3

Exploring Streamlit's Features and Functionalities

This chapter is a beginner-friendly tutorial that introduces users to the core features and functionalities of the Streamlit Python framework, aiming to help users get started with the library quickly and easily. The chapter covers the basic concepts of Streamlit, such as creating and customizing widgets, laying out the user interface, and adding visualizations and charts.

By the end of this chapter, you should be able to create and run your own Streamlit apps and have a solid understanding of the library’s features and capabilities. This is the first hands-on step of our exciting journey together!

In this chapter we’re going to cover the following main topics:

* Installing and launching Streamlit
* Streamlit features and widgets
* Dataframes, Plots, and Visualizations
* Date, Time and more

Technical requirements

* In this chapter, we will use the following libraries, packages and tools:
* Sublime Text
* Python3
* pipenv
* streamlit
* csv files
* Code in the chapter can be accessed through the following GitHub link: https://github.com/PacktPublishing/Web-App-Development-Made-Simple-with-Streamlit/tree/fcb2bd740a2df7263b4470164805926fee3157a1/Chapter03

Installing and launching Streamlit

Finally, we are ready to write our code in order to create beautiful web applications! Where to start from? The first thing to do is to install Streamlit.

So, let’s create a new directory, for example we can call it streamlit\_course, and once inside it, prepare a new virtual environment by typing the well-known command, pipenv shell. Once you are done running the pipenv shell command, you will get the following result:

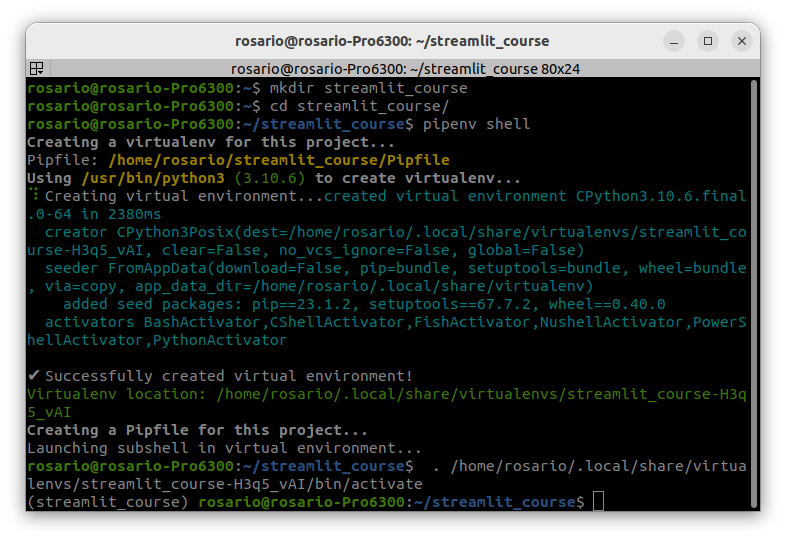


Figure 3.1: streamlit\_course virtual environment creation

Installing Streamlit is really very easy, a matter of a simple instruction as we can read on its official website ([www.streamlit.io](http://www.streamlit.io)).

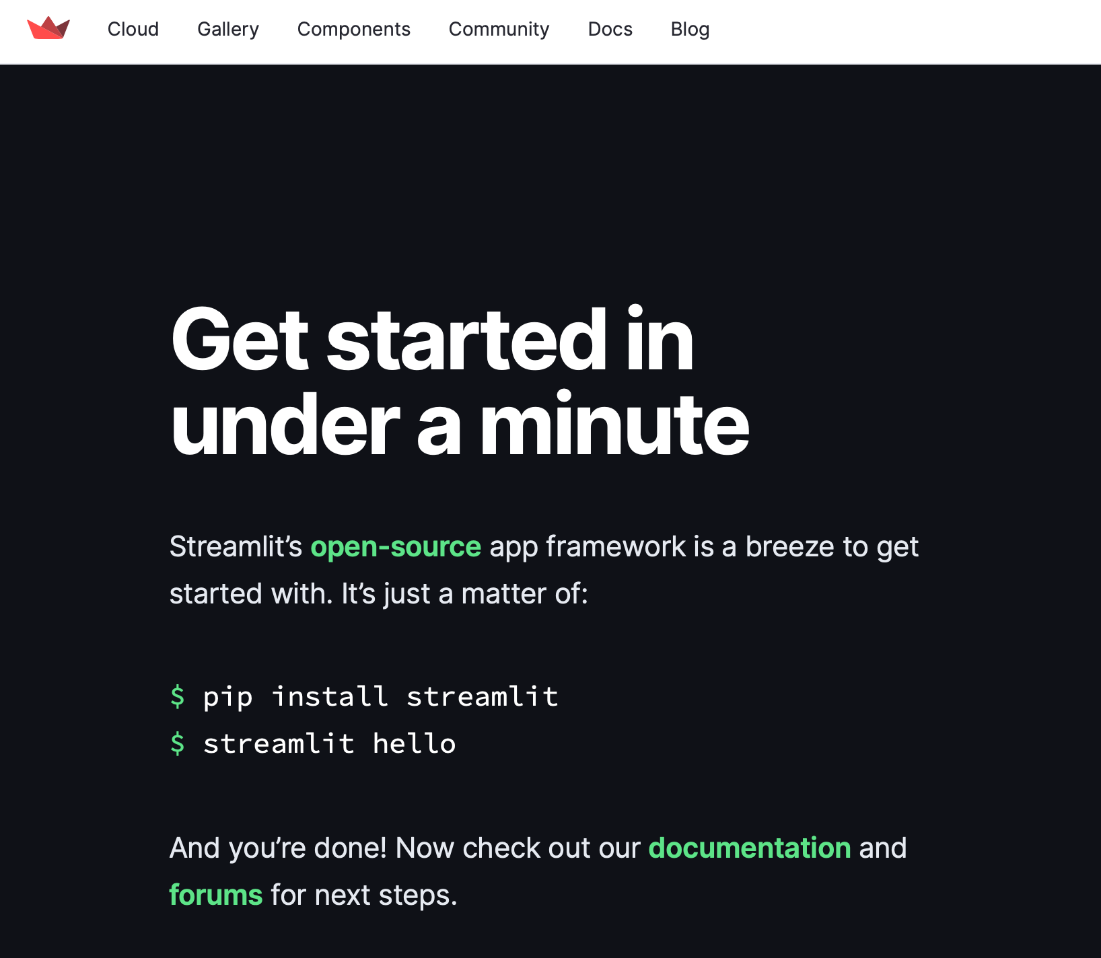


Figure 3.2: Installing Streamlit

Since we are using pipenv and streamlit\_course virtual environment that we just created, we have to modify a little bit the instruction suggested in Figure 3.2 in the following way:

pipenv install streamlit

You will get the following result:

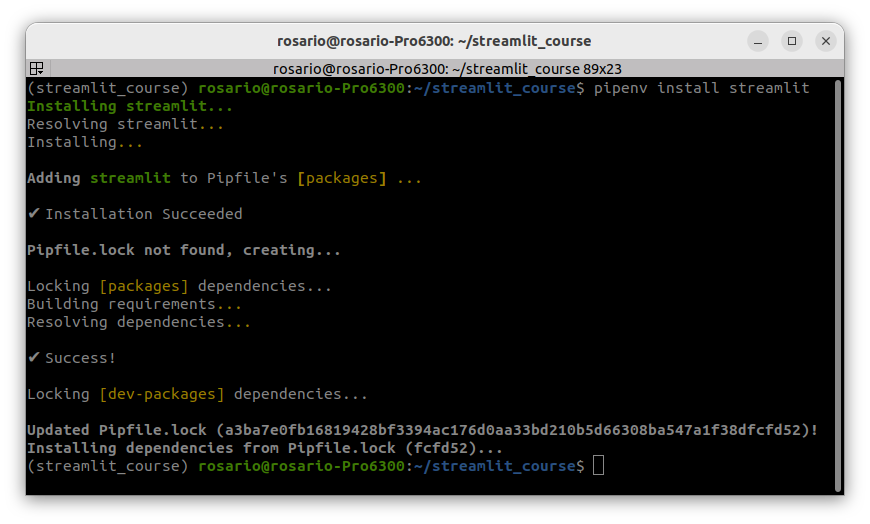


Figure 3.3: Streamlit installation with pipenv

In this way, Streamlit will be easily installed and the Pipfile will be updated.

Now, let’s create an empty file to be used as our python script by typing the following:

touch app.py

As we can see in our streamlit\_course directory, we have three files:



Figure 3.4: the files in the streamlit\_course directory

Up to now, we have installed Streamlit and we have an empty python file, even if not that much we are ready to launch our first web application!

The instruction to launch Streamlit is very simple, just type streamlit run <python\_file\_name.py>, where the python file in our case is the empty app.py we just created. But, since we are in a pipenv virtual environment, we have to tell to pipenv that we want to launch something, so finally we can write:

pipenv run streamlit run app.py

In this way, pipenv launches Streamlit that runs the app.py file. We see that our browser opens on an empty page, but if we check carefully the address bar we can see that something is running on localhost:8501. Our streamlit web app is alive and running on our local machine on its standard 8501 port. (if you are asked to write your email in the terminal, just push enter).

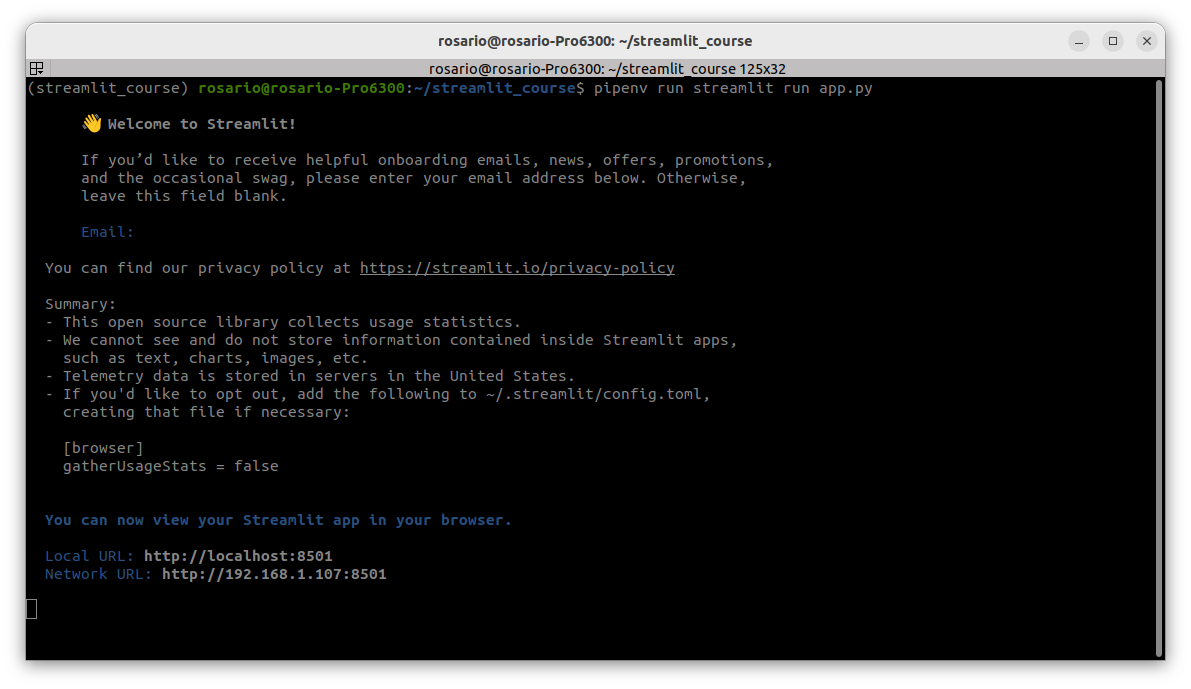


Figure 3.5: Streamlit running on localhost:8501

Next, our web app can be improved by adding widgets and elements. Open another terminal in our terminator (another tile), enter our virtual environment, and run our Sublime Text editor with "subl .".

Streamlit features and widgets

The very first step has been completed: Streamlit is up and running. What we need to do now is to add text, widgets, elements, etc. to make at the same time something beautiful and correctly working.

In order to start populating our web app with nice and useful widgets we need to write some Python code and the best way is to put one close to the other our Sublime Text and browser like the ones showed in the following image:

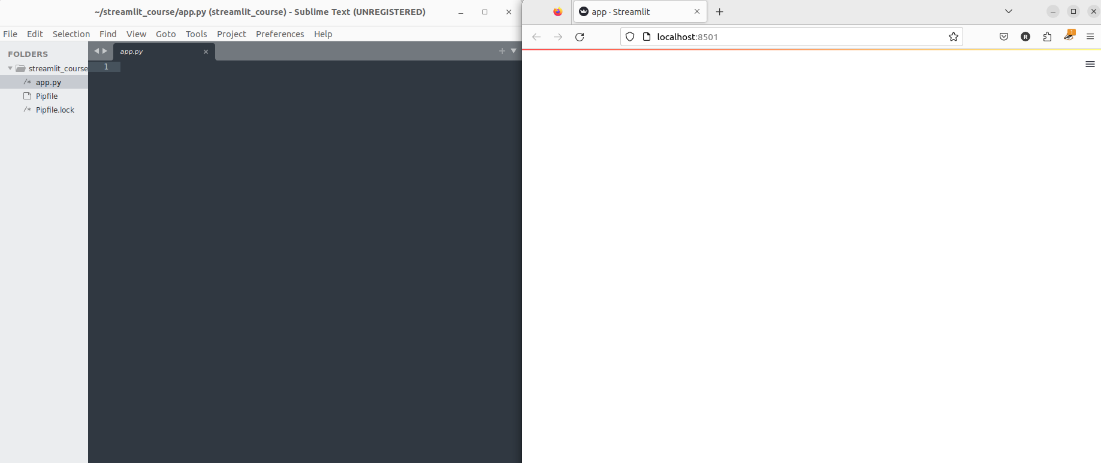


Figure 3.6: Sublime Text and Web Browser

This kind of visualization is very convenient because we can see immediately any change we do to the code (real-time, as soon as we save our code changes), in our editor directly, in the browser by just selecting Always Rerun in the top right menu of our web application.

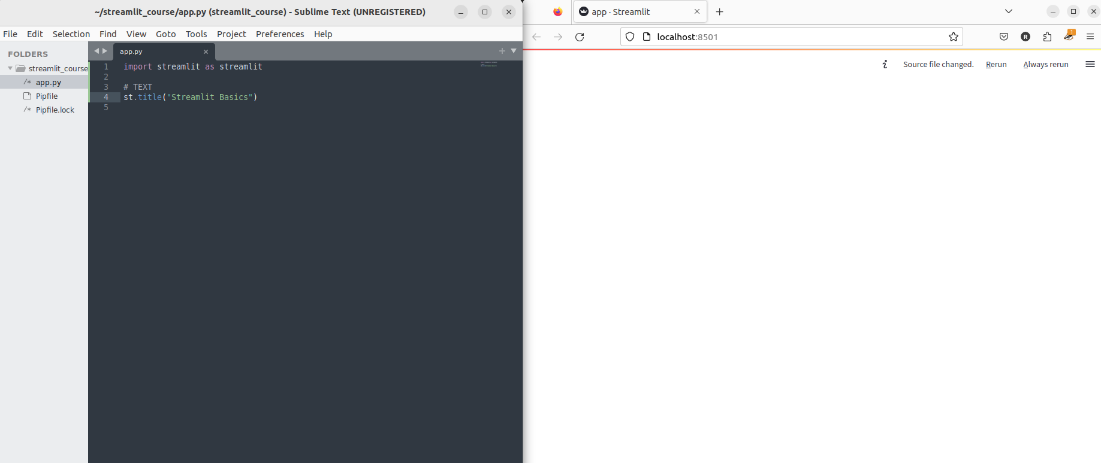


Figure 3.7: Code changes and “Always Rerun”

So, let’s import streamlit (with st as alias) and start dealing with some text. We can write:

import streamlit as st

st.title(“Streamlit Basics”)

The result is shown in Figure 3.8:

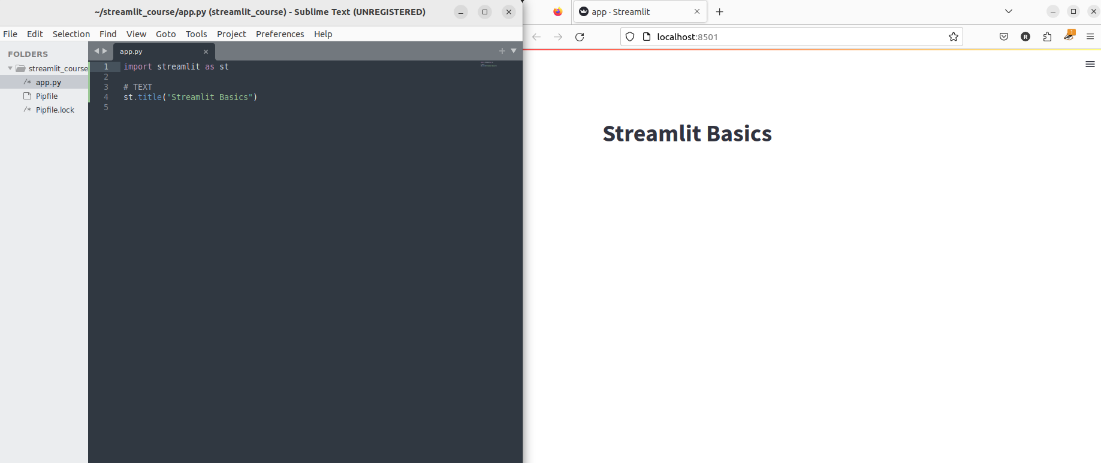


Figure 3.8: Code changes and its effect on the web app

st.title gives back a quite big text but for sure we can use many other text dimensions, in fact, always in our Sublime Text, we can write and save the following code:

st.header(“This is a header”)

st.subheader(“This is a subheader”)

st.text(“This is a simple text”)

st.write(“This is a write dimension”)

And since we already selected Always Rerun, we immediately see in the browser that our web app changes introducing the header, subheader, text and write text dimensions we decide to visualize.

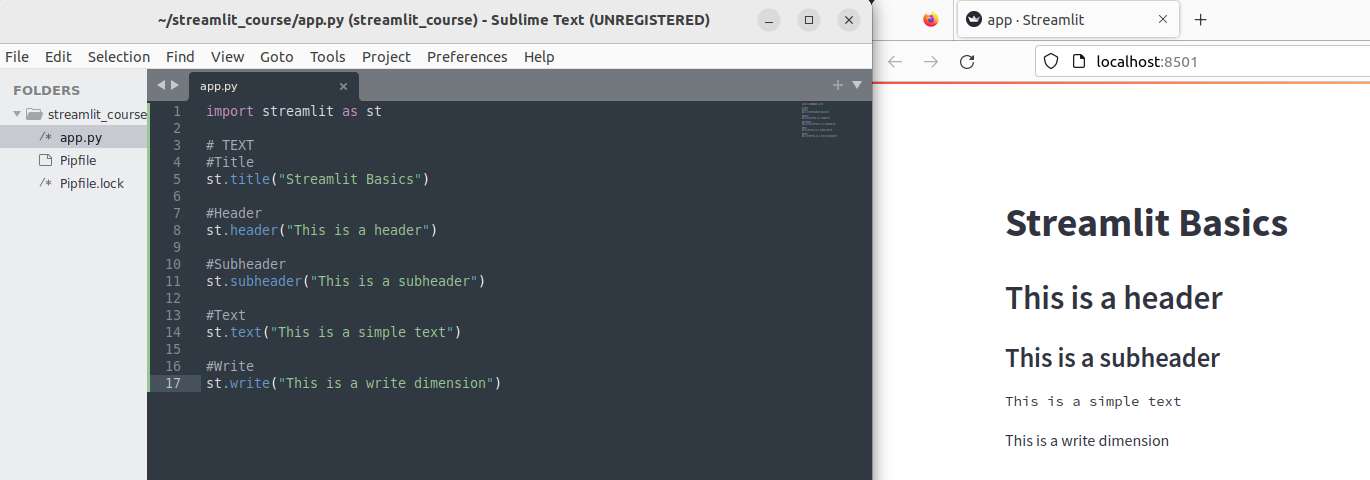


Figure 3.9: Different “text dimensions”

Streamlit can directly manage even the markdown, and also this is quite simple since we just have to use markdown and pass the text inside the parenthesis. For example, we can write:

st.markdown(“[Streamlit](https://www.streamlit.io)”)

In this way, we write on the screen the word “Streamlit” as an hyperlink to the official Streamlit website. If we prefer to put the link directly on the screen, so making visible the URL, we can write:

st.markdown(“https://www.streamlit.io”)

In Streamlit we can use also HTML in a very simple way, we just need to create a variable containinig all our HTML code, then put it inside a markdown instruction together with the argument unsafe\_allow\_html set to True, let’s see it:

html\_page = “””

<div style=”background-color:blue;padding:50px”>

<p style=”color:yellow;font-size:50px”>Enjoy Streamlit!</p>

</div>

“””

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

And this is the result we get:

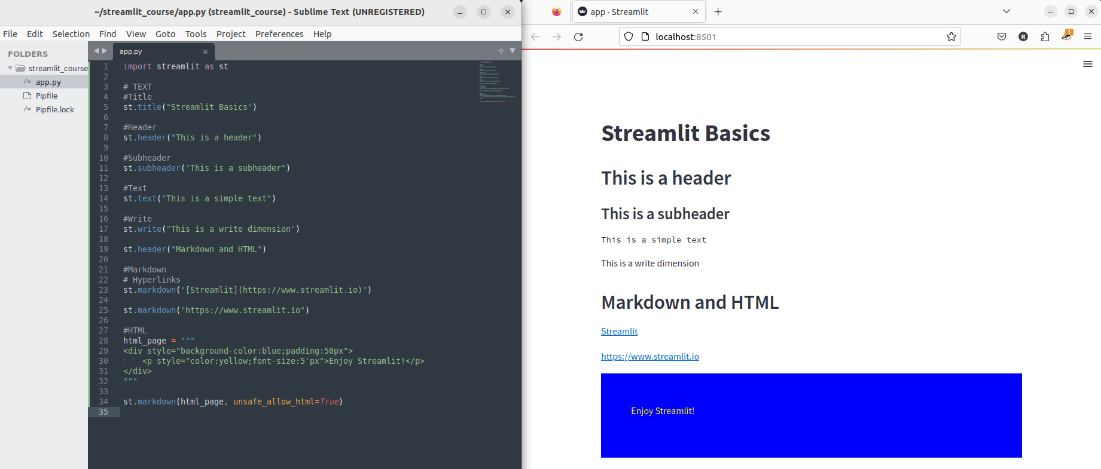


Figure 3.10: markdown and HTML

You can try by yourself to see what happens when you set unsafe\_allow\_html to False.

Colored text boxes

Continuing talking about text, we can have a beautiful text boxes of different colors to indicate a warning, an error, etc. And this kind of color code can be very useful during our web applications building. Let’s check the code:

st.success(“Success!”)

st.info(“Information”)

st.warning(“This is a warning!”)

st.error(“””This is an error!”)

The first one returns a green box with some text, the second one a light blue box with text, the third one a yellowish box containing text and the last one a red box with the error message.

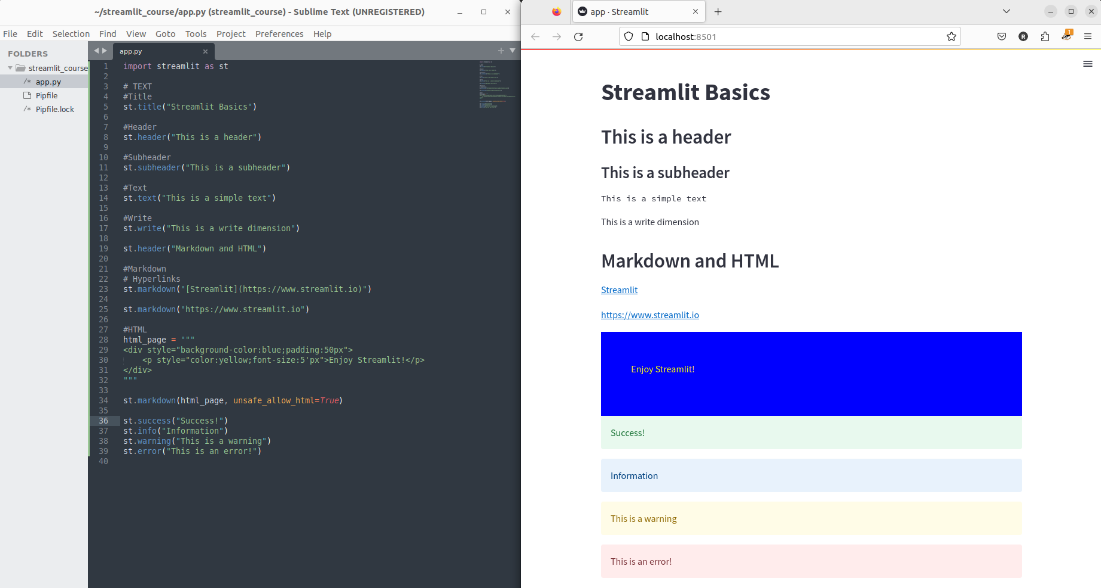


Figure 3.11: colored textboxes

//Color textboxes are something really interesting, since we can use it to advice about something wrong, for example an issue, using the reddish tone or something very good, for example a success case, using the greenish tone. Moreover, we can use this feature to give a little of vivacity to our text.

Images, audios, and videopip

In Streamlit it’s extremely easy also managing multimedia, so images, audio and video. Starting with images, we need to import the PIL library and then add a couple lines of code:

from PIL import Image

img = Image.open(“packt.jpeg”)

st.image(img, width=300, caption=”Packt Logo”)

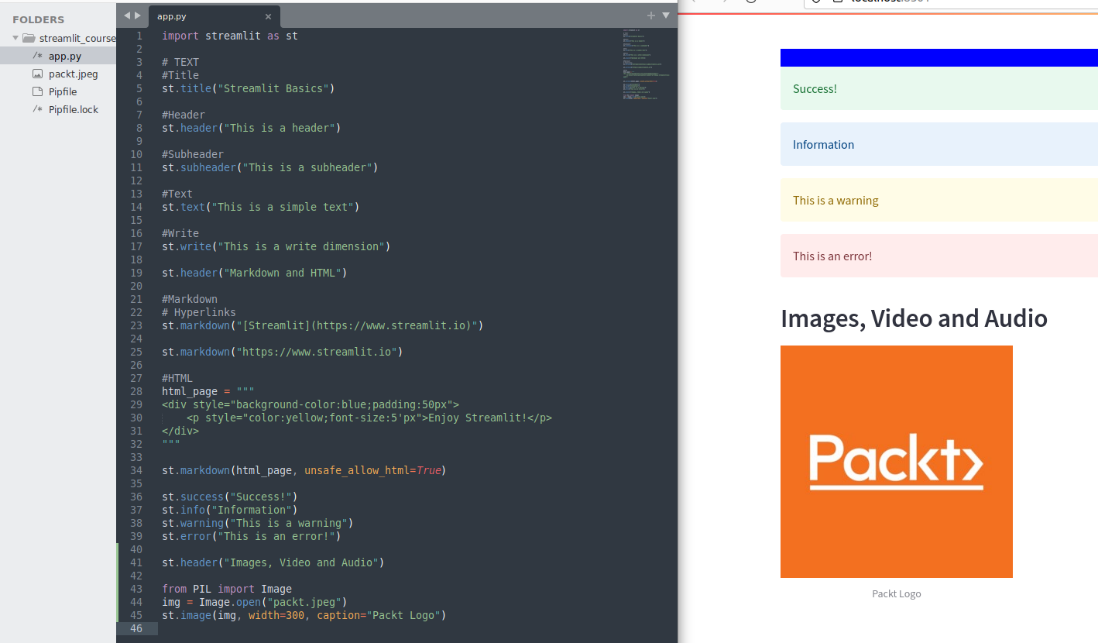


Figure 3.12: st.image

Please, note that the jpeg image is in the same directory of our app.py file, if we want, we can change the width and the caption of the image.

Working with video is not very different, in fact we can put a video file in the same directory of our app.py file and open it.

video\_file = open(“SampleVideo\_1280x720\_1mb.mp4”,”rb”)

video\_bytes = video\_file.read()

st.video(video\_bytes)

On the box, there are buttons for play/pause, volume control, and full-screen.

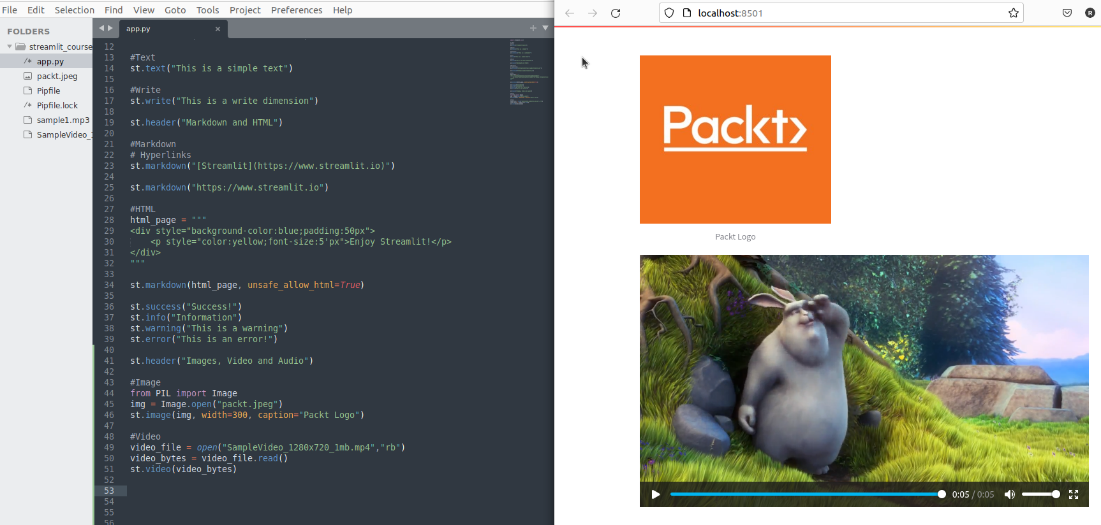


Figure 3.13: st.video from a file

Even more, we can open video directly from the web using a URL with the st.video widget. For example, we can write:

st.video(“https://www.youtube.com/watch?v=q2EqJW8VzJo”)

And the result is displayed in the following image:

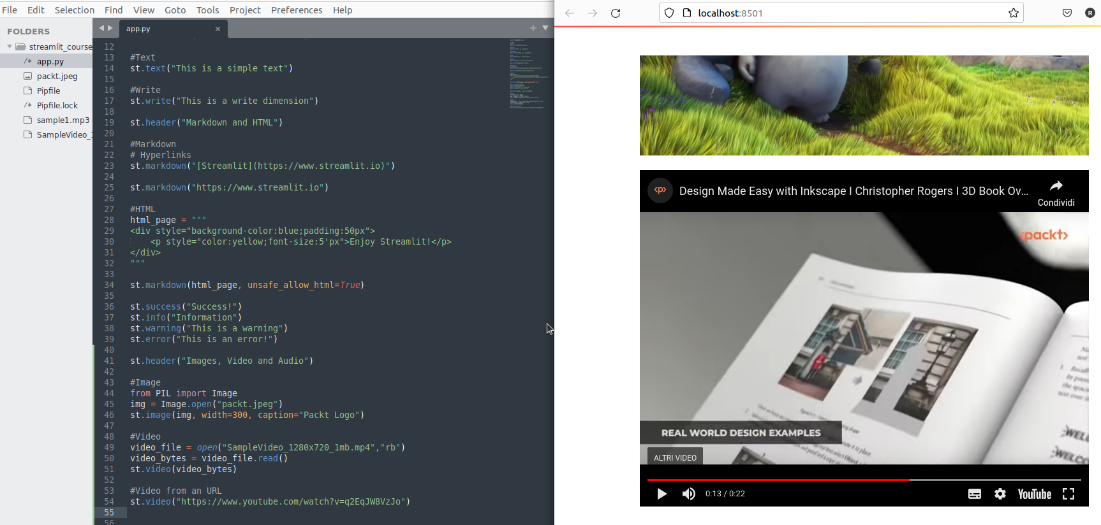


Figure 3.14: st.video from a URL

For audio files we can do more or less the same. We can write:

audio\_file = open(“sample1.mp3”, “rb”)

audio\_bytes = audio\_file.read()

st.audio(audio\_bytes, format=”audio/mp3”)

Please note that this time, we have to specify the format. Once again, out of the box, we get the play/pause button and volume control.

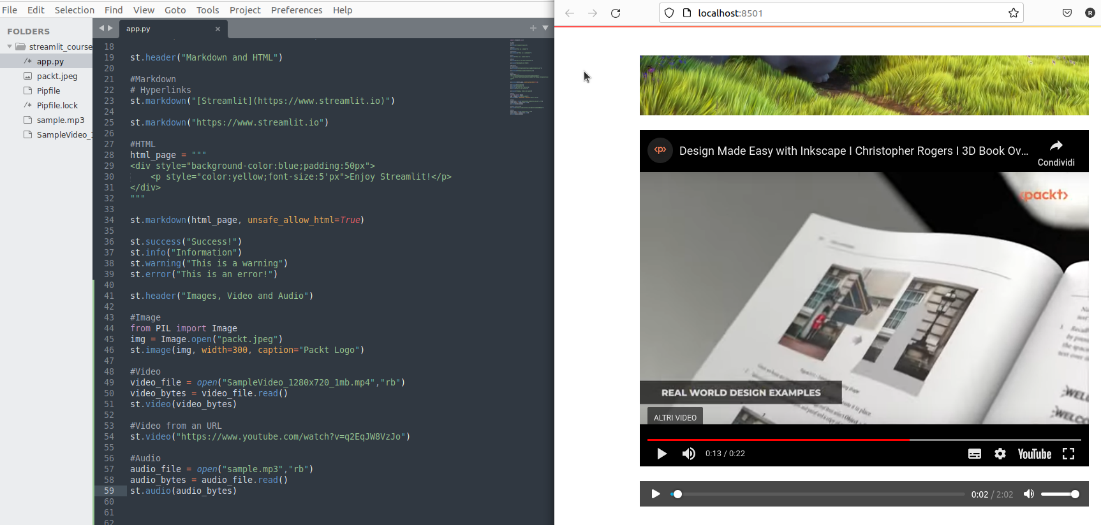


Figure 3.15: st.audio

Let’s move now to some other widget that will be very useful in next chapters. First up is the “button” widget. So please comment all code we wrote up to now (a comment is made just putting an “#” at the beginning of the line of code we want to ignore), excluding the instruction that imports Streamlit and continue. We can start writing a simple instruction:

st.button(“Play”)

This instruction gives us a beautiful button with a caption “Play” but when we click on it nothing happens!

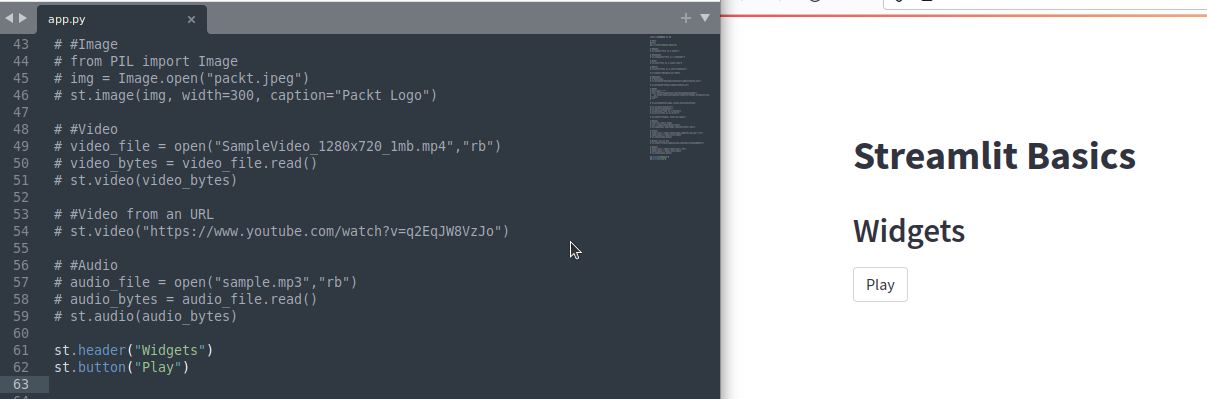


Figure 3.16: st.button

Nothing happens because there is no code related to the button, so things will change if we slightly change the previous line of code in the following way:

if st.button(“Play”):

st.text(“Hello world!”)

We see that when the “Play” button is clicked a beautiful text with “Hello World!” will appear.

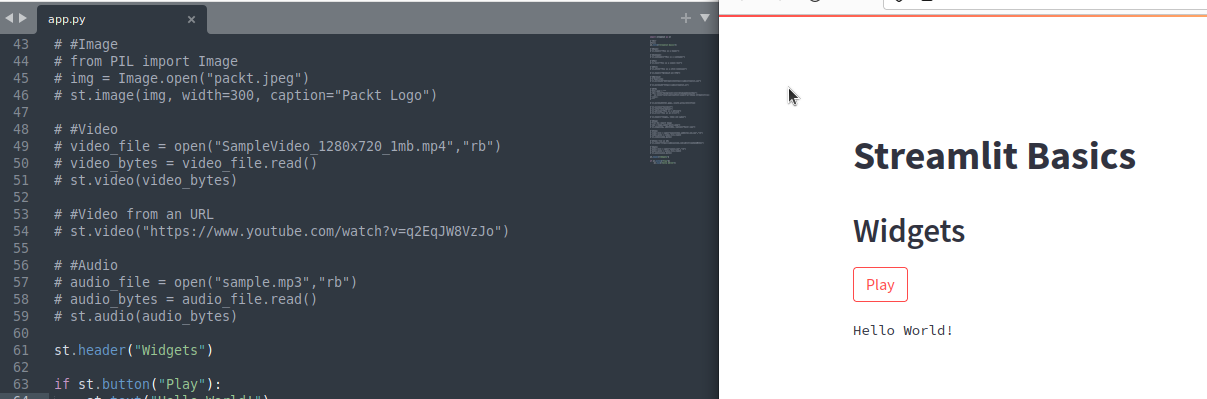


Figure 3.17: event associated to st.button

There are many other widgets that work in the same way, for example “checkbox”. If we write the following code:

if st.checkbox(“Checkbox”):

st.text(“Checkbox selected”)

We will get the result shown in Figure 3.18:

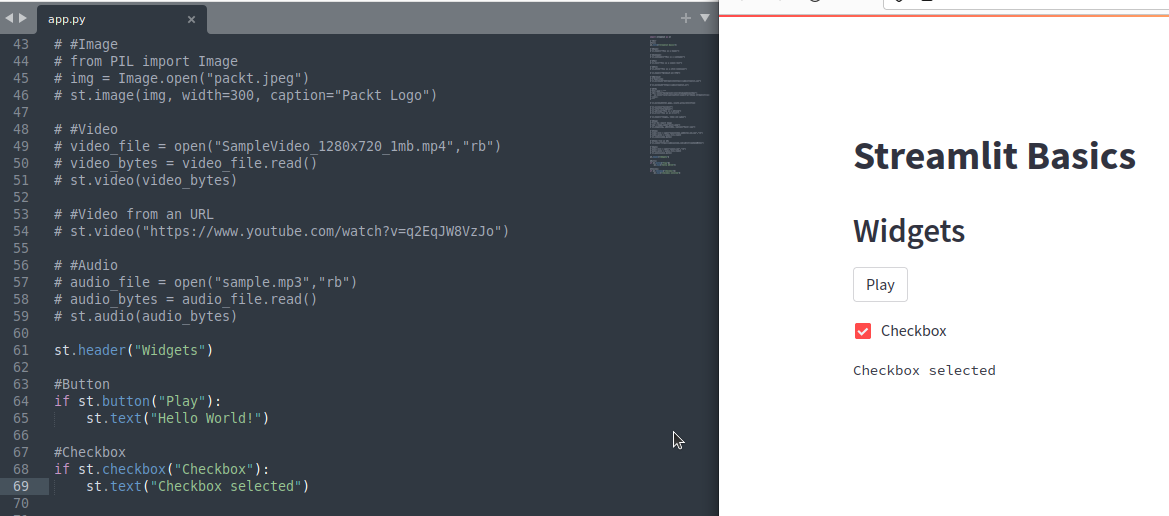


Figure 3.18: st.checkbox

The “Radio Button” works in a little different way, we have to specify a list of options then decide what happens when we select each one of them.

radio\_but = st.radio(“Your Selection”, [“A”, “B”])

if radio\_but == “A”:

st.info(“You selected A”)

else:

st.info(“You selected B”)

The preceding code will give the following result:

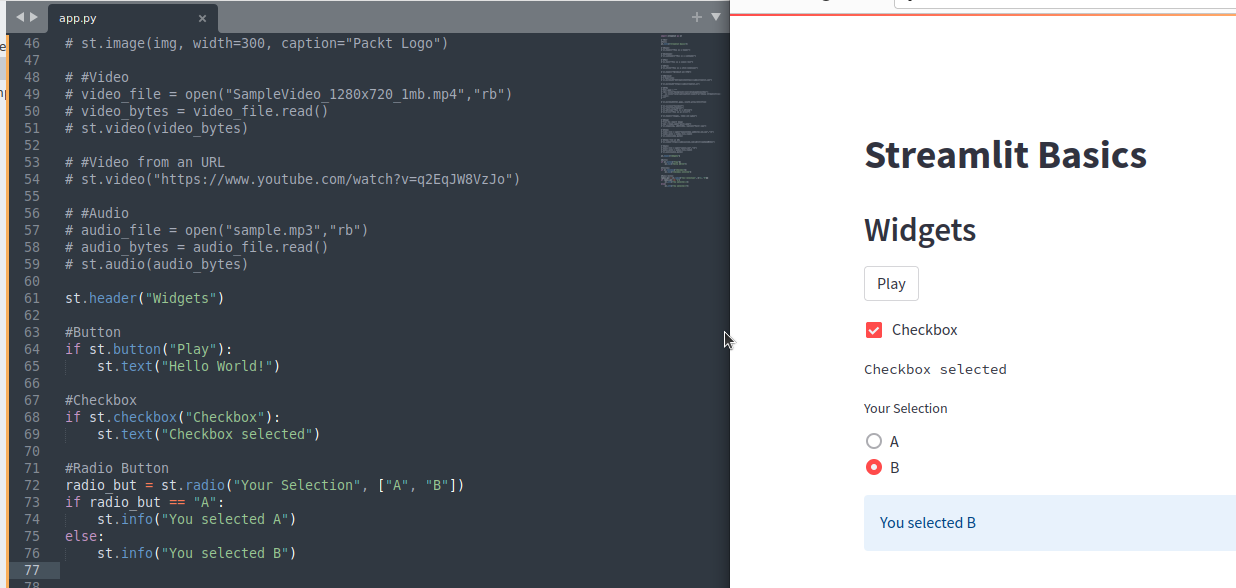


Figure 3.19: st.radio

Also in the selectbox we need to specify a list of options:

city = st.selectbox(“Your City”, [“Napoli”, “Palermo”, “Catania”])

We select one of the options (in this case an Italian city) that will be saved in the “city” variable.

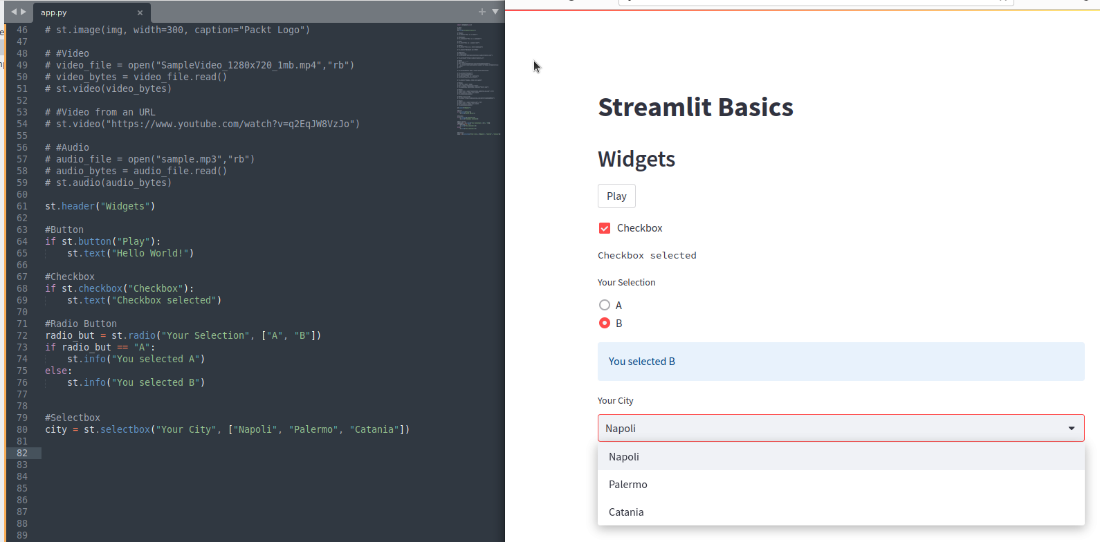


Figure 3.20: st.selectbox

In the case we want to have a multi-selection, we can use the multiselect widget in the following way:

occupation = st.multiselect(“Your Occupation”, [“Programmer”, “Data Scientist”, “IT Consultant”, “DBA”])

The coding is very similar to the previous one but this time we can select more than one option, as we can see in the figure where we selected two jobs (if we want clicking on the “x” we can cancel a selected option):

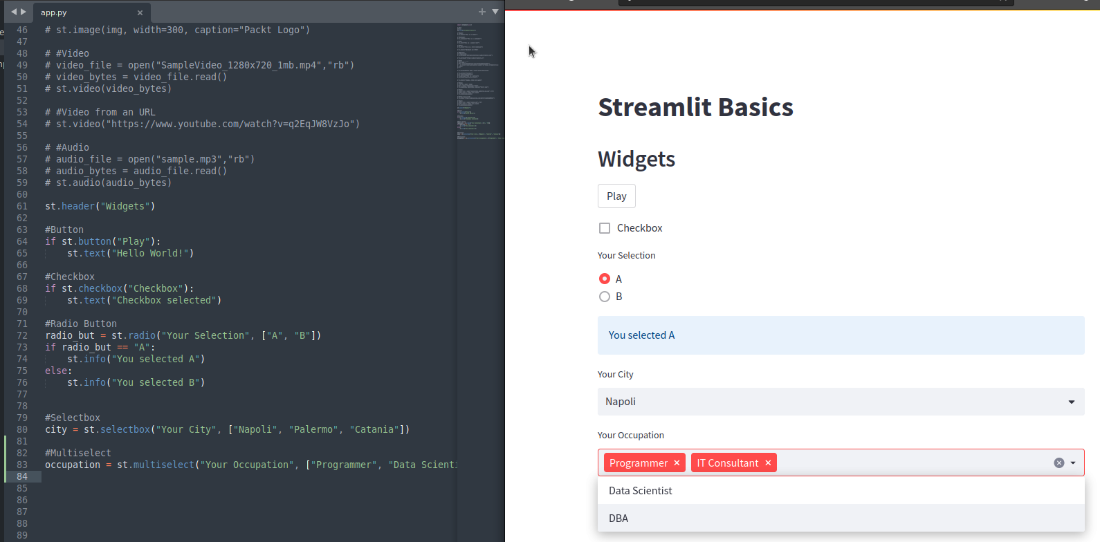


Figure 3.21: st.multiselect

Multiselect is a very elegant way to make multiple selections, keeping the screen clean and functional.

Inputing text and numbers

Another extremely useful function in our web application is the inputing, in few words the faculty of entering some information and obviously also in this case we have many widgets available out of the box.

In the text\_input widget we have only to specify a label or caption and a placeholder, very easy!.

name = st.text\_input(“Your Name”, “Write something...”)

st.text(name)

And everything we write will be saved in the “name” variable and printed on the screen thanks to st.text().

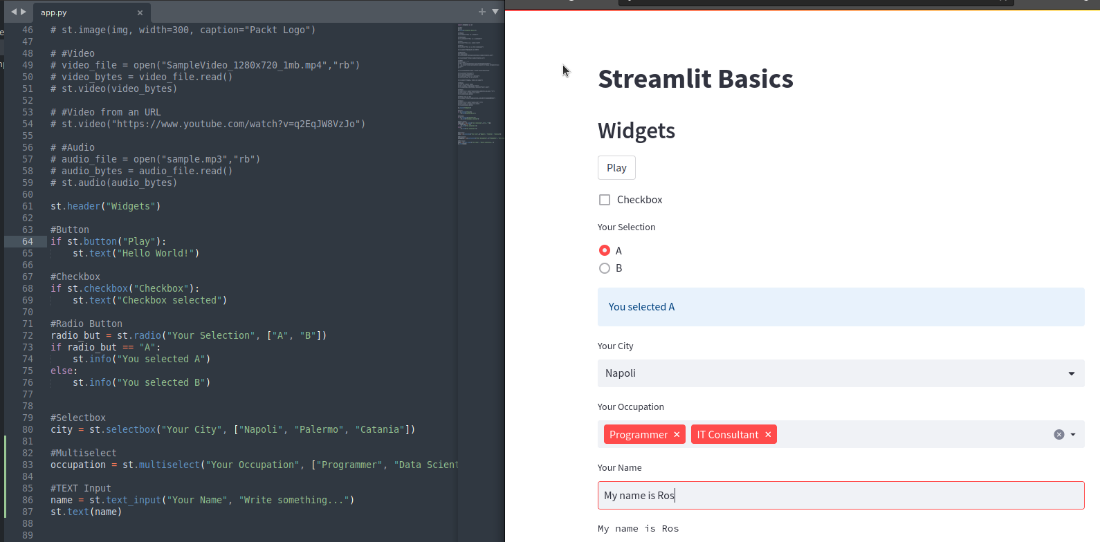


Figure 3.22: st.text\_input

In the same easy way, we can input also numbers and this time it’s possible to write directly a number or use the “+” and the “-” to increase or decrease it just using st.number\_input.

age = st.number\_input(“Input a number”)



Figure 3.23: st.number\_input

Moving back to text, to input text on more than one line we can use the text\_area widget, in the following way:

message = st.text\_area(“Your Message”, “Write something...”)

and as we can see, this time a wider text area will be displayed.

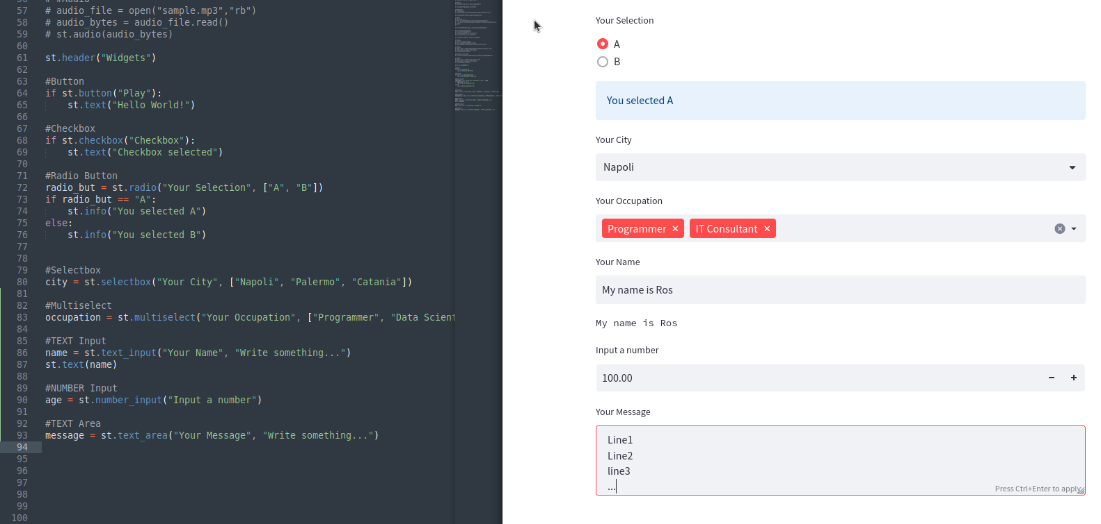


Figure 3.24: st.text\_area

Text Area is the perfect tool when we need to input a quite long text, and it is configurable accordingly to our real needs.

Slider

Another wonderful input widget is the slider, where we just need to specify a starting and an ending value to have a nice selector on the screen. The syntax is extremely easy:

select\_val = st.slider(“Select a Value”, 1, 10)

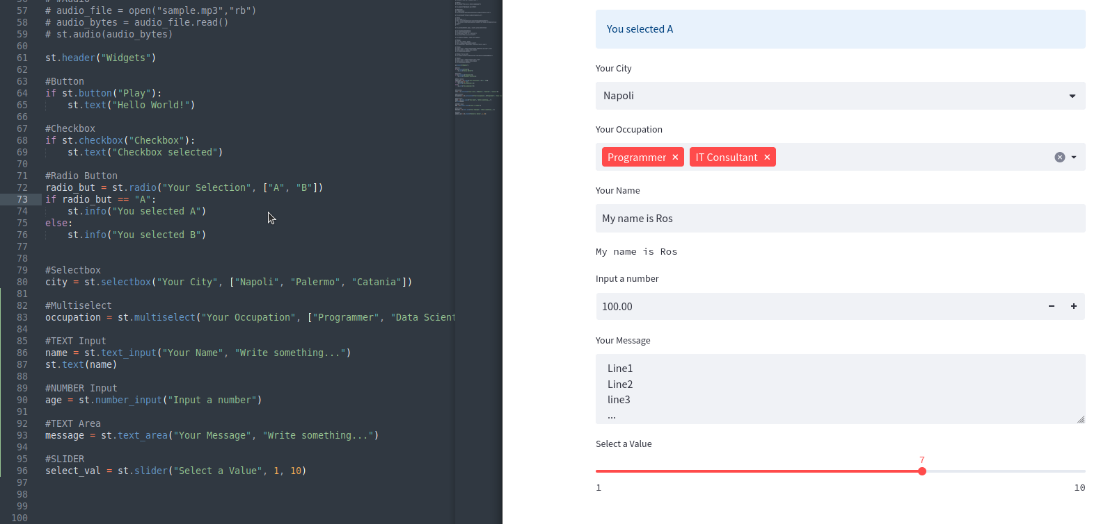


Figure 3.25: st.slider

Slider is quite nice to see and very effective in pursuing its task.

Balloons

A very nice widget is the balloons, think to a situation where you want to show happiness for example after something good happened, in this case you can use it for example clicking on a button as in the following code:

if st.button(“Balloons”):

st.balloons()

Try to discover by yourself what happens after clicking the button!

Dataframes, Plots and Visualizations

It’s time now to deal with dataframes, so let’s comment something and continue with our widget exploration.

Dataframe is the name that the pandas library gives to its data. When a file, for example a csv one, is imported in pandas the result will be a dataframe. We can think to a data frame to something like an Excel or Google Sheet table, so a piece of data made by columns and row. Columns are the features or variables and rows are the records or cases. So to keep things simple we can say that a dataframe is a data structure made by columns and rows.

First of all we need to install pandas, so please do it with:

pipenv install pandas

If you followed our suggestion and are using pipenv for virtual environments or use a simple:

pip install pandas

If you are in a plain, standard Python installation, then write the following code:

st.header(“Dataframes and Tables”)

import pandas as pd

df = pd.read\_csv(“auto.csv”)

st.dataframe(df.head(10))

First of all, please consider that all files used in this book are available in its GitHub repo including the auto.csv file. After that let’s comment the code:

At the beginning we import pandas, a powerful library for data/datasets management, with the alias pd, after that we load the auto.csv file into a variable named df (for dataframe), then we visualize the 10 first rows of the dataframe (it’s head) with the “dataframe” widget.

The result is shown in the following figure:

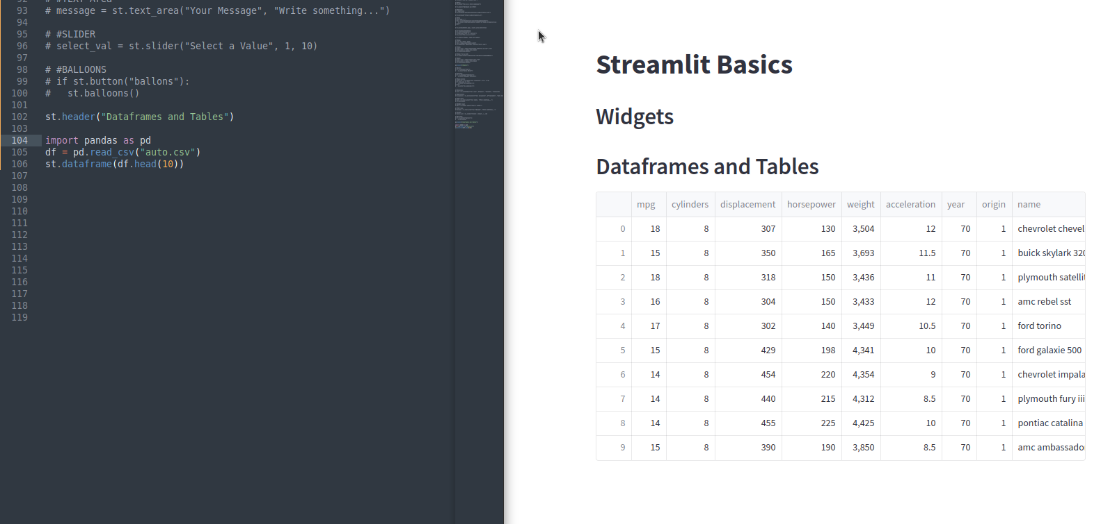


Figure 3.26: st.dataframe

As we can see, thanks to st.dataframe we can visualize the csv file in a very nice format, having all the columns and rows similar to a Microsoft Excel sheet, and if we go in the bottom or in the right part of it we can scroll left/right and up/down, moreover we have also the opportunity to maximize the dataframe!

If we prefer, we can visualize the dataframe in the “table” format since out of the box, we have also a nice “table” widget; in this case, unfortunately, it’s not possible to scroll our data.

So, if we write

st.table(df.head(10))

we’ll get the following table:

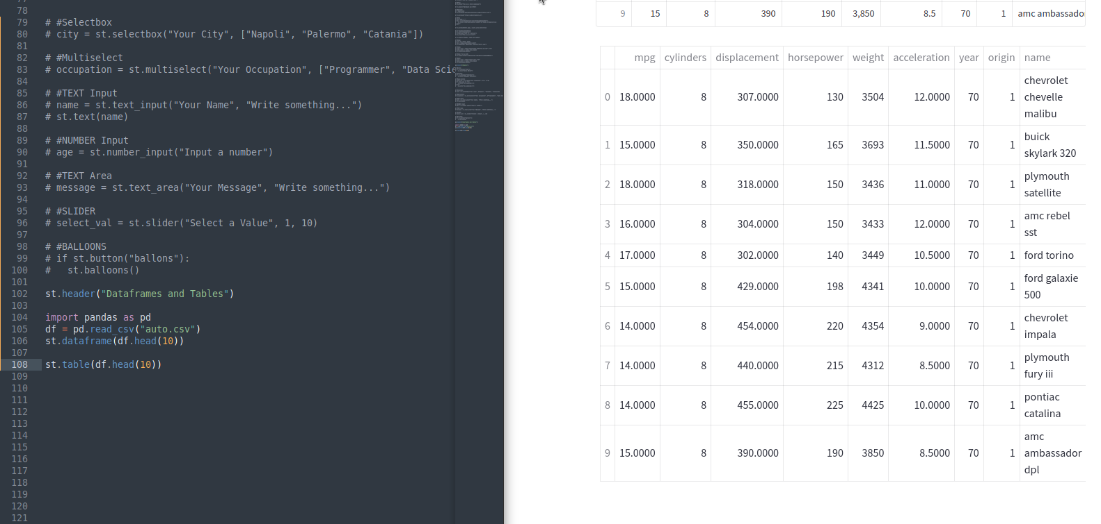


Figure 3.27: st.table

Out of the box, Streamlit is also able to manage plottings, in fact; for example, we can show area, bar and line charts. All we need is just a dataframe.

Automatically Streamlit, together with the plottings, displays the list (in different colors) of all the visualized variables (please note we are using just “mpg” and “cylinders”).

Here, we can use the dataframes we loaded in the previous example.

So, if we write

st.area\_chart(df[[“mpg”,”cylinders”]])

we’ll get the following area chart:

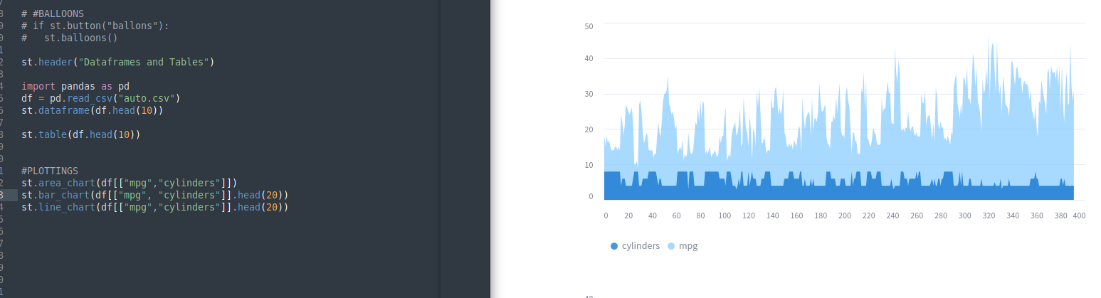


Figure 3.28: st.area\_chart

If we write

st.bar\_chart(df[[“mpg”,”cylinders”]].head(20))

we’ll get the following bar chart:

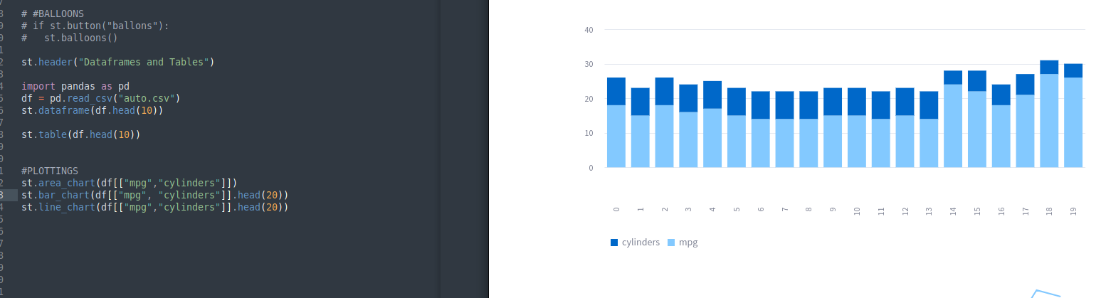


Figure 3.29: st.bar\_chart

And if we write

st.line\_chart(df[[“mpg”,”cylinders”]].head(20))

we’ll get the following line chart:

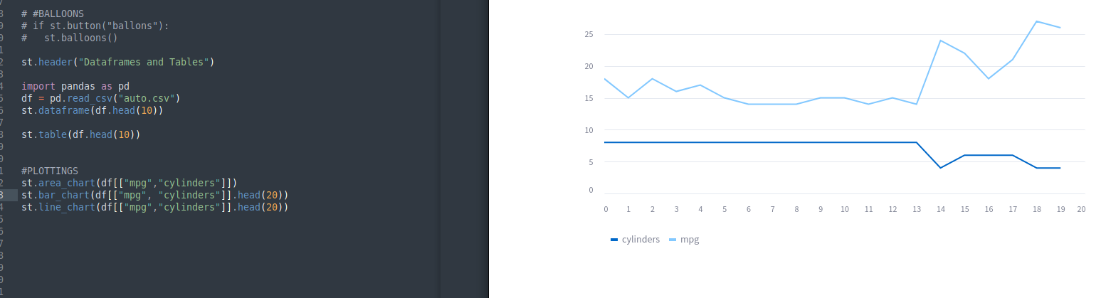


Figure 3.30: st.line\_chart

If we enter in the charts, we can zoom-in and zoom-out and clicking on the three dots in the top right of each plotting, we can save the charts as PNG, view the source and do some other operations.

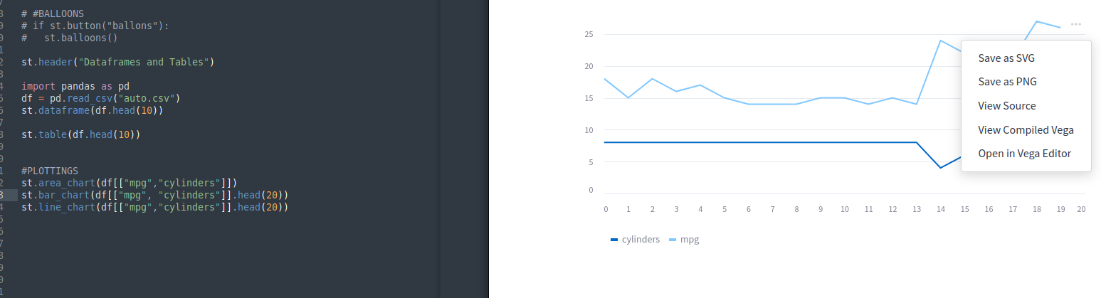


Figure 3.31: Out of the box plotting functions

Finally, exactly as we already saw with dataframes, we can maximize all the charts to see them full screen size.



Figure 3.32: Full screen plotting

With Streamlit we are also able to plot much more beautiful graph using the Python packages matplotlib and seaborn, so first of all as usual let’s install them typing:

pipenv install matplotlib seaborn

Or if we are in a pure Python environment:

pip install matplotlib seaborn

And then import them:

import matplotlib.pyplot as plt

import seaborn as sns

Now we are ready for plotting, for example, an heatmap of the correlation matrix of our dataframe, simply writing:

fig, ax = plt.subplots()

corr\_plot = sns.heatmap(df[[“mpg”,”cylinders”, “displacement”]].corr(), annot= True)

st.pyplot(fig)

The first instruction creates an empty figure, the second line leveraging seaborn creates an heatmap plotting(with annotations) of the correlation matrix coming from the variables in the df dataframe and, finally, the third command plots our figure using matplotlib.pyplot that is directly managed by Streamlit, here is the result:

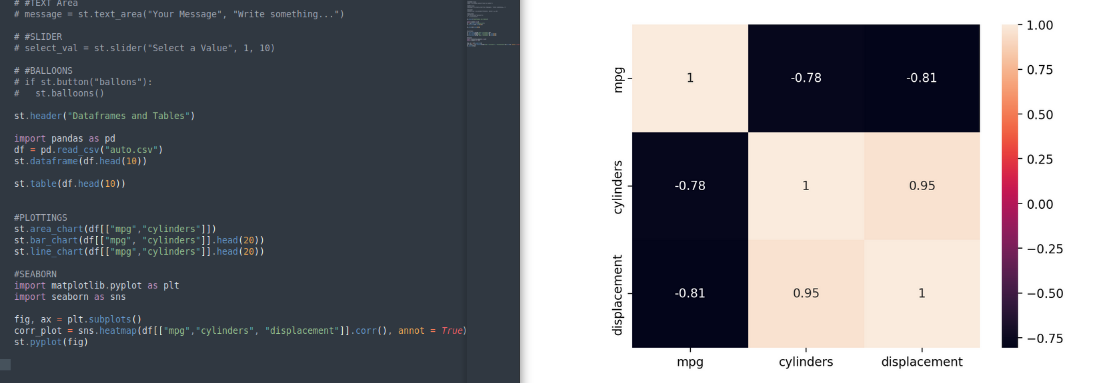


Figure 3.33: heatmap plotting with Seaborn

Please note that we have displayed the correlation plot with labels and the heatmap with colors and annotations.

Date, Time and more

Another very useful element that we can manage always out the box in Streamlit is “Date and Time”, so dates, hours, etc.

For example, to print on the screen today’s date we just have to write:

import datetime

today = st.date\_input(“Today is”,datetime.datetime.now())

Where the first line simply imports datetime package and the second using Streamlit’s date\_input asks to the user to select a date, finally this date will be saved in the today variable.

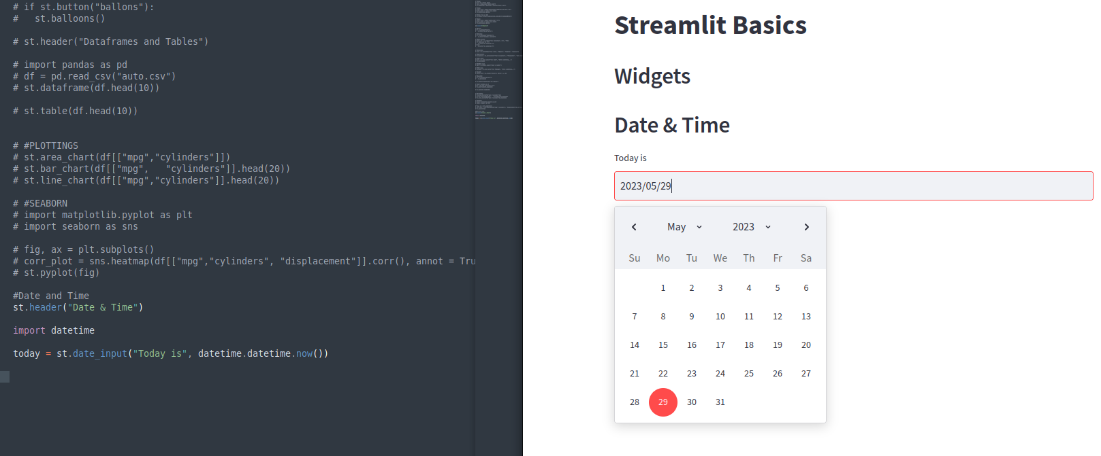


Figure 3.34: st.date\_input

Continuing with “Date and Time”, we can do the same with Time, so let’s write:

import time

hour = st.time\_input(“The time is”,datetime.time(12,30))

This time we are importing time and using time\_input where we specify that time is 12:30, on the screen we can select any time we want:

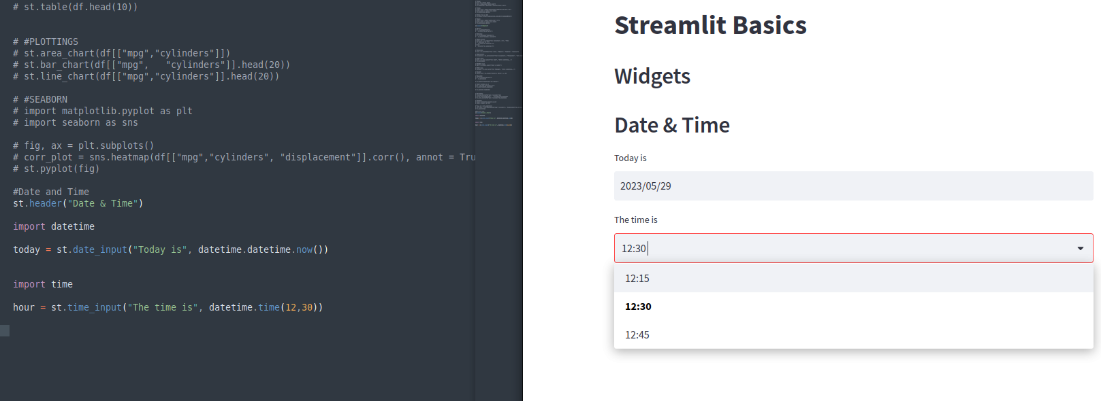


Figure 3.35: st.time\_input

Streamlit it’s really powerful and easy to use, we can even manage immediately the text for example in JSON or programming language format as, for example, Julia or Python.

Typing something like:

data = {“name”:”John”,”surname”:”Wick”}

st.json(data)

We create a variable “data” that contains two couples key:value and then display it on the screen in json format just using the st.json widget, easy and clean:

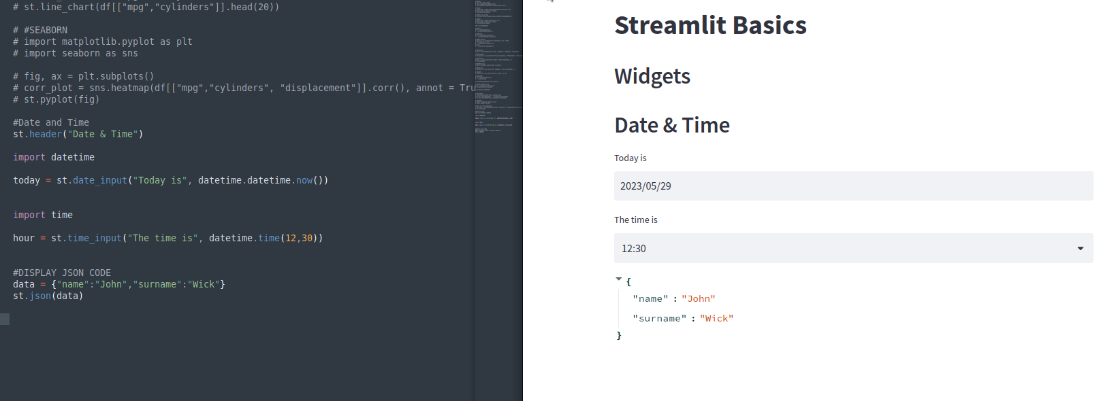


Figure 3.36: st.json

If we click on the arrow we can close/minimize the json.

Displaying code is very easy as well since we have to use st.code specifying the programming language as argument (for Python not necessary since it is the default), so once again:

st.code(“import pandas as pd”)

And we see:

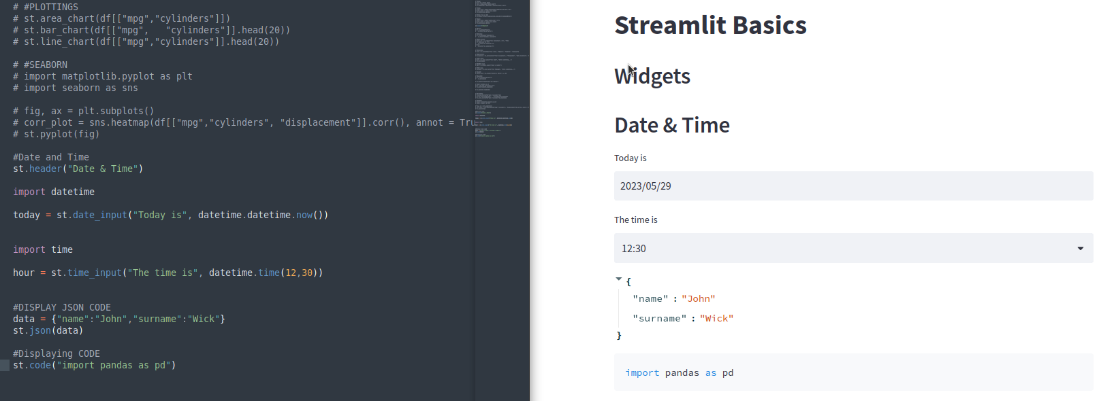


Figure 3.37: st.code for Python

In case of Julia we must specify the programming language, so we can write for example:

julia\_code = “””

function doit(num::int)

println(num)

end

“””

st.code(julia\_code, language=’julia’)

And this is the result:



Figure 3.38: st.code for Julia

As standard widgets we can use also progress bars and spinners, so let’s check how they work.

For example, to create a progress bar that goes from 0 to 100 increasing its value of 1 every 0.1 seconds, we can write:

import time

my\_bar = st.progress(0)

for value in range(100):

time.sleep(0.1)

my\_bar.progress(value+1)

The result it’s really very nice, for a faster bar use a time.sleep(0.01) and for a slower one a time.sleep(1). This is the result:

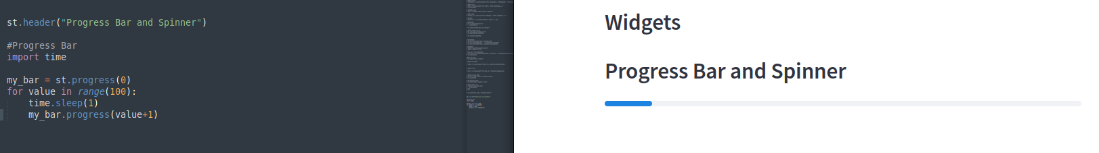


Figure 3.39: st.progress

The spinner works more or less in the same way of the progress bar, so we can write:

import time

with st.spinner(“Please wait...”):

time.sleep(10)

st.success(“Done!”)

Very easily, we can set the tarting message, wait for 10 seconds, like so:

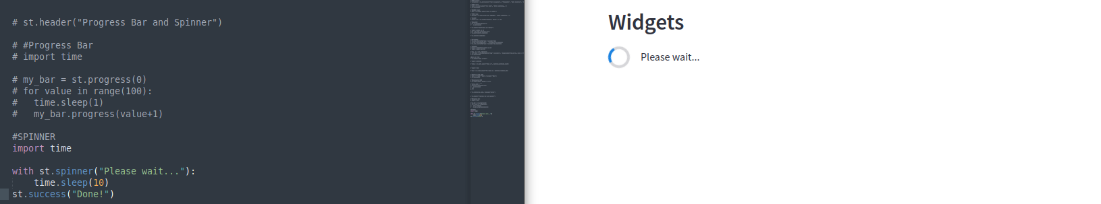


Figure 3.40: st.spinner during the “waiting time”

And finally print in green (success) “Done!”, like so:

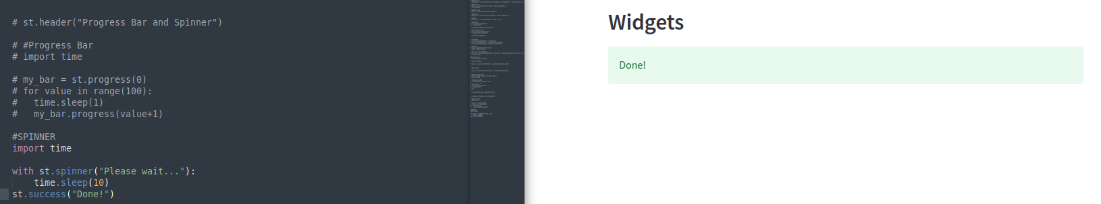


Figure 3.41: st.spinner after “completion”

Really very nice!

With progress bars and spinner we can close this quick introduction of Streamlit’s main functions and widgets, this kind of crash course or full immersion.

Summary

In this chapter we explored all Streamlit’s out-of-the-box main features and widgets. We started creating and empty Python file and launching Streamlit and we saw how to manage its web interface for example using the “rerun” feature and leveraging the real-time updating functionality.

We saw how to deal with text in various ways, in terms of size, colors and format. We also explored the multimedia widgets, images, audio and video.

A lot of elements, such as buttons, checkboxes, radio buttons, etc. have been explained and used.

Many different kind of inputs are supported natively, in fact it’s very easy input text, numbers, dates, time, and so on. Also widgets such as text areas or sliders are immediately out of the box ready to be used.

Data and plottings are extremely easy, we can use dataframes and plot bar, line or area charts with one line of code, even heatmaps clean and neat.

Even formatting the text in a programming language style, such as Python or Julia, is a matter of a couple lines of code.

If we need to wait for some calculation or activities in charge of our application we can use progress bars or spinners to create a nice “wait please...” effect on the screen.

All these components are the basic elements, the toolbox that we are going to use starting from the next Chapter in order to build up our real web applications and doing that we are even extending our knowledge about Streamlit’s more advanced features!