Streamlit - basics

Text and Table elements

```
# Import Streamlit library
import streamlit as st
# Writing Title
st.title("This is our Title")
```

```
# Import Streamlit library
import streamlit as st
# Writing Title with ANCHOR
st.title("This is our Title", anchor="Apress")
```

The anchor functionality is optional and can also be set to None if we are not using it.

```
# Header
st.header("""This is our Header""")
```

```
# sub-header
st.subheader("""This is our Sub-header""")
```

We can use an anchor parameter in st.title(), st. header(), and st.subheader(), but it is optional.

```
# Caption
st.caption("""This is our Caption""")
```

We know that caption text is an explanation that describes notes, footnotes, tables, images, and videos.

```
#Displaying Plain Text
st.text("Hi,\nPeople\t!!!!!!!!!")
st.text('Welcome to')
st.text(""" Streamlit's World""")
```

```
#Displaying Markdown
st.markdown("# Hi,\n# ***People*** \t!!!!!!!!!")
```

```
st.markdown('## Welcome to')
st.markdown("""### Streamlit's World""")
```

LaTeX is formatted text used mainly for technical documentation. Streamlit supports text written in LaTeX format.

```
#Displaying Latex
st.latex(r'''cos2\theta = 1 - 2sin^2\theta''')
st.latex("""(a+b)^2 = a^2 + b^2 + 2ab""")
st.latex(r'''\frac{\partial u}{\partial t}
    = h^2 \left( \frac{\partial^2 u}{\partial x^2}
    + \frac{\partial^2 u}{\partial y^2}
    + \frac{\partial^2 u}{\partial z^2} \right)''')
```

```
# Displaying Python Code
st.subheader("""Python Code""")
code = '''def hello():
     print("Hello, Streamlit!")'''
st.code(code, language='python')
# Displaying Java Code
st.subheader("""Java Code""")
st.code("""public class GFG {
    public static void main(String args[])
    {
        System.out.println("Hello World");
    }
}""", language='javascript')
st.subheader("""JavaScript Code""")
st.code(""" <p id="demo"></p>
<script>
try {
  addalert("Welcome guest!");
}
catch(err) {
  document.getElementById("demo").innerHTML = err.message;
}
</script> """)
```

If we do not add the programming language for the code specified, then the code will appear in an unstyled format, like our code in the HTML language. The language option is not mandatory unless there is more than one language used. We can also copy the code written in the application with a shortcut appearing at top-right corner of the code box.

Dataframes

A dataframe consists of two-dimensional data with rows and columns that act as labeled data. The Streamlit library displays a dataframe in an interactive table format, which means the table is scrollable and

can also change table size dynamically.

```
import streamlit as st
import pandas as pd
import numpy as np
# defining random values in a dataframe using pandas and numpy
df = pd.DataFrame(
    np.random.randn(30, 10),
    columns=('col_no %d' % i for i in range(10)))
st.dataframe(df)
```

We can highlight dataframe objects using minimum, maximum, and null values by using the highlight function.

```
# Highlighting minimum value objects
st.dataframe(df.style.highlight_min(axis=0))
```

Tables

Tables and dataframes are similar in nature, both having rows and columns. But a table displayed in Streamlit is static in nature and is displayed directly on the application page.

```
import streamlit as st
import pandas as pd
import numpy as np
# defining random values in a dataframe using pandas and numpy
df = pd.DataFrame(
    np.random.randn(30, 10),
    columns=('col_no %d' % i for i in range(10)))
# defining data in table
st.table(df)
```

The complete table is displayed in the application instead of the default data that is displayed using a dataframe. This is one of the major differences between a table and a dataframe. The table defined in the streamlit application is static in nature, and there is no option for scrolling and changing the dimensions of the table within the UI like we saw with dataframes.

Metrics

We can display data with indicators using the metric() function. This indicator helps the user see any changes in the data easily. We can define the change in data whether it is positive or negative or neutral. We will now display temperature data in our application by using the metric() function.

```
import streamlit as st
# Defining Metrics
```

```
st.metric(label="Temperature", value="31 °C", delta="1.2 °C")
```

Similarly, we can display any metrics such as speed, rainfall, energy, calories, etc., in the label by assigning application the value and delta value.

```
import streamlit as st
#Defining Columns
c1, c2, c3 = st.columns(3)
# Defining Metrics
c1.metric("Rainfall", "100 cm", "10 cm")
c2.metric(label="Population", value="123 Billions", delta="1 Billions",
delta_color="inverse")
c3.metric(label="Customers", value=100, delta=10, delta_color="off")
st.metric(label="Speed", value=None, delta=0)
st.metric("Trees", "91456", "-1132649")
```

JSON

We know that tabular data application in data science is sometimes in JSON format. JSON format is a way to interchange data. It is easy to read and write for users. When an object is a string, we will assume that it will contain serialized JSON.

```
#Defining Nested JSON
st.json(
{ "Books" :
    [{
    "BookName" : "Python Testing with Selenium",
    "BookID" : "1",
    "Publisher" : "Apress",
    "Year" : "2021",
    "Edition" : "First",
    "Language" : "Python",
    },
    {
        "BookName": "Beginners Guide to Streamlit with Python",
        "BookID" : "2",
        "Publisher" : "Apress",
        "Year" : "2022",
        "Edition" : "First",
        "Language" : "Python"
    }]
} )
```

The write( ) Function as a Superfunction

There is twist in the write() function in that it does more than just display text.

```
import streamlit as st
import pandas as pd
# Dataframe in write function
st.write(pd.DataFrame({
     'column one': [5.436, 6.372, 3.645, 4.554, 7.263],
     'column two': [99, 55, 75, 41, 37],
 }))
```

We will now see one more example with multiple arguments embedded in a write() function.

```
import streamlit as st
import pandas as pd
import numpy as np
df = pd.DataFrame(
     np.random.randn(30, 10),
     columns=('col_no %d' % i for i in range(10)))
# defining multiple arguments in write function
st.write('Here is our Data', df, 'Data is in dataframe format.\n',
"\nWrite is Super function")
```

We have not used a default dataframe definition in the write() function. Here the write() function automatically checks for the value corresponding to the data type and displays it in our application

Finally, we will see how we can use the write() function to display a simple chart in our application.

```
# importing Necessary Libraries
import pandas as pd
import numpy as np
import altair as alt
import streamlit as st
# Defining random Values for Chart
df = pd.DataFrame(
     np.random.randn(10, 2),
     columns=['a', 'b'])
# Defining Chart
chart = alt.Chart(df).mark_bar().encode(
     x='a', y='b',  tooltip=['a', 'b'])
# Defining Chart in write() function
st.write(chart)
```

Magic

Magic is a feature that enables us to define data elements in our application without explicitly stating any command.

```
# Math calculations with no functions defined
"Adding 5 & 4 =", 5+4
# Displaying Variable 'a' and its value a= 5
'a', a
# Markdown with Magic
"""
# Magic Feature
Markdown working without defining its function explicitly.
"""
# Dataframe using magic
import pandas as pd
df = pd.DataFrame({'col': [1,2]})
'dataframe', df
```

We can even display a chart in our application using the magic feature.

```
# Magic working on Charts
import matplotlib.pyplot as plt
import numpy as np
s = np.random.logistic(10, 5, size=5)
chart, ax = plt.subplots()
ax.hist(s, bins=15)
# Magic chart
"chart", chart
```

The magic feature works in the main python source app file, as it is currently not supported during file import.

Visualization

In this chapter, we will learn what data visualization is and explore Streamlit's functions to visualize data. Later, you'll see how flexible is Streamlit with various Python Libraries to visualize data.

With Streamlit, we can develop customized dashboards for visualizing our data. We can also create any kind of visualization tool in Streamlit as it supports all the Python libraries. We will look at a few main libraries in this chapter such as Plotly, Altair, Seaborn, and Matplotlib. We will also discuss various graphs that can be visualized in the Streamlit application.

In Streamlit, we can create visualization dashboards or tool with less code.

Bar

To represent data points in vertical bars, we can use a bar chart. The built-in st.bar_chart() function can plot the data points in a Streamlit application. It is similar to st.altair_chart(), which is defined later in the chapter.

```
import pandas as pd
import numpy as np
st.title('Area')
```

```
# Defining dataframe with its values
df = pd.DataFrame(
    np.random.randn(40, 4),
    columns=["C1", "C2", "C3", "C4"])
# Bar Chart
st.bar_chart(df)
```

Line

The st.line_chart() function automatically detects indices and their values.

```
import pandas as pd
import numpy as np
st.title('Area')
# Defining dataframe with its values
df = pd.DataFrame(
    np.random.randn(40, 4),
    columns=["C1", "C2", "C3", "C4"])
# Bar Chart
st.line_chart(df)
```

The line chart is created by passing the dataframe to it. The values of four columns are represented using four different colors in the chart. The x-axis defines the row number of each column, and the y-axis defines the values in that row.

Area

An area graph or chart is based on the line chart shown in the previous section. The area chart is used to compare different quantitative values of one or more columns.

```
import pandas as pd
import numpy as np
st.title('Area')
# Defining dataframe with its values
df = pd.DataFrame(
    np.random.randn(40, 4),
    columns=["C1", "C2", "C3", "C4"])
# Bar Chart
st.area_chart(df)
```

Map

To display data points on a map, we will be using the st.map() function of Streamlit. A scatter plot is applied on top of a map provided by mapbox. The latitude and longitude specify the position of the map and where the scatter plot is plotted.

```
import pandas as pd
import numpy as np
st.title('Map')
# Defining Latitude and Longitude
locate_map = pd.DataFrame(
  np.random.randn(50, 2)/[10,10] + [15.4589, 75.0078],
  columns = ['latitude', 'longitude'])
# Map Function
st.map(locate_map)
```

Graphviz

Graphviz is an open-source Python library in which we can create graphical objects having nodes and edges. The language of nodes and edges is known as the DOT language.

```
import graphviz as graphviz
st.title('Graphviz')
# Creating graph object
st.graphviz_chart('''
    digraph {
        "Training Data" -> "ML Algorithm"
        "ML Algorithm" -> "Model"
        "Model" -> "Result Forecasting"
        "New Data" -> "Model"
} ''')
```

The nodes and edges are added in the DOT language, which is easy to understand. We have defined four nodes with connecting edges. We can define the same edge and nodes as shown here:

```
import graphviz as graphviz
st.title('Graphviz')
# Create a graphlib graph object
graph = graphviz.Digraph()
graph.edge('Training Data', 'ML Algorithm')
graph.edge('ML Algorithm', 'Model')
graph.edge('Model', 'Result Forecasting')
graph.edge('New Data', 'Model')
st.graphviz_chart(graph)
```

We can use this Graphviz code to visualize various ML algorithms or neural networks to help us understand the architecture or flows encountered in them while processing data.

Seaborn

Seaborn is a Python visualization library used to get beautifully styled graphs in various color palettes, making graphs more attractive.

Count

The count graph is used to represent a number of occurrences or counts of a categorical variable.

```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
# Data Set
df = pd.read_csv("./files/avocado.csv")
# Defining Count Graph/Plot
fig = plt.figure(figsize=(10, 5))
sns.countplot(x = "year", data = df)
st.pyplot(fig)
```

The categorical data used is year, where the number of occurrences is counted to plot the graph. The count_plot() function is used from the Seaborn library to display the count graph.

Violin

To represent numerical data for more than one group, we can use a violin chart or graph that uses density curves.

```
import streamlit as st
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
# Data Set
df = pd.read_csv("./files/avocado.csv")
# Defining Violin Graph
fig = plt.figure(figsize=(10, 5))
sns.violinplot(x = "year", y="AveragePrice", data = df)
st.pyplot(fig)
```

We have used the violinplot() method to display a violin graph with the x and y parameters defined. The larger the frequency, the larger the width of the curve in the violin region

Strip

A strip graph represents a summarized univariate dataset. To display all the observations, we can use a strip graph.

```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
# Data Set
df = pd.read_csv("./files/avocado.csv")
# Defining Strip Plot
```

```
fig = plt.figure(figsize=(10, 5))
sns.stripplot(x = "year", y="AveragePrice", data = df)
st.pyplot(fig)
```

Altair

An altair is one of the most used statistical visualization python library based on Vega. The visualization process is handled by the Altair library once we define values for x and y. We can also limit the size and set the color of the graphs that are displayed in the application. At first we need to install the Altair library. Altair supports all type of charts/graphs, but we will discuss only a few types of graphs.

Boxplot

We can use a boxplot to represent data in terms of skewness by showing data quartiles and averages.

```
import altair as alt
import streamlit as st
import pandas as pd
#Read albany Dataset
df = pd.read_csv("./files/albany.csv")
# Box Plot
box_plot = alt.Chart(df).mark_boxplot().encode(
x = "Date",
    y = "Large Bags"
)
st.altair_chart(box_plot)
```

Area

As we know, an area graph or plot is displayed to represent quantitative data based on line charts.

```
import altair as alt
import streamlit as st
import pandas as pd
#Read albany Dataset
df = pd.read_csv("./files/albany.csv")
# Area Plot
area = alt.Chart(df).mark_area(color="orange").encode(
x = "Date",
    y = "Large Bags"
)
st.altair_chart(area)
```

Heatmap

A heatmap is graphical representation of values in color. The values determine the intensity of color corresponding to two dimensions.

```
import altair as alt
import streamlit as st
import pandas as pd
#Read albany Dataset
df = pd.read_csv("./files/albany.csv")
heat_map  = alt.Chart(df).mark_rect().encode(
        alt.Y('AveragePrice:Q'),
        alt.X('Large Bags:Q'),
        alt.Color('AveragePrice:Q'),
        tooltip = ['AveragePrice', 'type', 'Large Bags', 'Date']
    ).interactive()
st.altair_chart(heat_map)
```

Plotly

Plotly is a Python library that can make high-quality interactive graphs. It supports different types of graphs such as bar charts, line charts, boxplots, pie charts, scatter plots, histograms, etc. We will see how this Python library can be used in our Streamlit application.

Pie

For the Avocado dataset, we will be visualizing a percentage of organic and conventional avocados.

```
import streamlit as st
import plotly.graph_objects as go
import pandas as pd
#Read Avocado Dataset
data = pd.read_csv("./files/avocado.csv")
st.header("Pie Chart")
# Implementing Pie Plot
pie_chart = go.Figure(
    go.Pie(
    labels = data.type,
    values = data.AveragePrice,
    hoverinfo = "label+percent",
    textinfo = "value+percent"
))
st.plotly_chart(pie_chart)
```

There are two more optional parameters named hoverinfo and textinfo where information is displayed when the mouse is hovered on the pie chart. The function plotly_chart allows us to add a defined pie chart in our application. Our pie chart is divided into two parts: organic and conventional. When the mouse is hovered over a part, we can see the name and percentage of that particular section of pie chart. The percentage and total number of avocados corresponding to organic and conventional are displayed in different color.

Donut

Plotly has created a different submodule to plot donut charts. The name of the module is express(). We need to import the express() module in our code.

```
import streamlit as st
import pandas as pd
import plotly.express as px
#Read Avocado Dataset
data = pd.read_csv("./files/avocado.csv")
st.header("Donut Chart")
# Donut Chart
donut_chart = px.pie(
    names = data.type,
    values = data.AveragePrice,
    hole=0.25,
)
st.plotly_chart(donut_chart)
```

We have provided same dataset values that are used for pie charts in our earlier example. The pie() method in the Express module takes three parameters: names, values, and hole. The hole specifies the radius of the hole in a donut. We can convert a donut into a pie chart without specifying a hole parameter.

Scatter

We will now implement a scatter plot in our application by using the scatter() method from the Plotly library.

```
import streamlit as st
import pandas as pd
import plotly.express as px
#Data Set
data = pd.read_csv("./files/avocado.csv")
st.header("Scatter Chart")
#Scatter
scat = px.scatter(
    x = data.Date,
    y = data.AveragePrice
)
st.plotly_chart(scat)
```

Line

We can create a line graph using the line() method from the Express module in Plotly. The x- and y-axes are the parameters used in a line() function.

```
import pandas as pd
import plotly.express as px
```

```
# Data Set
data = pd.read_csv("./files/avocado.csv")
# Minimizing Dataset
albany_df = data[data['region']=="Albany"]
al_df = albany_df[albany_df["year"]==2015]
#Line
line_chart = px.line(
    x = al_df["Date"],
    y = al_df["Large Bags"]
)
st.header("Line Chart")
st.plotly_chart(line_chart)
```

Bar

We can represent a bar graph using the express.bar() function. Similar to a line chart, a bar graph also has x and y parameters with an optional title parameter.

```
import pandas as pd
import plotly.express as px
# Data Set
data = pd.read_csv("./files/avocado.csv")
# Minimizing Dataset
albany_df = data[data['region']=="Albany"]
al_df = albany_df[albany_df["year"]==2015]
# Bar graph
bar_graph = px.bar(
    al_df,
    title = "Bar Graph",
    x = "Date",
    y = "Large Bags"
    )
st.plotly_chart(bar_graph)
```

We can also change the color by introducing the color parameter in the bar() function, as shown here:

```
#Bar Color
bar_graph = px.bar(
    x = al_df["Date"],
    y = al_df["Large Bags"],
    title = "Bar Graph",
    color=al_df["Large Bags"]
)
```

The color intensity is associated with the value defined in it. The color intensity in the bar graph helps to understand the range of values depicted.

Bar Horizontal

We can flip the bars in a bar graph to be horizontal. To flip a bar graph to be horizontal, we need to specify the orientation parameter in the bar() function.

```
import streamlit as st
import pandas as pd
import plotly.express as px
# Data Set
data = pd.read_csv("./files/avocado.csv")
# Minimizing Dataset
albany_df = data[data['region']=="Albany"]
al_df = albany_df[albany_df["year"]==2015]
# Horizontal Bar Graph
bar_graph = px.bar(
    al_df,
    x = "Large Bags",
    y = "Date",
    title = "Bar Graph",
    color="Large Bags",
    orientation='h'
)
```

Subplots

Subplots are used to load different graphs next to each other. In Plotly, we can use the make_subplots() method to create multiple graphs to be placed next to one other or to be stacked one below the other. To define subplots, we need to import the potly.subplots module.

```
import streamlit as st
import pandas as pd
import plotly.graph_objects as go
from plotly.subplots import make_subplots
# Data Set
data = pd.read_csv("./files/avocado.csv")
# Minimizing Dataset
albany_df = data[data['region']=="Albany"]
al_df = albany_df[albany_df["year"]==2015]
fig = make_subplots(rows=3, cols=1)
# First Subplot
fig.add_trace(go.Scatter(
    x=al_df["Date"],
    y=al_df["Total Bags"],
), row=1, col=1)
# Second SubPlot
fig.add_trace(go.Scatter(
    x=al_df["Date"],
    y=al_df["Small Bags"],
), row=2, col=1)
```

```
# Third SubPlot
fig.add_trace(go.Scatter(
   x=al_df["Date"],
   y=al_df["Large Bags"],
), row=3, col=1)
st.plotly_chart(fig)
```

Data and Media Elements

Images

```
# Displaying Image by specifying path
st.image("files/animal7.jpg")
```

We can also use any Python library such as OpenCV, PIL, etc., that supports opening an image. It is not compulsory to use any one library. Later, we need to pass the same image file stored in a variable in the image() function of Streamlit.

```
# Displaying Image URL
st.image("https://tinyurl.com/322vu3ab")
```

The syntax for the image() function with other parameters is as follows: st.image(image, caption, width, use_column_width, clamp, channels, output_format)

Multiple images

```
animal_images = [
    'files/animal1.jpg',
    'files/animal2.jpg',
    'files/animal3.jpg'
]
# Displaying Multiple images with width 150
st.image(animal_images, width=150)
```

Background image

```
import base64
# Function to set Image as Background
def add_local_backgound_image_(image):
    with open(image, "rb") as image:
        encoded_string = base64.b64encode(image.read())
    st.markdown(
    f"""
    <style>
    .stApp {{
        background-image: url(data:files/
{"jpg"};base64,{encoded_string.decode()});
        background-size: cover
    }}
    </style>
    """,
    unsafe_allow_html=True
    )
# Calling Image in function
add_local_backgound_image_('files/animal7.jpg')
```

Resizing an Image

In machine learning, images are trained at a uniform level to extract features from them and obtain an image model. When models are deployed and we want to test our model with new images, then the images need to be resized to the size of the trained model. Hence, we will see how the resizing of an image is done. The same will be applied while image uploads are used.

```
from PIL import Image
original_image = Image.open("files/animal9.jpg")
# Display Original Image
st.title("Original Image")
st.image(original_image)
# Resizing Image to 600*400
resized_image = original_image.resize((600, 400))
#Displaying Resized Image
st.title("Resized Image")
st.image(resized_image)
```

To resize an image, we used the PIL Python library. We can use any Python library that supports image resizing in our application.

Audio

```
# Open Audio using filepath with filename
sample_audio = open("files/audio.wav", "rb")
#Reading Audio File
audio_bytes = sample_audio.read()
```

```
# Display Audio using st.audio() function with start time set to 20
st.audio(sample_audio, start_time = 20)
```

```
# Open Audio using filepath with filename and read the audio file
sample_url = st.audio("https://www.learningcontainer.com/wp-
content/uploads/2020/02/Kalimba.mp3")
```

Streamlit supports most of the audio formats like MP3, WAV, OGG, etc. When no audio format is specified, then Streamlit will automatically take audio in audio/wav format.

Video

We can upload any video to our application using the st.video() function. Similar to audio files, we can specify the starting time of the video uploaded instead of the default time starting from 0. Here, the integer value specifies the time in seconds.

```
# Open Video using filepath with filename and read the video file
sample_video = open("files/ocean.mp4", "rb").read()
# Display Video using st.video() function
st.video(sample_video, start_time = 10)
```

```
# Displaying Video using youtube URL
st.video("https://www.youtube.com/watch?v=OMkEVX23BdM")
```

Emojis

```
# Emojis with/without shortcodes
emojis = """:rain_cloud: :coffee: :love_hotel: :couple_with_heart: """
# Displaying Shortcodes
st.title(emojis)
```

You can check for all emoji shortcodes that are available for Streamlit application at https://streamlit-emoji-shortcodes-streamlit-app- gwckff.streamlitapp.com/. You can also display any other emoji that is not available using other emoji sites.

Buttons and sliders

Here we will discuss how to create different types of buttons. These buttons are a medium for interaction between the user and the application.

Buttons

```
st.title('Creating a Button') # Defining a Button
button = st.button('Click Here')
if button:
    st.write('You have clicked the Button')
else:
     st.write('You have not clicked the Button')
```

Radio Buttons

```
st.title('Creating Radio Buttons')
# Defining Radio Button
gender = st.radio(
    "Select your Gender",
    ('Male', 'Female', 'Others'))
if gender == 'Male':
    st.write('You have selected Male.')
elif gender == 'Female':
    st.write("You have selected Female.")
else:
    st.write("You have selected Others.")
```

By default, the first radio button is selected, but you can change it by using indexing.

```
st.title('Creating Radio Buttons')
gender = ('Male', 'Female', 'Others')
# Defining Radio Button with index value
gender = st.radio(
    "Select your Gender",
    gender,
    index = 1)
if gender == 'Male':
    st.write('You have selected Male.')
elif gender == 'Female':
    st.write("You have selected Female.")
else:
    st.write("You have selected Others.")
```

Check Boxes

```
st.title('Creating Checkboxes')
st.write('Select your Hobies:')
# Defining Checkboxes
check_1 = st.checkbox('Books')
check_2 = st.checkbox('Movies')
check_3 = st.checkbox('Sports')
```

```
st.title('Pre-Select')
check = st.checkbox('Accept all Terms and Conditions***', value=True)
```

To preselect a checkbox, we need to add the Boolean value True for the value attribute.

Drop-Downs

A drop-down contains options listed within it that are revealed when it is clicked by the user.

```
st.title('Creating Dropdown')
# Creating Dropdown
hobby = st.selectbox('Choose your hobby: ',
        ('Books', 'Movies', 'Sports'))
```

We can change the default option selected by providing the index value.

```
hobby = st.selectbox('Choose your hobby: ', ('Books', 'Movies', 'Sports'),
index=1)
```

Multiselects

```
# Defining Multi_Select with Pre-Selection
hobbies = st.multiselect(
       'What are your Hobbies',
       ['Reading', 'Cooking', 'Watching Movies/TV Series', 'Playing',
'Drawing', 'Hiking'],
       ['Reading', 'Playing'])
```

In this multiselect code, we have preselected two hobbies. One can skip the preselection by removing the selected options from the code.

```
hobbies = st.multiselect(
      'What are your Hobbies',
      ['Reading', 'Cooking', 'Watching Movies/TV Series', 'Playing',
'Drawing', 'Hiking']
```

Download buttons

When we allow a user to download results or a file directly from our application, Streamlit provides a download button for the user to click. The file format can be JPG/PNG, TXT, CSV, etc.

```
# Creating Download Button
down_btn = st.download_button(
        label="Download Image",
        data=open("./files/fam.jpg", "rb"),
        file_name="lions.jpg",
        mime="image/jpg"
)
```

Similarly, we can use the download button to download a CSV file. In some cases, we can convert the dataframe analyzing the results to CSV by using a download button.

```
st.download_button(
     label="Download CSV",
     data=open("./files/avocado.csv", "rb"),
     file_name='data.csv',
     mime='csv',
)
```

Progress Bars

We have seen progress bars when downloading something from the Web. We can use it in the same way to let the user know that something is still downloading.

```
import time
st.title('Progress Bar')
# Defining Progress Bar
download = st.progress(0)
for percentage in range(100):
    time.sleep(0.1)
    download.progress(percentage+1)
st.write('Download Complete')
```

Spinners

A spinner is used to display a message while the user is waiting for some task to execute in our application. Specifically, we can use a spinner while uploading data that is being processed by the application. The message is temporary in nature. A spinner is often used when results are being processed in an application.

```
# Defining Spinner
with st.spinner('Loading...'):
    time.sleep(5)
st.write('Hello Data Scientists')
```

Forms

Text Box

```
name = st.text_input("Enter your Name")
st.write("Your Name is ", name)
```

We can limit the characters to be entered in a text box by using the max_chars parameter.

```
name = st.text_input("Enter your Name", max_chars=10)
```

We can use the same text box as a password box that hides text data because it's a password.

```
password = st.text_input("Enter your password", type='password')
```

We can also implement a character limit on the password input box. We can see the password by clicking the eye icon at the end of the text box.

Text area

```
# Creating Text Area
input_text = st.text_area("Enter your Review")
# Printing entered text
st.write("""You entered:  \n""",input_text)
```

Number input

In this input type, a user can enter only numbers in the input box.

```
st.number_input("Enter your Number")
```

We can subtract or add numbers by using the - and + provided beside the number box. The numbers can be decimals or integers. We can also limit the number range by specifying the starting and ending number. We can also set the default value if the user does not specify any value. There is one more parameter in the number_input() function known as step_size.

```
# Create number input
num = st.number_input("Enter your Number", 0, 10, 5, 2)
st.write("Min. Value is 0,  \n  Max. value is 10")
st.write("Default Value is 5,  \n  Step Size value is 2")
st.write("Total value after adding Number entered with step value is:",
num)
```

Time

```
st.title("Time")
# Defining Time Function
st.time_input("Select Your Time")
```

We cannot set a time before the current time. The time provided by this function is one hour in the future from the current time, with a time gap of 15 minutes.

Date

The date helps us to define the date with the year, month, and day, respectively. We can also change the default format of the year, month, and date. The date can be picked with a calendar. We can also set on which date the calendar starts and ends by specifying the min_value and max_value values in the date_input() function.

```
st.date_input("Select Date")
```

```
import datetime
st.title("Date")
# Defining Time Function
st.date_input("Select Your Date", value=datetime.date(1989,
12, 25),
    min_value=datetime.date(1987, 1, 1),
    max_value=datetime.date(2005, 12, 1))
```

Color

Streamlit provides a unique color selection box as an input feature. By using the color widget, we can select any color of our choice. The color codes are hexadecimal numbers used to define the color.

```
# Defining color picker
color_code = st.color_picker("Select your Color")
st.header(color_code)
```

File Upload

Sometimes users need to upload files in an ML/DS application for analysis from the local storage. Files might be used for object detection/recognition in an image, forecasting from CSV data, text detection/summarization (NLP technique) from text files, etc.

Uploading a file is one way to input data in an application. The data may be in the form of text file, CSV file, or image. We will see how we can upload such diverse data into our application and also store it in a specific directory. We will begin with text documents.

A file that contains only textual data in it is considered to be a text document. A text file can be created using a text editor like Notepad. We will see how a text document, specifically a .docx file, can be uploaded in our application. To view the contents of a .docx file, we will be using the docx2txt Python library. There are other similar libraries that can be used.

Text and docx

```
import docx2txt
st.title("DOCX & Text Documents")
# Defining File Uploader Function in a variable
text_file = st.file_uploader("Upload Document", type=["docx","txt"])
# Button to check document details
details = st.button("Check Details")
# Condition to get document details
if details:
    if text_file is not None:
        # Getting Document details like name, type and size
        doc_details = {"file_name":text_file.name,
"file_type":text_file.type,
                       "file_size":text_file.size}
        st.write(doc_details)
        # Check for text/plain document type
        if text_file.type == "text/plain":
            # Read document as string with utf-8 format
            raw_text = str(text_file.read(),"utf-8")
            st.write(raw_text)
        else:
            # Read docx document type
            docx_text = docx2txt.process(text_file)
            st.write(docx_text)
```

The size limit specified by Streamlit is 200 MB and cannot be uploaded beyond it. The document type to be uploaded can also be seen in the application. We will be displaying file details such as the name of the uploaded file, its format type, and its size. These details after the user clicks the Check Details button.

PDF

PDF is one of the most commonly used data types. It contains textual and image data, which makes it different from the plain .docx file shown earlier. We will upload the file and display the content in it.

```
import pdfplumber
st.title("PDF File")
pdf_file = st.file_uploader("Upload PDF", type=["pdf"])
details = st.button("Check Details")
if details :
```

```
    if pdf_file is not None:
        pdf_details = {"filename":pdf_file.name,
        "filetype":pdf_file.type,
                       "filesize":pdf_file.size}
        st.write(pdf_details)
        pdf = pdfplumber.open(pdf_file)
        pages = pdf.pages[0]
        st.write(pages.extract_text())
    else:
        st.write("No PDF File is Uploaded")
```

We have installed the pdfplumber Python library using the pip command (pip install pdfplumber) in the environment defined for Streamlit.

CSV

In the next section, we will discuss the tabular data format that can be uploaded in our application. We will now discuss data that is available in tabular format and that is the CSV type. CSV documents contain textual data separated by comma, making a table- or column-like structure. We will see how to upload such files and visualize the data in tables.

```
import pandas as pd
st.title("CSV Data")
data_file = st.file_uploader("Upload CSV",type=["csv"])
details = st.button("Check Details")
if details:
    if data_file is not None:
        file_details = {"file_name":data_file.name, "file_
        type":data_file.type,
                       "file_size":data_file.size}
        st.write(file_details)
        df = pd.read_csv(data_file)
        st.dataframe(df)
    else:
        st.write("No CSV File is Uploaded")
```

We have used the Pandas library to display the results in table format for the uploaded CSV file. You can download this data from Kaggle (https://www.kaggle.com/datasets/neuromusic/avocado-prices). We have defined only the CSV type to be uploaded.

We can see the contents and complete details of the CSV file uploaded by clicking the Check Details button.

Image upload

```
from PIL import Image
import io
st.title("Upload Image")
```

```
image_file = st.file_uploader("Upload Images", type=["png","jpg","jpeg"])
check_details = st.button("Check Details")
if check_details:
    if image_file is not None:
            # To See details
            image_details = {"file_name":image_file.name,
            "file_type":image_file.type,
                           "file_size":image_file.size}
            st.write(image_details)
            # To View Uploaded Image
            image_data = image_file.read()
            image = Image.open(io.BytesIO(image_data))
            st.image(image, width=250)
    else:
        st.write("No Image File is Uploaded")
```

Uploading multiple images

We can also upload multiple images at the same time to get results or train a model.

```
import io
from PIL import Image
uploaded_files = st.file_uploader("Multiple Image Uploader", type=
['jpg','jpeg','png'],
                    help="Upload Images in jpg, jpeg, png
                    format", accept_multiple_files=True,)
details = st.button("Check Details")
for uploaded_file in uploaded_files:
    if details:
        if uploaded_file is not None:
            bytes_data = uploaded_file.read()
            image = Image.open(io.BytesIO(bytes_data))
            st.write("file_name:", uploaded_file.name)
            st.image(image, width=100)
        else:
            st.write("No Image File is Uploaded")
            break
```

The parameter accept_multiple_files in st.upload() is a Boolean value that allows our application to accept multiple files to be uploaded when set as True. When the parameter is not defined, the function will accept only a single file, which is the default function.

Saving Uploaded Documents

In some cases, we need to use the uploaded images for training the model or provide the results associated with them. For this, we need the images to be saved in a directory that can be later reused by our application. We will first get an image using the file_uploader() function and then provide a directory/folder path for it to be saved in.

```
from PIL import Image
import os
import io
# Defining File Upload Method of Streamlit
st.title("Saving File to Directory")
image_file = st.file_uploader("Upload Images",
            type=["png","jpg","jpeg"])
# Defining path where file to be saved
file_save_path = "F:/Books/Apress Streamlit/Chapters/02 Codes/
chapter 6 forms/files"
save_file = st.button("Check Details & Save")
if save_file:
    if image_file is not None:
            # To See details
            image_details = {"file_name":image_file.name,
            "file_type":image_file.type,
                            "file_size":image_file.size}
            st.write(image_details)
            # To View Uploaded Image
            image_data = image_file.read()
            image = Image.open(io.BytesIO(image_data))
            st.image(image, width=250)
            with open(os.path.join(file_save_path,image_file.
            name),"wb") as f:
                f.write((image_file).getbuffer())
            st.success("Image Saved Successfully")
    else:
        st.write("No Image File is Uploaded")
```

We need to specify file path where the user-uploaded image will be stored. We need to use a relative path for the uploaded image to be saved through our application. The variable file_save_path has been defined in our code. Later the file to be uploaded will be saved when we click the Check Details & Save button in our application.

We will receive an “Image Saved Successfully” notification once the image is saved1, into our specified directory. The directory shows where the uploaded image has been stored. We can use the same method to store various file documents in the directory.

If a file with the same name exists in the specified dir or folder it will override the existing one.

Submit button

When input form data is changed by the user, the application needs to rerun, causing a bad user experience. To solve this issue of the application rerunning every time the user makes changes in the data, we can use the form_submit_button() function.

```
my_form = st.form(key='form')
my_form.text_input(label='Enter any text')
```

```
# Defining submit button
submit_button = my_form.form_submit_button(label='Submit')
```

When we hit the Submit button, the application reruns. It is similar to the st.button() function but differs in functionality. The Submit button posts the state of the widgets in batches stored in a form. The st.form_submit_button should be associated with a form or Streamlit will enter an error state.

Columns and Navigation

We will discuss how to set up columns, layouts, and navigation into which we can insert elements. These will help to divide our application into grids. This type of navigation is used when we need more than one page in our application. We will also explore how to use sidebars.

Columns

We will be using the st.columns() function to define columns, which is built in to Streamlit. The columns created are responsive in nature, which means when an application is viewed on different devices, the columns change their size accordingly.

```
import streamlit as st
#Defining Columns
col1, col2 = st.columns(2)
# Inserting Elements in column 1
col1.write("First Column")
col1.image("files/fam.jpg")
# Inserting Elements in column 2
col2.write("Second Column")
col2.image("files/fam.jpg")
```

Spaced-out columns

We can set a column's size by specifying its width. The column with the maximum width is known as a spaced-out column.

```
import streamlit as st
from PIL import Image
img = Image.open("files/fam.jpg")
st.title("Spaced-Out Columns")
# Defining two Rows
for _ in range(2):
    # Defining no. of columns with size
    cols = st.columns((3, 1, 2, 1))
    cols[0].image(img)
    cols[1].image(img)
    cols[2].image(img)
    cols[3].image(img)
```

In this listing, we have defined four columns with two rows that have images in them. The first column is the widest, followed by the third column, and the other two columns have the same size. The first and third columns are said to be spaced-out as they are wider compared to the second and fourth columns. We can specify any number of columns with spaced-out columns. A tuple can also be used in the st.streamlit() function to specify the width of each column.

Columns with padding

Padding is added to create extra space between two columns. padding is an empty column placed between two columns to create a space there.

```
import streamlit as st
from PIL import Image
img = Image.open("files/fam.jpg")
st.title("Padding")
# Defining Padding with columns
col1, padding, col2 = st.columns((10,2,10))
with col1:
    col1.image(img)
with col2:
    col2.image(img)
```

Grids

Grids can be created using the st.columns() function as shown earlier. Creating multiple columns in a loop will get us a grid. Grids are used to align the elements inside them using size and position. In our example, we have defined (4 * 4) rows and columns with the same size. Grids can be used for an image gallery or a recommendation system for books, music, movies, etc., where we need to specify a large amount of data in a grid format on a single page.

```
import streamlit as st
from PIL import Image
img = Image.open("files/fam.jpg")
st.title("Grid")
# Defining no of Rows
for _ in range(4):
    # Defining no. of columns with size
    cols = st.columns((1, 1, 1, 1))
    cols[0].image(img)
    cols[1].image(img)
    cols[2].image(img)
    cols[3].image(img)
```

Expanders/Accordions

When we want to hide additional information from the user or don't want information to always appear on our application, we can use accordions. These are also known as expanders as when the user toggles an

expander into its open state, it expands and displays the extra information. We can use the write() function to add information in accordions or expanders.

```
import streamlit as st
st.title('Expanders')
# Defining Expanders
with st.expander("Streamlit with Python"):
    st.write("Develop ML Applications in Minutes!!!!")
```

Containers

To insert more than one element, we can use a container in Streamlit. A container is invisible in nature and can be defined with st.container(). We have used a with statement in the st.container() to insert multiple elements in the container.

```
import streamlit as st
import numpy as np
st.title("Container")
with st.container():
   st.write("Element Inside Contianer")
   # Defining Chart Element
   st.line_chart(np.random.randn(40, 4))
st.write("Element Outside Container")
```

We can insert elements in the container out of order, which can be done by using the container.write() function. The order of inserting elements varies and hence is called out of order. This helps in creating more flexible applications.

```
import streamlit as st
import numpy as np
st.title("Out of Order Container")
# Defining Containers
container_one = st.container()
container_one.write("Element One Inside Container")
st.write("Element Outside Container")
# Now insert few more elements in the container_one
container_one.write("Element Two Inside Container")
container_one.line_chart(np.random.randn(40, 4))
```

Empty containers

As the name suggests, the container is empty, and we can insert only one element into it.

```
import streamlit as st
import time
```

```
# Empty Container
with st.empty():
    for seconds in range(5):
        st.write(f"⏳ {seconds} seconds have passed")
        time.sleep(1)
    st.write("✔ Time is up!")
```

Sidebars

A sidebar is a pane that is displayed on the side of the application. It allows the user to stay focused on the main content. We will use the st.sidebar() function to define a sidebar in our application.

```
import streamlit as st
# Sidebar
st.sidebar.title("Sidebar")
st.sidebar.radio("Are you a New User", ["Yes", "No"])
st.sidebar.slider("Select a Number", 0,10)
```

We can use a sidebar to display a navigation link where one can switch from one page to other.

Multipage navigation

We can create multiple-page apps by using navigation. In Streamlit, the navigation sidebar shows the pages that are created. The pages can be navigated quickly and easily as the front end never reloads. The multipage application is divided into two parts: the main page and the pages.

First, we will create a Python file in a directory that acts as the main page. This is similar to the single-page application we created earlier. We have named the file home.py:

```
# home.py
import streamlit as st
st.title("# Main page ")
st.write("This is Main Page")
```

Now we will see how we can add other pages to this home.py page. We will add multiple Python pages in the folder named pages where we have the main_page.py file. We will create two files named page2.py and page3.py in the pages folder.

```
# page2.py in pages folder
import streamlit as st
st.title("# Page 2 ")
st.write("You have navigated to page one")
```

```
# page3.py in pages folder
import streamlit as st
st.title("# Page 3 ")
st.write("You have navigated to page one")
```

Streamlit takes all the file names of the files that are available in the pages folder and places them in the sidebar of the application. Works only in streamlit versions above 1.10.

Control Flow and Advanced Features

In this chapter, we will discuss the different alert boxes available. Then we will learn about control flow aspects in Streamlit. We will also discuss how to change the default flow of a Streamlit application with the methods provided. Finally, we will learn advanced features including changing the default configuration of Streamlit and improving performance using caching techniques.

Alert Box

In this section, we will review some of the alert boxes that are available in Streamlit. These boxes are used to give certain information to the user in order for them to interact with the application.

- st.info(): this function helps to provide additional information to the user so they can interact with the application.
- st.warning( ): this will pop up a warning message to the user.
- st.success( ): the st.success() function displays a success message. For example, we can use it when form data is submitted correctly.
- st.error( ): this is used to display an error message on the application. It can be used when the file upload does not support certain types of files.
- st.exception( ): when we want to handle any exception in our application, we can use the st.exception() function and display a notification once the exception is hit.

```
st.success("Successful")
st.warning("Warning")
st.info("Info")
st.error("Error")
st.exception("It is an exception")
pass
```

Control Flow

We know that Streamlit executes the complete script of an application, but this can be controlled by allowing certain functions to execute, and therefore the default flow of the application can be changed.

Stop Execution

We can stop the execution of the script immediately with st.stop(). In the example below the script will come out of the if condition when any text is entered in the textbox, as we have defined the st.stop()

method, and execute the next condition. This changes the default flow of the application.

```
name = st.text_input('Text')
if not name:
  st.info('Enter any Text.')
  # Stop function
  st.stop()
st.success('Text Entered Successfully.')
```

Rerun the Script

We can rerun the script at any given points of time. To rerun the script, we use the following command: st.experimental_rerun()

```
import time
st.title('Hello World')
st.info('Script Runs Everytime rerun hits.')
time.sleep(2)
# rerun statement
st.experimental_rerun()
st.success('Scipt Never Runs.')
```

When we hit the rereun() statement in the script, the script runs again from the top, and hence the next script will not be executed.

st.form_submit_button

This button is used to submit all the data entered in the form in batches. When the user clicks the submit button, the script reruns. It is a special type of button that should be associated with forms.

```
sub_form = st.form(key='submit_form')
user_name = sub_form.text_input('Enter your username')
# Submit button associated with form
submit_button = sub_form.form_submit_button('Submit')
st.write('Press submit see username displayed below')
if submit_button:
    st.write(f'Hello!!!! {user_name}')
```

The sub_form is associated with the submit button; hence, when the button is clicked, it will send the data in batches and be displayed on the application. This button can be associated to any of the form widgets that we discussed previously.

Advanced features

Configuring the page

We can change the default page title and icon used by Streamlit by configuring the page. This page configuration is done by using the st. config_page() method.

```
st.set_page_config(page_title='ML App', page_icon=':robot_face:')
st.title("Page Configured")
```

st.echo

When we want the code that creates the graphics at the front end, we can use echo. We can also adjust when the code is displayed, i.e., before or after execution of the code. In this example, we have displayed the code after execution by getting the text input from the user.

```
with st.echo():
    txt = st.text_input('Text')
    if not txt:
        st.warning('Input a text to see sample code.')
        st.stop()
    st.success('Thank you for text input.')
```

st.experimental_show

To debug our application, we can use the experimental_show() method. We can pass multiple arguments to debug the application. When this method is called, it returns a None value so that the method can be used only for debugging.

```
import pandas as pd
import numpy as np
st.title("Experimental Show for Debugging")
# Dataframe
df = pd.DataFrame(np.random.randint(0,100,size=(5, 4)),
columns=list('WXYZ'))
# Defining Experimental show
st.experimental_show(df)
```

Session State

Session state is a unique feature of the Streamlit library where we can store values and share the same values between the application reruns. It holds a key-value pair similar to a Python dictionary. We can use the following command to access session state: st.session_state. The session state holds the value of only the current session, and after refreshing, a new session state is created. Once we refresh the page, a new session is created, and the value stored in the session state is erased. The application then stores the value of the new session.

```
# Session state initialization
if 'sum' not in st.session_state:
    st.session_state.sum = 0
# Button to add value
add = st.button('Add One')
if add:
    st.session_state.sum += 1
st.write('Total Sum = ', st.session_state.sum)
```

We have a button that adds one value to the total sum when we click it. Every time we click the button, the application reruns and stores the incremented value in the session state. If the session state is not initialized, then the session state value is set to zero.

Performance

In this section, we will look into some of the techniques that help to increase performance by increasing the robustness of computations in an application. We will discuss how to increase the performance of the application by using the default settings as well as advanced options.

Caching

When we want to load a large dataset, manipulate data, or perform heavy computational work, we need our application to remain stable and perform as expected. The caching mechanism is provided by Streamlit to handle such things.

When we use a cache in our application as a decorator, Streamlit will look at the following things: • The input parameters of the called function • The external variable value used in the function • The body of the function • Any function body used inside the cached function

When Streamlit sees these components for the first time in the same order, it will store the values when the script is executed. Streamlit monitors if there are any changes in the components by using a hashing technique. This is similar to a key-value pair in the memory store wherein the key is a hash and the value is the object passed by reference.

```
import time
# Defining cache
@st.cache
def add(x, y):
    # Function takes 5 secs to run
    time.sleep(5)
    return x + y
x = 10
y = 60
res = add(x, y)
st.write("Result:", res)
```

st.experimental_memo

In the case of high computations, we use experimental_memo to store data. It holds cache data in key-value format. The command to initialize experimental_memo is as follows: @st.experimental_memo We can use @st.experimental_memo in place of @cache to store the results from heavy computations. It is derived from the same API as the cache.

st.experimental_memo.clear( )

To clear the cache store in st.experimental_memo, we will use the following command: st.experimental_memo.clear(). The memo can be used to store downloaded data or for dataframe computations or calculations between n digits, respectively.

st.experimental_singleton

All users connected to the app share each singleton object. Since they can be accessed by multiple threads at once, singleton objects must be thread- safe. To define a singleton, we will use the following statement: @st.experimental_memo. A Streamlit application shares this key-value store with all of its sessions. It works well for storing large singleton objects over sessions such as database connections or sessions for TensorFlow, Torch, or Keras.

st.experimental_singleton.clear

When the singleton is holding the data and needs to be cleared, we use the following function: st.experimental_singleton.clear()

Natural Language Processing

189-201

Computer Vision in Streamlit

202-