take a single argument which will be a list of tab names.

한 페이지에 다수개의 페이지를 가질 수 있다.

In the sample code below, we create the tabs on line 1 and then iterate over them to populate each tab with a unique message via st.write().

import streamlit as st

import pandas as pd

st.sidebar.header("Sidebar Header")

st.header("Columns")

cols = st.columns(2)

cols[0].write("Column 1")

cols[1].write("Column 2")

expander = st.expander("This is an Expander")

expander.write("This is some text in an expander...")

st.header("Container")

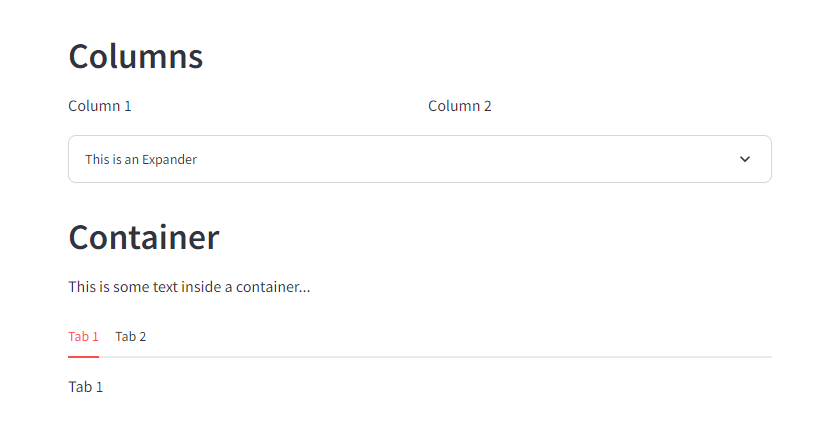
container = st.container()

container.write("This is some text inside a container...")

tabs = st.tabs(["Tab 1", "Tab 2"])

for i, tab in enumerate(tabs):

tabs[i].write(f"Tab {i+1}")



#### Empty

The final layout widget in Streamlit is st.empty(). This takes no arguments. It sits on the page, rather like the st.container() widget, but unlike st.container(), st.empty() will only display the most recent item sent to it. This is useful in rather niche scenarios when you want to display data individually. You can use the following code to iterate over a list of names. Note in the image below that only the last name on the list appears. This is because it was the final item in the list.

import streamlit as st

import pandas as pd

st.sidebar.header("Sidebar Header")

st.header("Columns")

cols = st.columns(2)

cols[0].write("Column 1")

cols[1].write("Column 2")

expander = st.expander("This is an Expander")

expander.write("This is some text in an expander...")

st.header("Container")

container = st.container()

container.write("This is some text inside a container...")

tabs = st.tabs(["Tab 1", "Tab 2"])

for i, tab in enumerate(tabs):

tabs[i].write(f"Tab {i+1}")

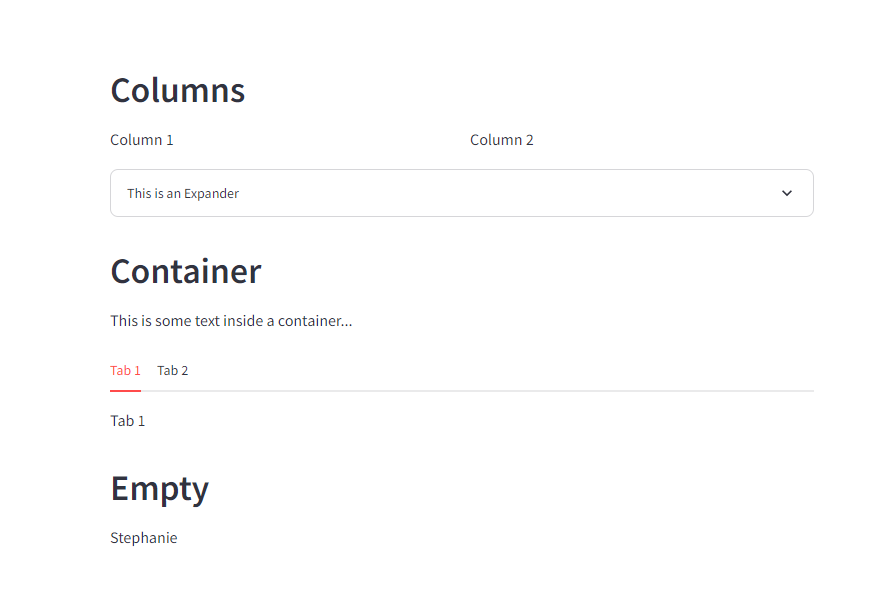
st.header("Empty")

empty = st.empty()

items = ["Tom", "Fred", "Stephanie"]

for item in items:

empty.write(item)



## Streamlit Cache and Session States

Being able to make more modular, custom, and dynamic applications means getting to know how to leverage advanced features in Streamlit when it comes to working with data stored in memory. In Streamlit, we have two ways to store data in memory, either by caching the data with @st.cache() or with the st.session\_state.

### Caching Data with st.cache

When working with large data-driven projects, run time will become an issue with Streamlit. This is because Streamlit reruns the Python file each time something changes in the application. With large datasets, this means that each time a user does anything within your application, Streamlit will need to reload all the data. For this reason, it is essential to know how to store large datasets (or models) in cache so that Streamlit does not need to reload large memory-intensive data or models each item it reruns.

사용자의 액션이 있을 때 마다 전체 코드를 재실행한다.

We can cache our data with an @st.cache() above a function that loads the data. If we wanted to load our Titanic dataset and store it in memory, therefore, we would use the following code snippet.

@st.cache()

def load\_df():

df = pd.read\_csv("./data/titanic.csv")

return df

This is precisely the code that we will walk through when we create our first application in Streamlit later in this part of the textbook.

## Storing Data with st.session\_state

Aside from storing large data with cache, we can also store previous states of data with the st.session\_state. The Streamlit Session State gives greater flexibility to an application. It functions as a dictionary that stores data that remains the same during any given state of the app. This means that if your app is rerun by the user because they interacted with the application, then the variable stored in the session state would remain the same.

This is essential for more complex data-driven applications. Let’s consider the simple example that we saw earlier in this chapter when we examined the st.metric() widget.

import streamlit as st

if "prev\_word\_count" not in st.session\_state:

st.session\_state["prev\_word\_count"] = 5

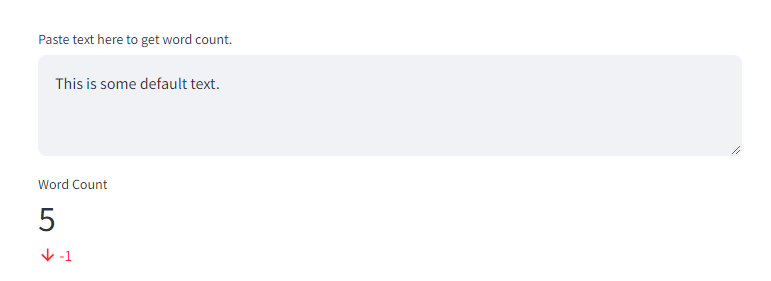
text = st.text\_area("Paste text here to get word count.", "This is some default text.")

word\_count = len(text.split())

change = word\_count - st.session\_state.prev\_word\_count

st.metric("Word Count", word\_count, change)

st.session\_state.prev\_word\_count = word\_count



In this sample, we start off with a conditional:

if "prev\_word\_count" not in st.session\_state:

This line looks to see if a variable name that we want to use is stored in our session state. If it is not stored there, then we want to create that new key. We do that with the following line:

st.session\_state["prev\_word\_count"] = 5

Here we are setting the prev\_word\_count key to 5.

Next, we give the user the ability to input some text for which they wish to receive a word count. In order for the metric to know if the new metric is higher or lower than the previous one, we must store the previous text’s total word count. To do this, we access the previous session state in the final line of the snippet below.

text = st.text\_area("Paste text here to get word count.", "This is some default text.")

word\_count = len(text.split())

change = word\_count-st.session\_state.prev\_word\_count

Once we have populated those results, we then can update the st.session\_state.previous\_word\_count value to the new value. This allows us to always know the state of the previous word count, so that when we display the change value, we know precisely how much our metric has changed.

st.metric("Word Count", word\_count, change)

st.session\_state.prev\_word\_count = word\_count

## Custom HTML

Adding custom HTML in Streamlit allows you to develop more customized applications that fit your need. While it is not always necessary, it is useful to understand how to embed HTML within an application.

Custom HTML can be added via st.markdown(). In order for your HTML to appear on the page, however, you must pass a keyword argument unsafe\_allow\_html=True. This allows the HTML to be rendered. Let’s look at a basic example where we want to display text with a background color of yellow. We can do this by wrapping our text in an a tag in HTML and setting the style’s background color to yellow.

html = """

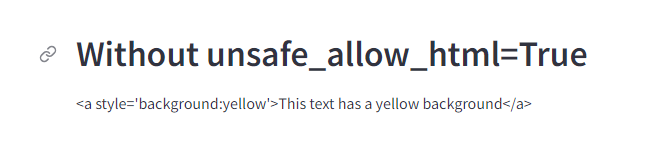
<a style='background:yellow'>This text has a yellow background</a>

"""

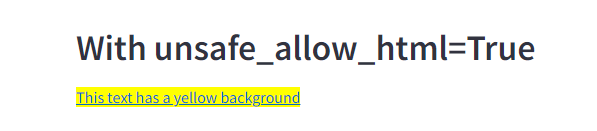
st.header("Without unsafe\_allow\_html=True")

st.markdown(html)

If we do not set unsafe\_allow\_html to True, then our result will look like this:



If we do set it to True, then our result will look like this:



## Multi-Page Applications

Finally, Streamlit allows users to design applications that have multiple pages. In order to design a multi-page application, you must have a pages subfolder in your main directory. All Python files inside this directory will be considered to be pages for your application. If you wish to have your pages appear in a special order, you can do so by naming each page 01-page name and 02-page\_name. Streamlit will automatically remove the leading numbers and dash.

* 다중 페이지 애플리케이션을 디자인
* pages 폴더 생성 필요
* pages 폴더에 있는 파일은 Python 파일은 애플리케이션의 페이지로 간주
* 페이지가 특별한 순서로 표시되도록 하려면 각 페이지의 이름을 01-page name, 02-page\_name으로 지정
* Streamlit은 앞 번호와 대시를 자동으로 제거

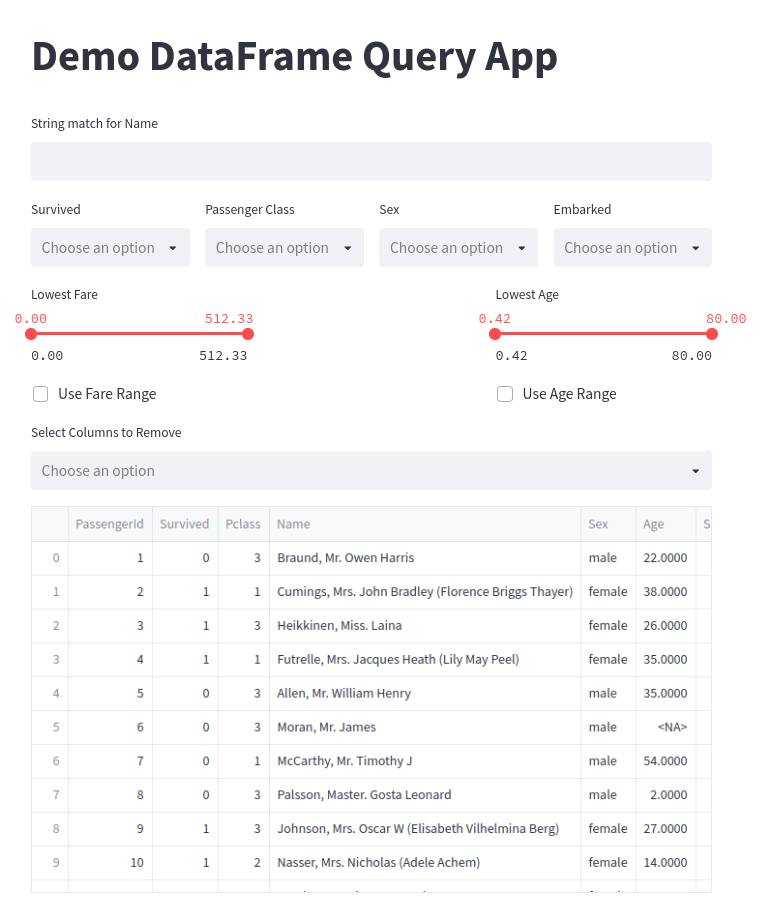
# Building a Database Query Application

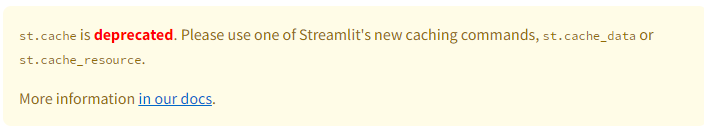
Now that you know the basics of Streamlit and some of its more advanced features, then it comes time to put that knowledge to practice. In this chapter, we will design a Streamlit application from scratch. It will be built around the Titanic dataset that we used in the Pandas portion of this textbook. We will not only design this application, we will also put it in the cloud so that users can access it.

In Section 1 of this chapter we will walk through the basics of designing the application in Python. In Section 2, we will then focus on getting our application running in the cloud. The purpose of this chapter is to give you hands-on experience with designing an application as well as provide a template for you to use in a future project that requires querying a Pandas dataframe in Streamlit.

## Building a Database Query Application

Throughout this section, we will be developing an application in Streamlit that looks like this:





We will be working with the following code:

import streamlit as st

import pandas as pd

# Cache our data

@st.cache\_data()

def load\_df():

df = pd.read\_csv("./data/titanic.csv")

# 생존 여부

survival\_options = df.Survived.unique()

# 객실

p\_class\_options = df.Pclass.unique()

# 성별

sex\_options = df.Sex.unique()

# 출발

embark\_options = df.Embarked.unique()

# 요금

min\_fare = df.Fare.min()

max\_fare = df.Fare.max()

# 나이

min\_age = df.Age.min()

max\_age = df.Age.max()

return df, survival\_options, p\_class\_options, sex\_options, embark\_options, min\_fare, max\_fare, min\_age, max\_age

def check\_rows(column, options):

return res.loc[res[column].isin(options)]

st.title("Demo DataFrame Query App")

df, survival\_options, p\_class\_options, sex\_options, embark\_options, min\_fare, max\_fare, min\_age, max\_age = load\_df()

res = df

name\_query = st.text\_input("String match for Name")

cols = st.columns(4)

survival = cols[0].multiselect("Survived", survival\_options)

p\_class = cols[1].multiselect("Passenger Class", p\_class\_options)

sex = cols[2].multiselect("Sex", sex\_options)

embark = cols[3].multiselect("Embarked", embark\_options)

range\_cols = st.columns(3)

min\_fare\_range, max\_fare\_range = range\_cols[0].slider("Lowest Fare", float(min\_fare), float(max\_fare),

[float(min\_fare), float(max\_fare)])

min\_age\_range, max\_age\_range = range\_cols[2].slider("Lowest Age", float(min\_age), float(max\_age),

[float(min\_age), float(max\_age)])

if name\_query != "":

res = res.loc[res.Name.str.contains(name\_query)]

if survival:

res = check\_rows("Survived", survival)

if p\_class:

res = check\_rows("Pclass", p\_class)

if sex:

res = check\_rows("Sex", sex)

if embark:

res = check\_rows("Embarked", embark)

if range\_cols[0].checkbox("Use Fare Range"):

res = res.loc[(res.Fare > min\_fare\_range) & (res.Age < max\_fare\_range)]

if range\_cols[2].checkbox("Use Age Range"):

res = res.loc[(res.Age > min\_age\_range) & (res.Age < max\_age\_range)]

removal\_columns = st.multiselect("Select Columns to Remove", df.columns.tolist())

for column in removal\_columns:

res = res.drop(column, axis=1)

st.write(res)

By the end of this chapter, you will be able to understand and parse each line of this code.

* Importing the Libraries
* Caching Data
* Creating our App Layout
* Using User Inputs to Produce a New DataFrame

### Importing the Libraries

At the start of our Python file, we first import the required libraries. We are using Streamlit for the app development and Pandas for working with our data.

import streamlit as st

import pandas as pd

### Caching Data

Let’s examine the next section of code.

@st.cache()

def load\_df():

df = pd.read\_csv("./data/titanic.csv")

survival\_options = df.Survived.unique()

p\_class\_options = df.Pclass.unique()

sex\_options = df.Sex.unique()

embark\_options = df.Embarked.unique()

min\_fare = df.Fare.min()

max\_fare = df.Fare.max()

min\_age = df.Age.min()

max\_age = df.Age.max()

return df, survival\_options, p\_class\_options, sex\_options, embark\_options, min\_fare, max\_fare, min\_age, max\_age

The first line is:

@st.cache\_data()

This decorator followed by st.cache\_data() establishes that resulting objects from the function that proceeds it should be cached into memory.

The next line begins the creation of our function.

def load\_df():

Once we have defined our function we begin working with our data. First, we load the data:

df = pd.read\_csv("./data/titanic.csv")

Next, our application will leverage four st.multiselect() widgets for four different columns in our dataframe. In order to populate a list of options for users to select, we need to know the unique values of each column. We can grab each unique value with .unique() on each column.

survival\_options = df.Survived.unique()

p\_class\_options = df.Pclass.unique()

sex\_options = df.Sex.unique()

embark\_options = df.Embarked.unique()

Next, our application will also leverage two sliders: one for Age and one for Fare. These will allow a user to find results based on a person’s age or the fare of their ticket. We need to know the max value and the minimum value for each of these so that we can automatically set the slider minimum and maximum values.

min\_fare = df.Fare.min()

max\_fare = df.Fare.max()

min\_age = df.Age.min()

max\_age = df.Age.max()

Finally, we return all of these values so that when the function is called each of these will be returned.

return df, survival\_options, p\_class\_options, sex\_options, embark\_options, min\_fare, max\_fare, min\_age, max\_age

Once we have created this function, we can call it and create all the objects that we need with the following code:

df, survival\_options, p\_class\_options, sex\_options, embark\_options, min\_fare, max\_fare, min\_age, max\_age = load\_df()

We will also create another object whose variable name will be res. This will be the dataframe that gets manipulated by the user and populates the results in the app.

res = df

### Creating our App Layout

After preparing all the data, now comes the time to design the general layout of our application. We use the following code to that:

name\_query = st.text\_input("String match for Name")

cols = st.columns(4)

survival = cols[0].multiselect("Survived", survival\_options)

p\_class = cols[1].multiselect("Passenger Class", p\_class\_options)

sex = cols[2].multiselect("Sex", sex\_options)

embark = cols[3].multiselect("Embarked", embark\_options)

range\_cols = st.columns(3)

min\_fare\_range, max\_fare\_range = range\_cols[0].slider("Lowest Fare", float(min\_fare), float(max\_fare),

[float(min\_fare), float(max\_fare)])

min\_age\_range, max\_age\_range = range\_cols[2].slider("Lowest Age", float(min\_age), float(max\_age),

[float(min\_age), float(max\_age)])

Let’s break down this section of the code. In the first line, we create an object name\_query. This will be a string that is returned from a st.text\_input() widget. We will use this input to query the Name field in the dataframe.

name\_query = st.text\_input("String match for Name")

Next, we will create four columns that we can populate with our st.multiselect() widgets.

cols = st.columns(4)

Now that we have our columns, we can create our four st.multiselect() widgets. The user will be able to select which items that want to see returned for each field in the dataframe. Each of these will return a list of options. We will gather data for four fields: Survived, Pclass, Sex, and Embarked.

survival = cols[0].multiselect("Survived", survival\_options)

p\_class = cols[1].multiselect("Passenger Class", p\_class\_options)

sex = cols[2].multiselect("Sex", sex\_options)

embark = cols[3].multiselect("Embarked", embark\_options)

Next, we need three new columns so that we can populate our two range sliders. We are using three columns here so that there is a large gap between the two sliders.

range\_cols = st.columns(3)

We will populate the first and last slider, we will place two st.slider() widgets. We will use the minimum and maximum values for the Age and Fare fields in the dataframe.

min\_fare\_range, max\_fare\_range = range\_cols[0].slider("Lowest Fare", float(min\_fare), float(max\_fare),

[float(min\_fare), float(max\_fare)])

min\_age\_range, max\_age\_range = range\_cols[2].slider("Lowest Age", float(min\_age), float(max\_age),

[float(min\_age), float(max\_age)])

### Using User Inputs to Produce a New DataFrame

With the general layout designed, we can then work with the input from the user to modify our res dataframe. The following code manipulates the dataframe through a set of conditions we generate from the user input.

if name\_query != "":

res = res.loc[res.Name.str.contains(name\_query)]

if survival:

res = check\_rows("Survived", survival)

if p\_class:

res = check\_rows("Pclass", p\_class)

if sex:

res = check\_rows("Sex", sex)

if embark:

res = check\_rows("Embarked", embark)

if range\_cols[0].checkbox("Use Fare Range"):

res = res.loc[(res.Fare > min\_fare\_range) & (res.Age < max\_fare\_range)]

if range\_cols[2].checkbox("Use Age Range"):

res = res.loc[(res.Age > min\_age\_range) & (res.Age < max\_age\_range)]

removal\_columns = st.multiselect("Select Columns to Remove", df.columns.tolist())

for column in removal\_columns:

res = res.drop(column, axis=1)

st.write(res)

First, we check to see if the user has written anything in the st.text\_input() widget. If it is, then we will narrow the dataframe down to anything that matches the user’s string.

if name\_query != "":

res = res.loc[res.Name.str.contains(name\_query)]

For each of the st.multiselect() widget inputs, we check to see if the user has selected how to narrow the dataframe:

if survival:

res = check\_rows("Survived", survival)

if p\_class:

res = check\_rows("Pclass", p\_class)

if sex:

res = check\_rows("Sex", sex)

if embark:

res = check\_rows("Embarked", embark)

Next, we will use the input from the minimum and maximum values of the sliders for Age and Fare. So that we do not ignore results where Age or Fare are NaN in the dataset, we want to give the user the ability to check an st.checkbox() widget. This will allow the user to activate or deactivate the sliders.

if range\_cols[0].checkbox("Use Fare Range"):

res = res.loc[(res.Fare > min\_fare\_range) & (res.Age < max\_fare\_range)]

if range\_cols[2].checkbox("Use Age Range"):

res = res.loc[(res.Age > min\_age\_range) & (res.Age < max\_age\_range)]

We also want to give the user one final input, the ability to narrow down and delete columns from the dataset. The reason for this is because not all fields will be relevant to every user and since this is a large dataset, it makes sense to give them the ability to limit which fields they are seeing to the ones they want.

removal\_columns = st.multiselect("Select Columns to Remove", df.columns.tolist())

for column in removal\_columns:

res = res.drop(column, axis=1)

As we have manipulated the res dataframe throughout each of these lines, we are finally ready to display the data:

st.write(res)

## Deploying an App in the Cloud with Streamlit Share

Once you have designed an application and have tested it locally, it comes time to share it with others. We can do this with Streamlit by leveraging several different cloud-based services. Fortunately, Streamlit offers a free Streamlit Share service that lets users share their apps for free.

* Create a GitHub Account
* Upload Application to GitHub
* Connect Streamlit Share to your GitHub
* Create a New App
* Set Custom Subdomain

### Create a GitHub Account

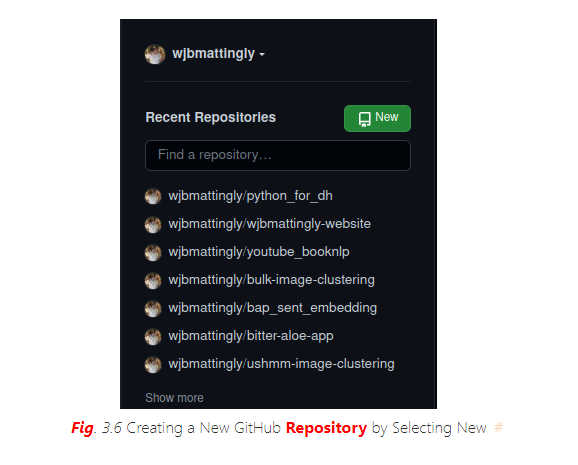
In order to deploy your application, you will need a place to store your application’s source code. The easiest way to do this is via GitHub which Streamlit and many other application hosting services support.

### Upload Application to GitHub

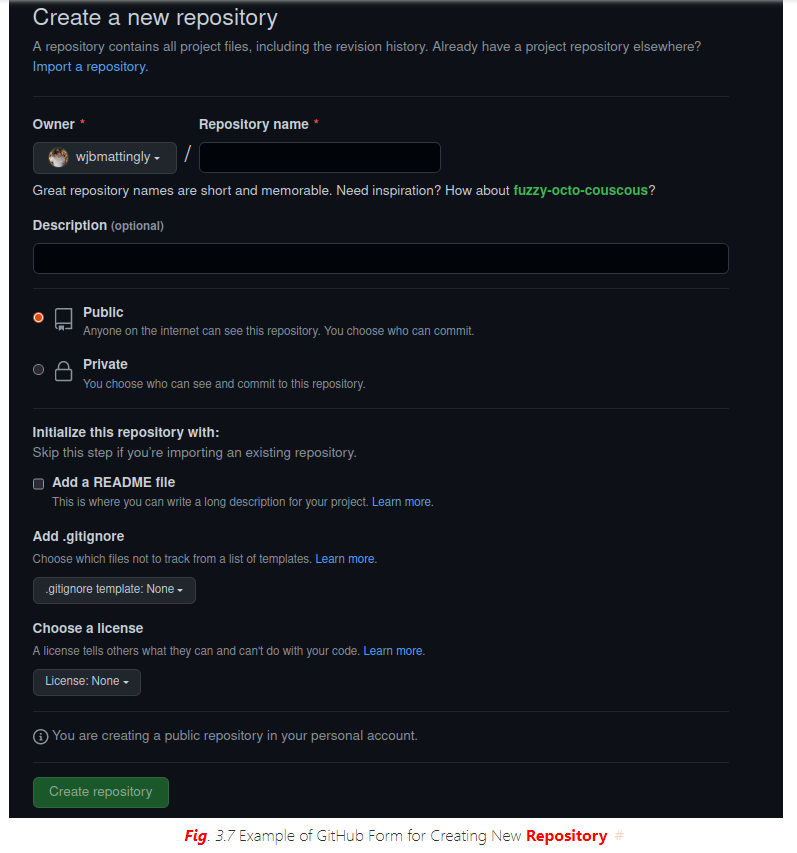
Once you have created your GitHub account, you can create a repository and upload the code into the repository. If you are just starting out with GitHub, getting used to Git can be a bit daunting. Git is powerful, but has a steep learning curve. It allows you to maintain your code, especially in teams, and version control everything. This means that your code is always backed up and you can access previous versions of your code at any single time.

To upload your application formally, you would want to be familiar with Git and how to perform basic tasks, such as cloning a repository, adding files to it, committing those changes, and then pushing them to the repository. While this is the better approach, it is not the only one. For now, you can get your application up and running entirely in your browser.

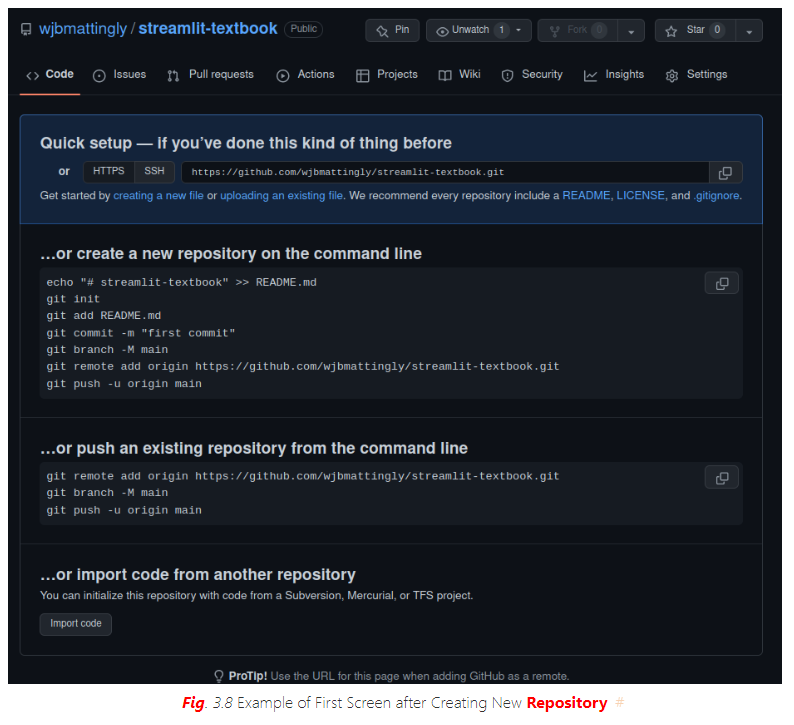
First, we will create a new repository. To do this, we will go to our GitHub main page and select New, found in the top-left side of your screen.



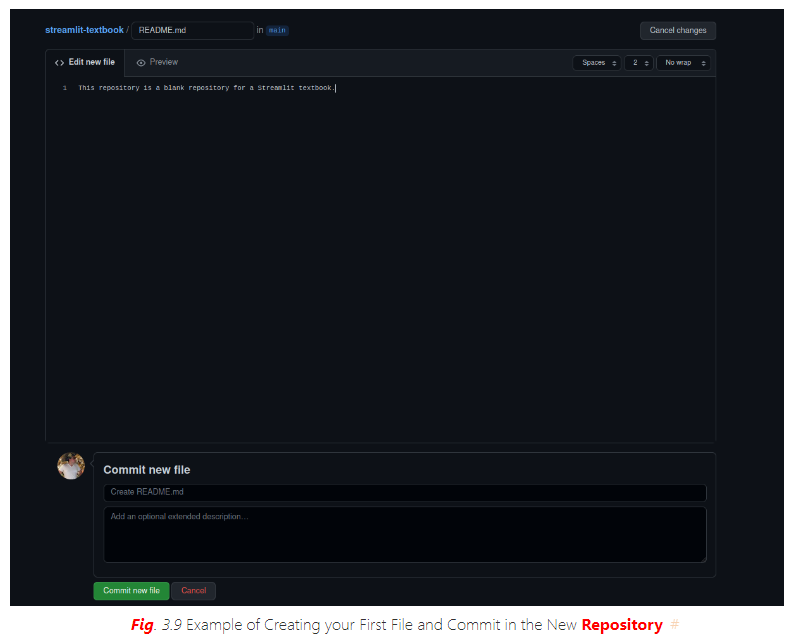
Once you click New, you will see the following screen:



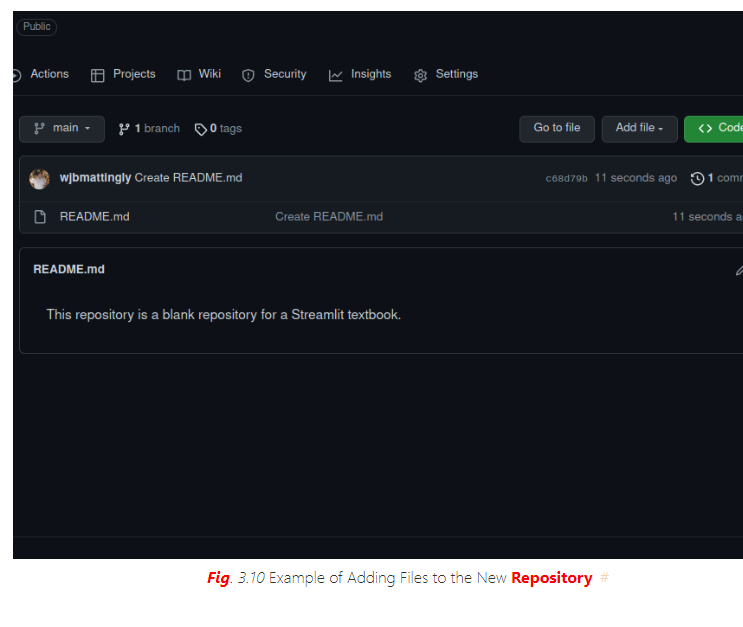
Here, we will fill out the details of our repository. We need to first give it a unique name. We can keep all other settings the default for now. Once done, we will click Create repository. After this, you will see a new screen that looks like this:



You will now select creating a new file. Once you do, you will see a screen that looks like this:



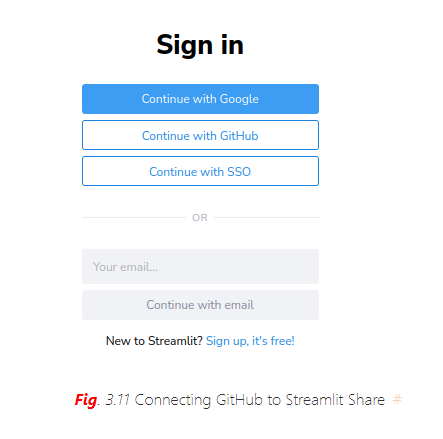
Here, you will create a file called README.md and assign some text to it. This will ideally describe your application. It will be what users see when they visit your GitHub repository. Once you have a description you are happy with, click Commit new file. This will lead to a final new screen.



Next, we need upload our application and data to our repository. To do this, we will click Add file and upload our local files that are necessary for running our Streamlit application. You will also want to make a requirements.txt file lists all required libraries, such as Pandas.

### Connect Streamlit Share to your GitHub

Now that we have our repository created, we need to link Streamlit Share to our GitHub. To do so, you will need to visit https://share.streamlit.io/ Once on this page, you will see a screen that looks like this:

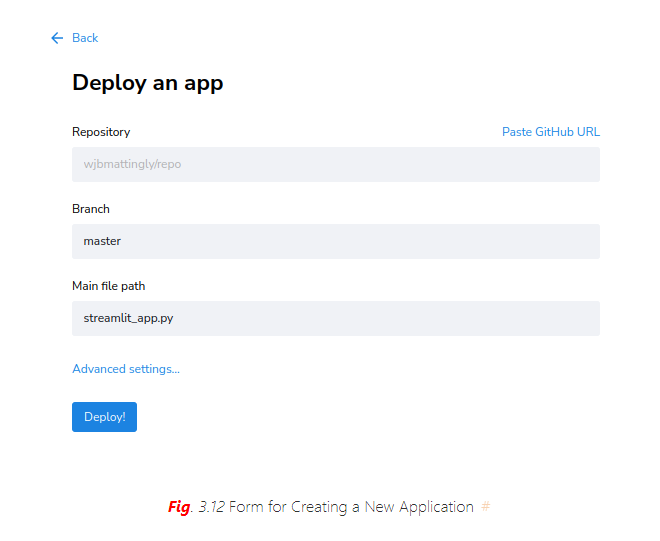


Select Continue with GitHub and use your GitHub credentials for login verification. Once you are logged in, you will see a screen that looks like this at the top:

````{figure} `../images/streamlit/streamlit\_mainpage.png Main Screen of Streamlit Share

### Create a New App

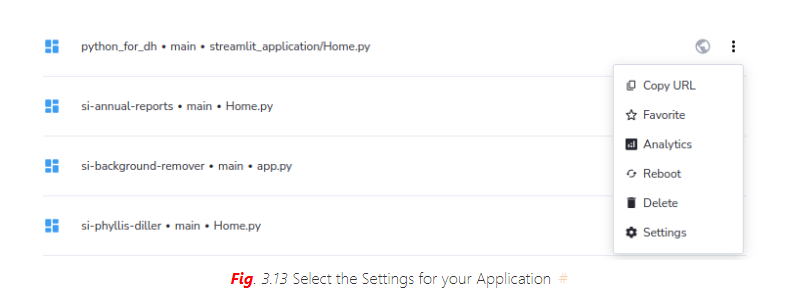
Select New App. After a few seconds, you will be taken to a page that looks like this:



Click on each of these fields. First, select your repository. Streamlit will be connected to your GitHub, so you will be able to see all your repositories, both private and public. Next, select the branch of your repository. We have not covered branches in this textbook. Branches are a function of Git. By default, your application will be on the main branch of your repository. Finally, select the main Python file in which your application rests. Once you have filled out these three fields, click Deploy!.

### Set Custom Subdomain

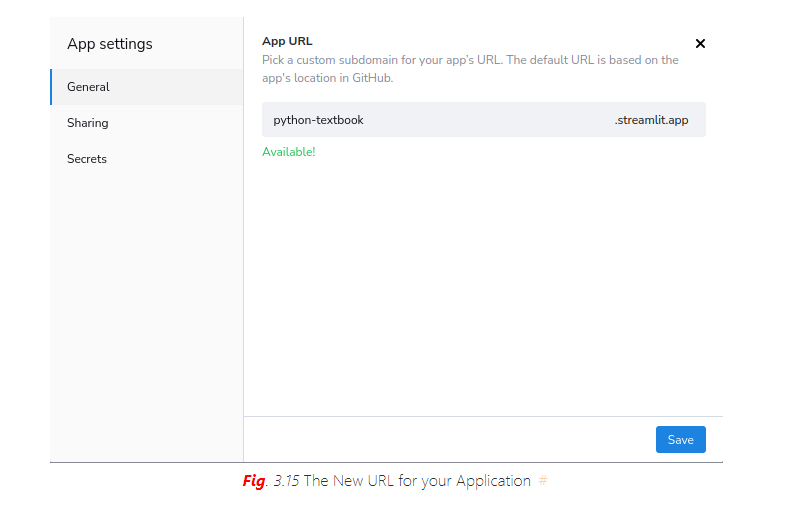
It will take several minutes (depending on the complexity of your application and the number of required libraries you wish to have installed). At this point, Streamlit Share is building an environment on a server to host your application in the cloud. Once complete, it will provide you with a unique domain for your application with the extension. This will be based on your repository name and your GitHub username. We can create a custom subdomain for free by returning to our Streamlit Share homepage and selecting the three dots next to our application’s name.



Once here, select Settings and you will see a page that looks like this:



Note that in the middle of the screen, you will see your long subdomain. You can now change it to something unique and easier to remember for distributing to users. After you change your subdomain, click Save.



Now, your app is in the cloud with a unique and easy-to-remember subdomain!

