Building dashboards with Plotly and Dash

Outline

- Introduction to Plotly
- Charts in Plotly
- Introduction to Dash
- App structure in Dash
- Layout and Interactivity and Callbacks
- Practical Examples

What is plotly ?

- Plotly is a Python library used for creating interactive, web-based visualizations.

- It supports a wide range of charts and plots, including line graphs, scatter plots, and 3D graphs.

- Its ultimate strength lies in the ability to build visually appealing, dynamic graphics that can be easily integrated into web applications and dashboards.

plotly.express module

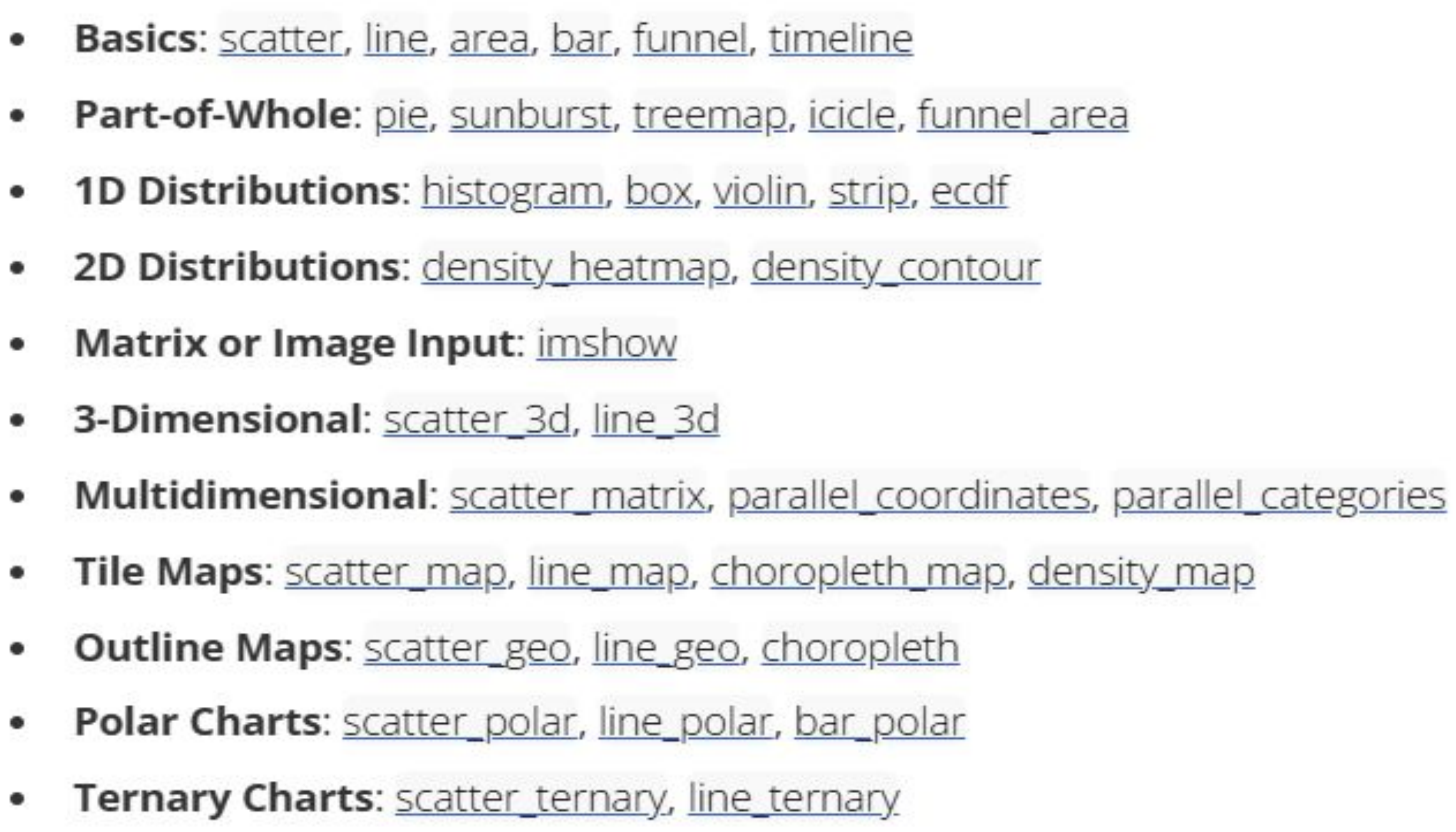
- The plotly.express module (usually imported as px) contains functions that can create entire figures at once, and is referred to as Plotly Express or PX.

- Every Plotly Express function uses graph objects internally and returns a plotly.graph_objects.Figure instance.

- Any figure created in a single function call with Plotly Express could be created using graph objects alone, but with between 5 and 100 times more code.

Plotly Express currently includes the following functions:

https://plotly.com/python/plotly-express/

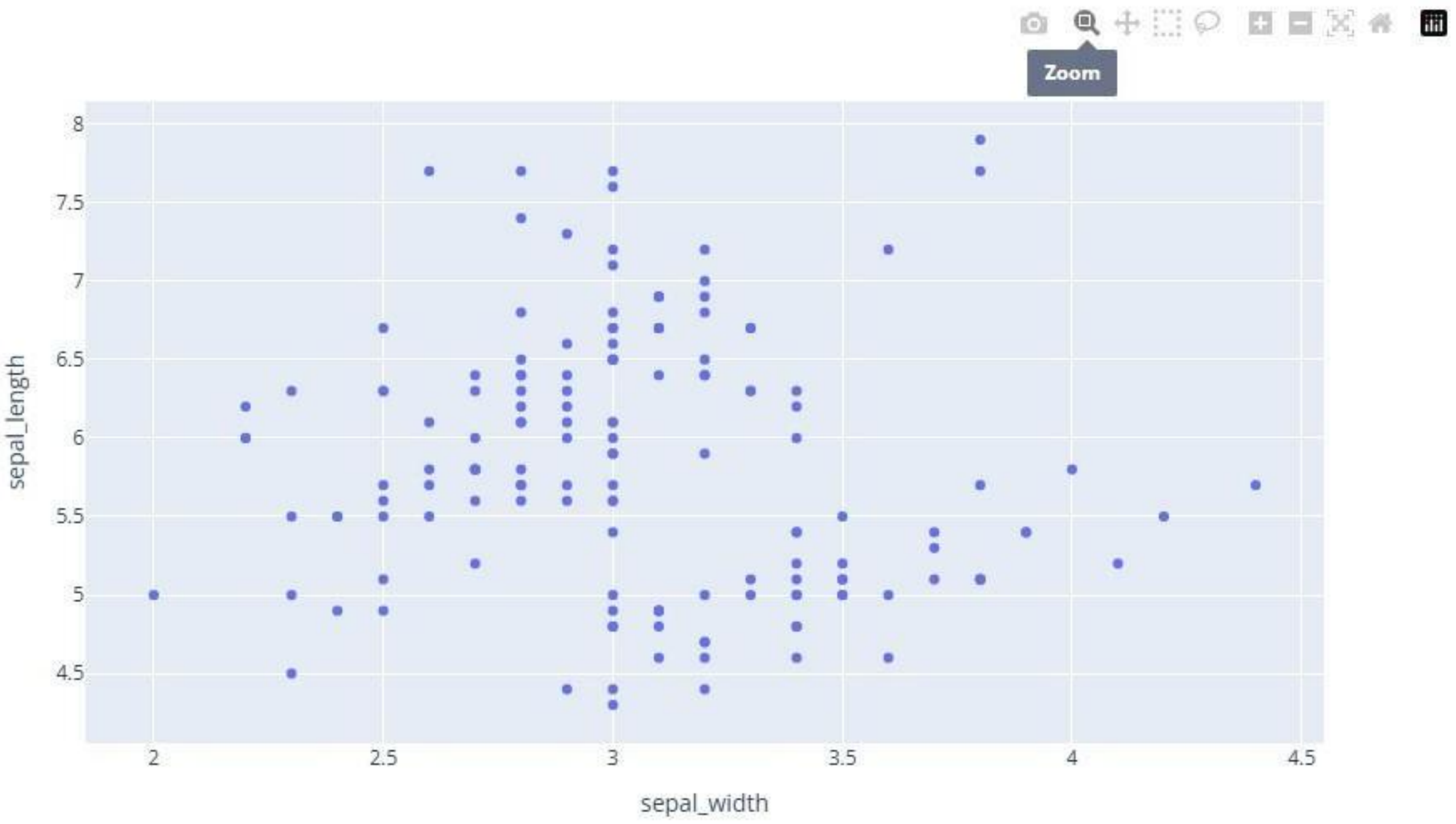


- **Basics**: scatter, line, area, bar, funnel, timeline
- **Part-of-Whole**: pie, sunburst, treemap, icicle, funnel_area
- **1D Distributions**: histogram, box, violin, strip, ecdf
- **2D Distributions**: density_heatmap, density_contour
- **Matrix or Image Input**: imshow
- **3-Dimensional**: scatter_3d, line_3d
- **Multidimensional**: scatter_matrix, parallel_coordinates, parallel_categories
- **Tile Maps**: scatter_map, line_map, choropleth_map, density_map
- **Outline Maps**: scatter_geo, line_geo, choropleth
- **Polar Charts**: scatter_polar, line_polar, bar_polar
- **Ternary Charts**: scatter_ternary, line_ternary

Scatter, Line, and Bar Charts

1. px.scatter()

- data_frame
- x
- y
- color
- size
- facet_row
- facet_col
- title
- labels
- template
- height
- width
- animation_frame

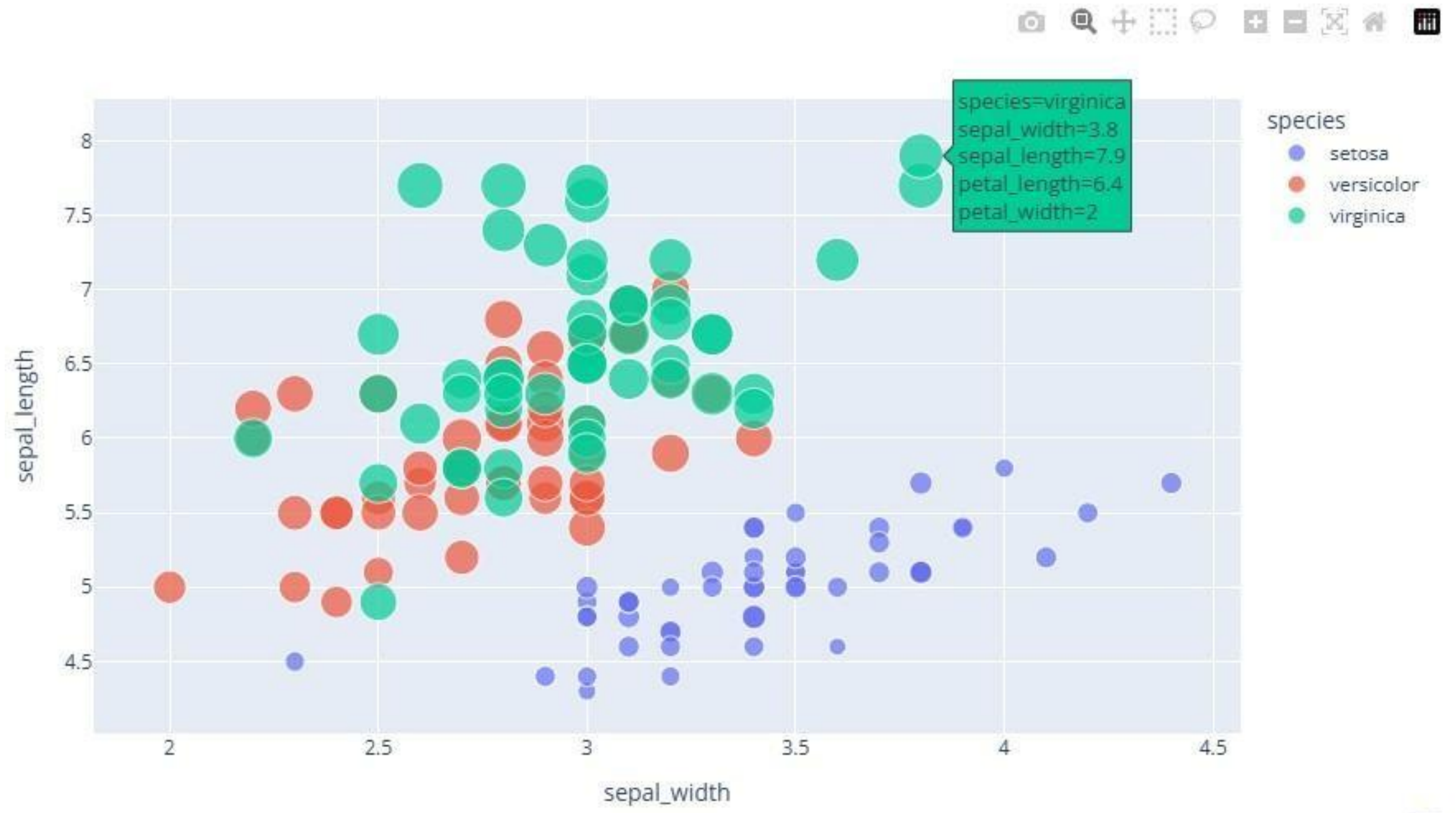


```
df = px.data.iris()
fig = px.scatter(df, x="sepal_width", y="sepal_length")
fig.show()
```

Setting size and color with column names :

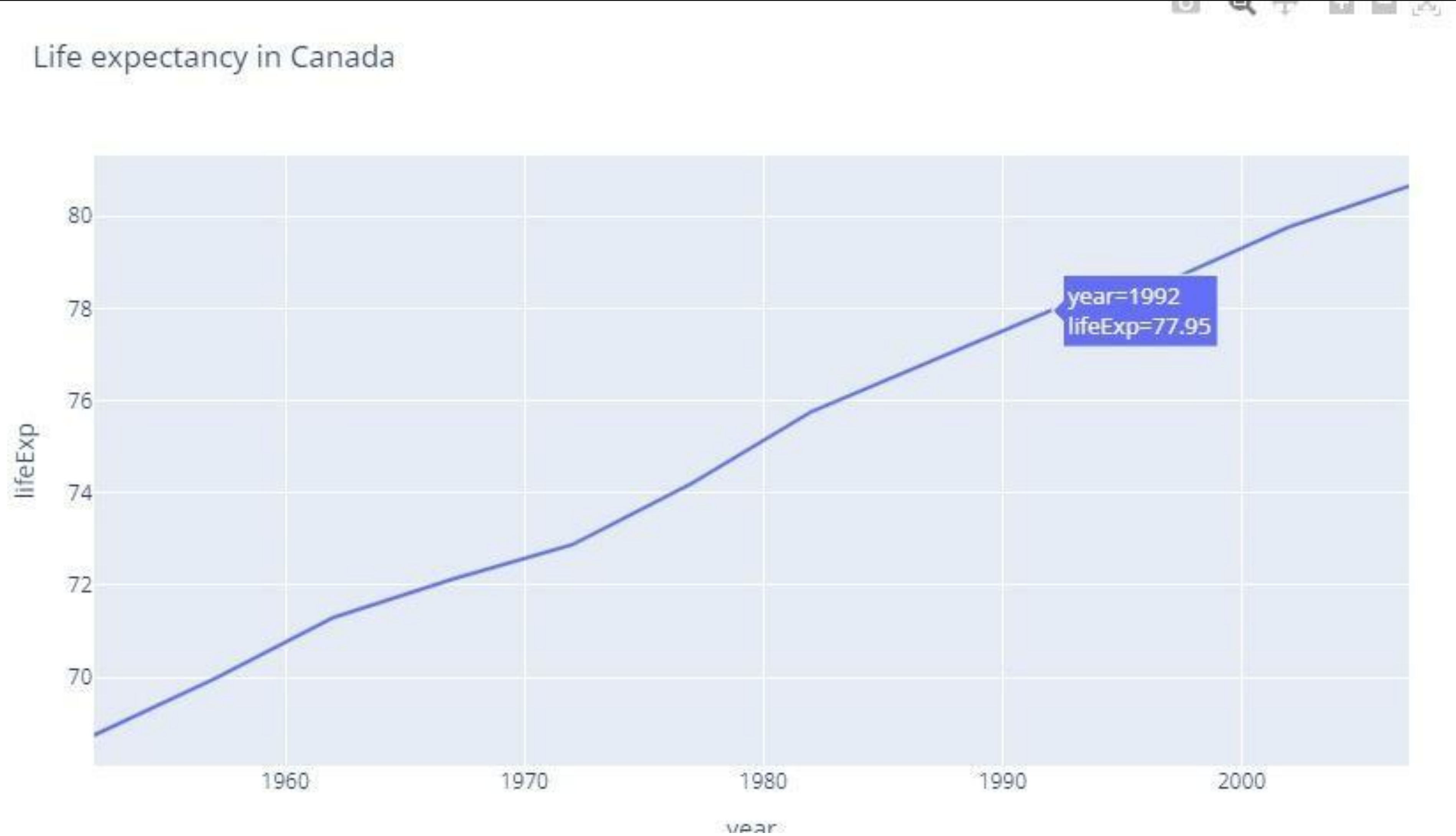
- Scatter plots with variable-sized circular markers are often known as **bubble charts**
- Note that color and size data are added to hover information. You can add other columns to hover data with the hover_data argument of px.scatter

```
import plotly.express as px
df = px.data.iris()
fig = px.scatter(df, x="sepal_width", y="sepal_length", color="species",
                 size='petal_length', hover_data=['petal_width'])
fig.show()
```

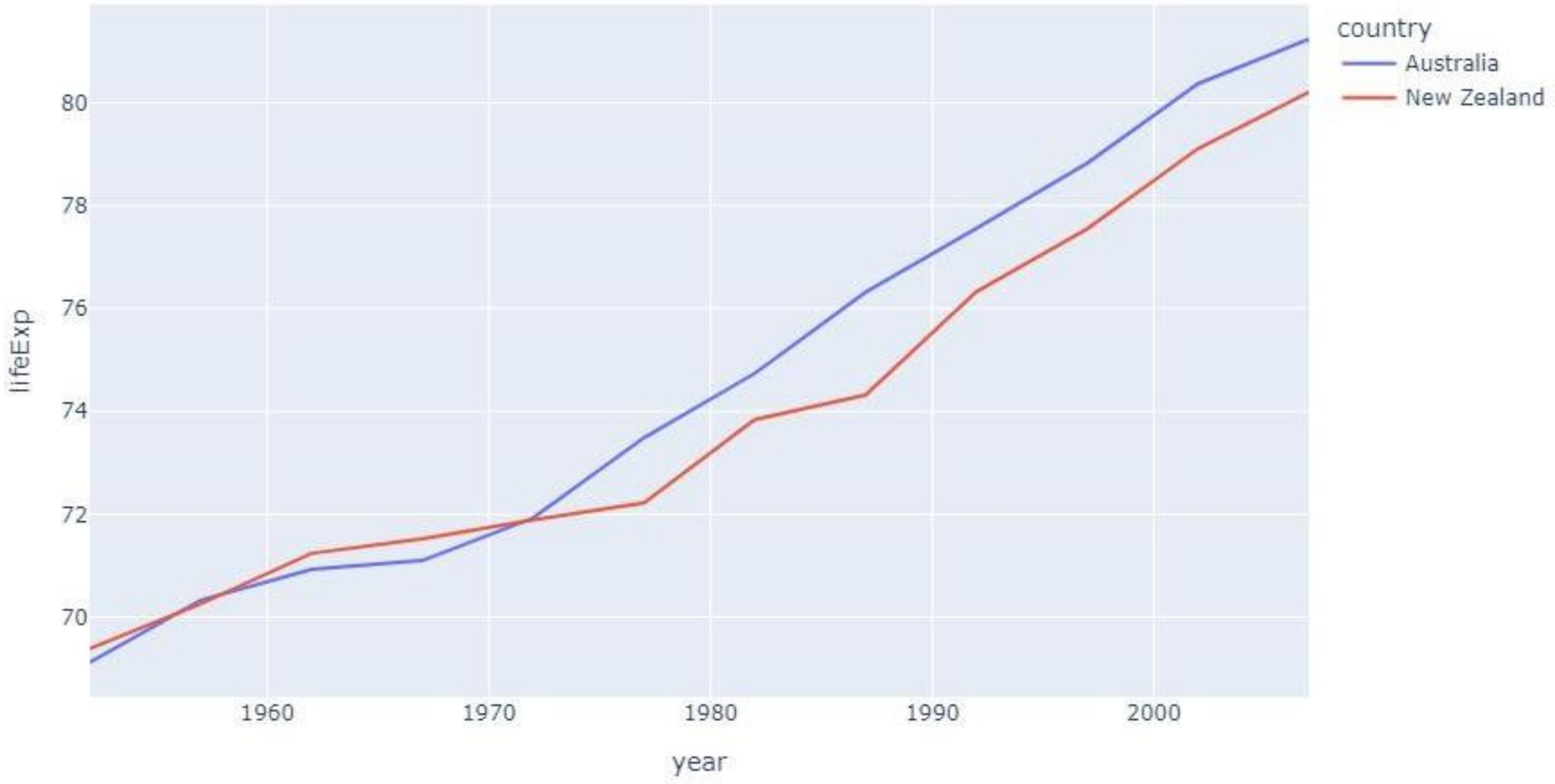

species=virginica
sepal_width=3.8
sepal_length=7.9
petal_length=6.4
petal_width=2
species
setosa
versicolor
virginica
sepal_length
sepal_width
8
7.5
7
6.5
6
5.5
5
4.5
2
2.5
3
3.5
4
4.5

2. px.line()

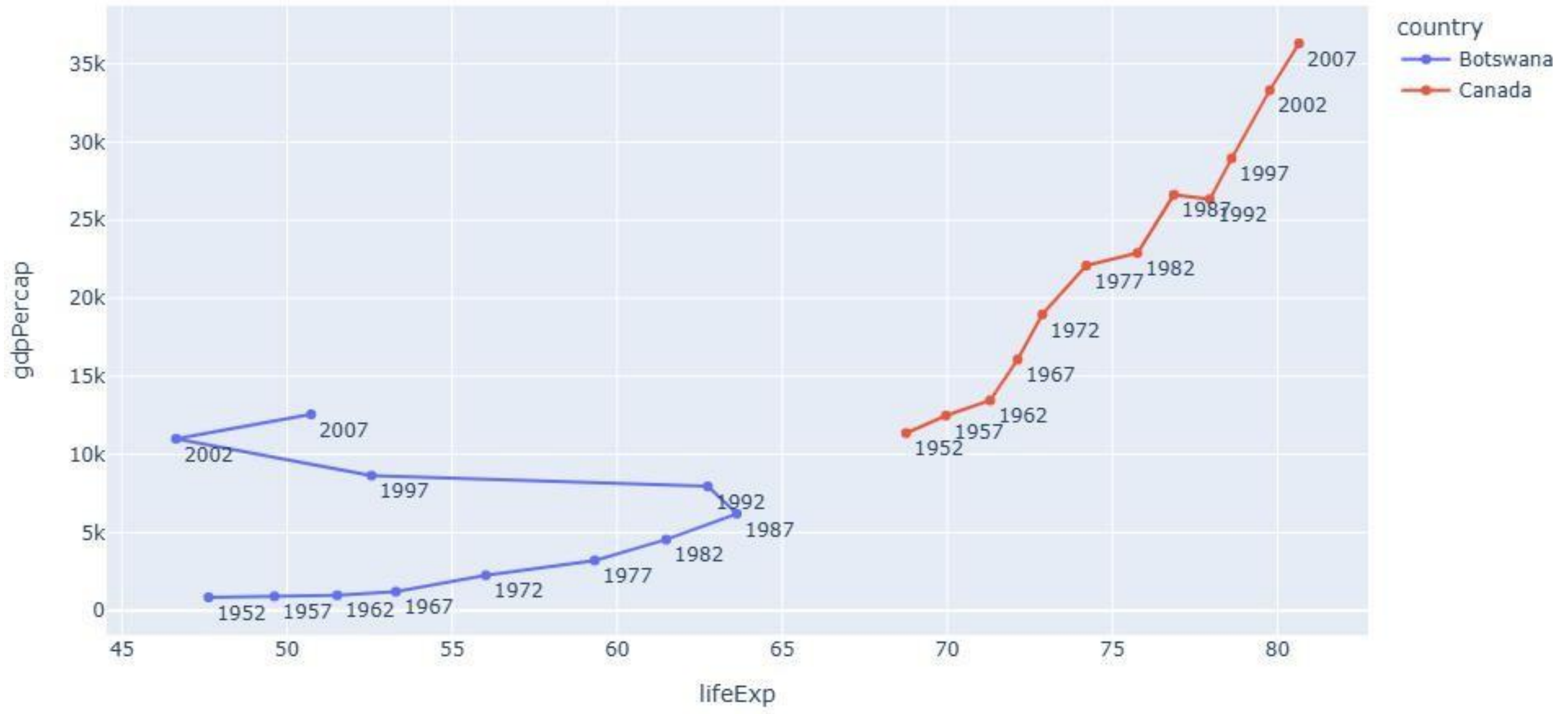
- data_frame
- x
- y
- color
- size
- facet_row
- facet_col
- title
- labels
- template
- height
- width
- animation_frame
- line_shape
- range_x
- range_y
- line_color



```
df = px.data.gapminder().query("country=='Canada'")
fig = px.line(df, x="year", y="lifeExp", title='Life expectancy in Canada')
fig.show()
```



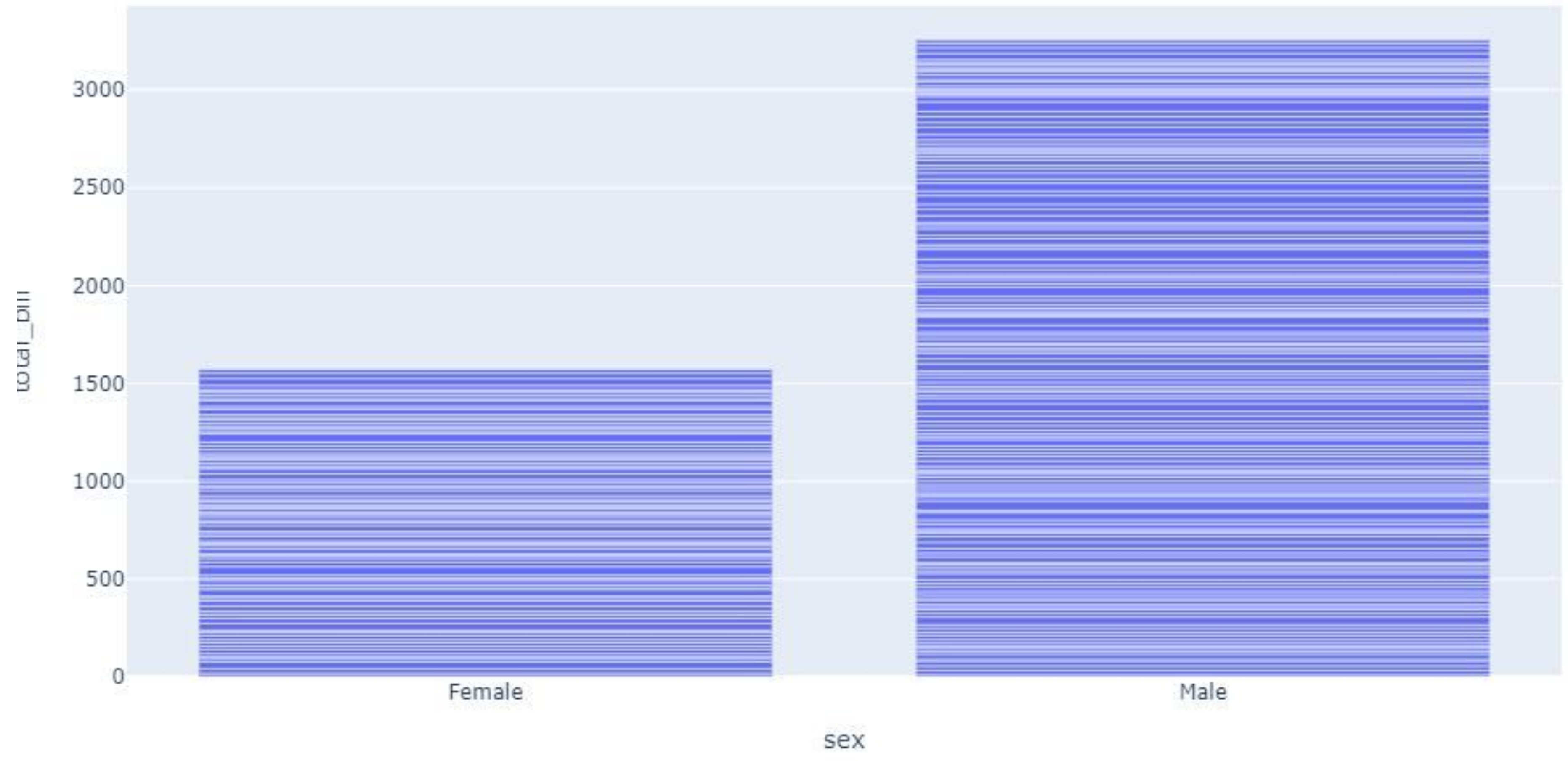
```
df =
px.data.gapminder().query("continent=='Oceania'") fig
= px.line(df, x="year", y="lifeExp", color='country')
```



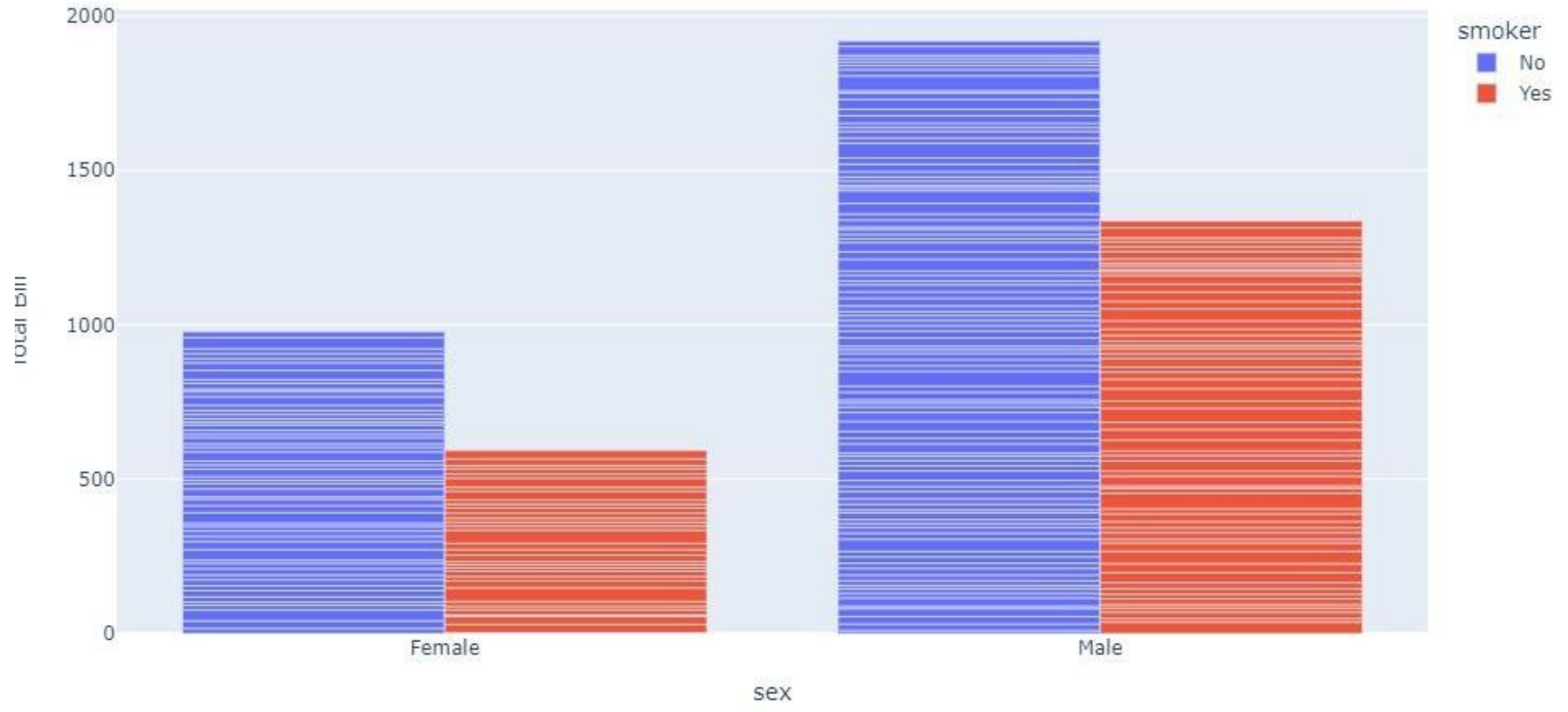
```
fig =
px.line(df,x="lifeExp",y="gdpPercap",color="country",text="year")
fig.update_traces(textposition="bottom right")
```

3. px.bar()

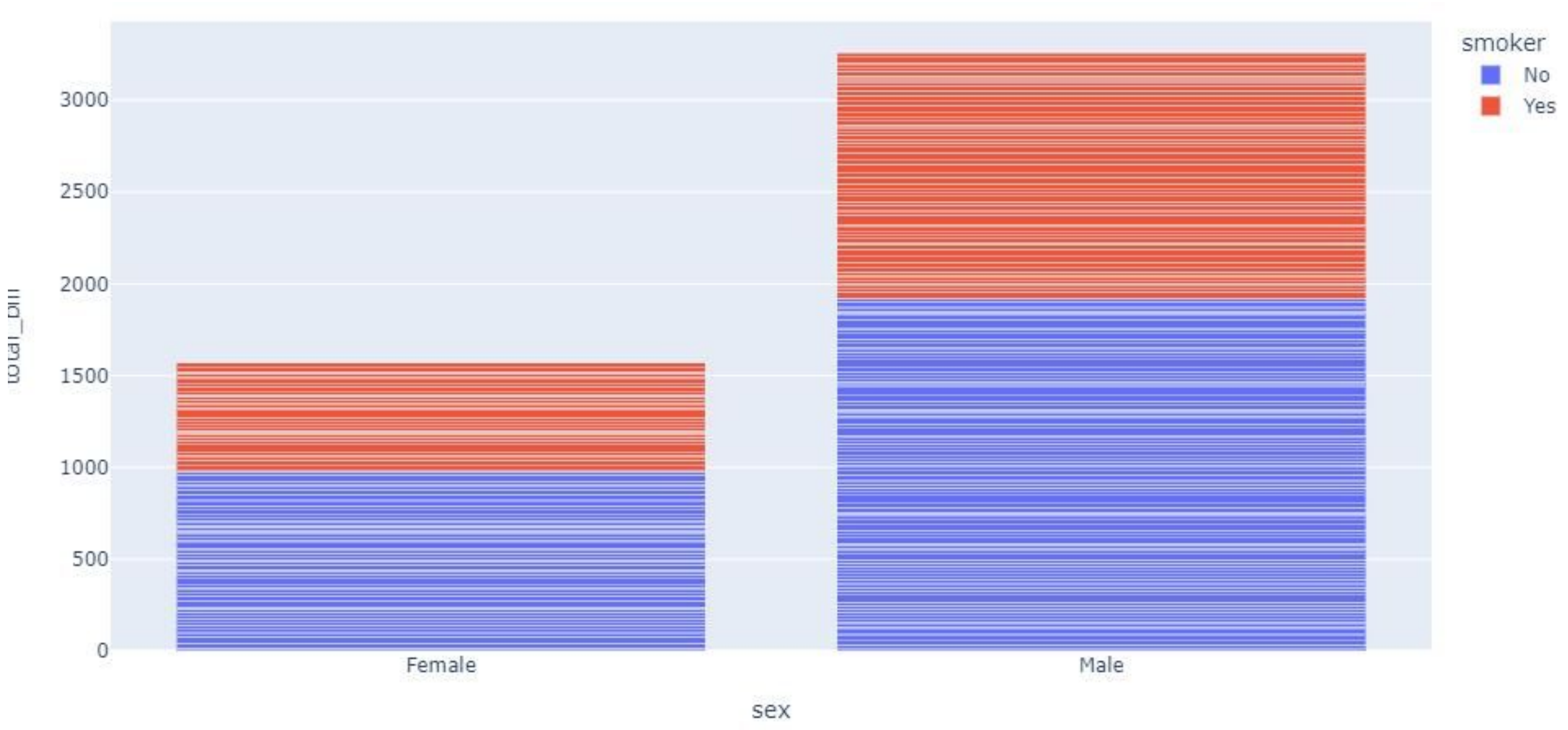
- data_frame
- x
- y
- color
- size
- facet_row
- facet_col
- title
- labels
- template
- height
- width
- animation_frame
- barmode



```
df = px.data.tips()
fig = px.bar(df, x="sex", y="total_bill")
fig.show()
```



```
df = px.data.tips()
fig =
px.bar(df,x="sex",y="total_bill",color="smoker",barmode="group")
```

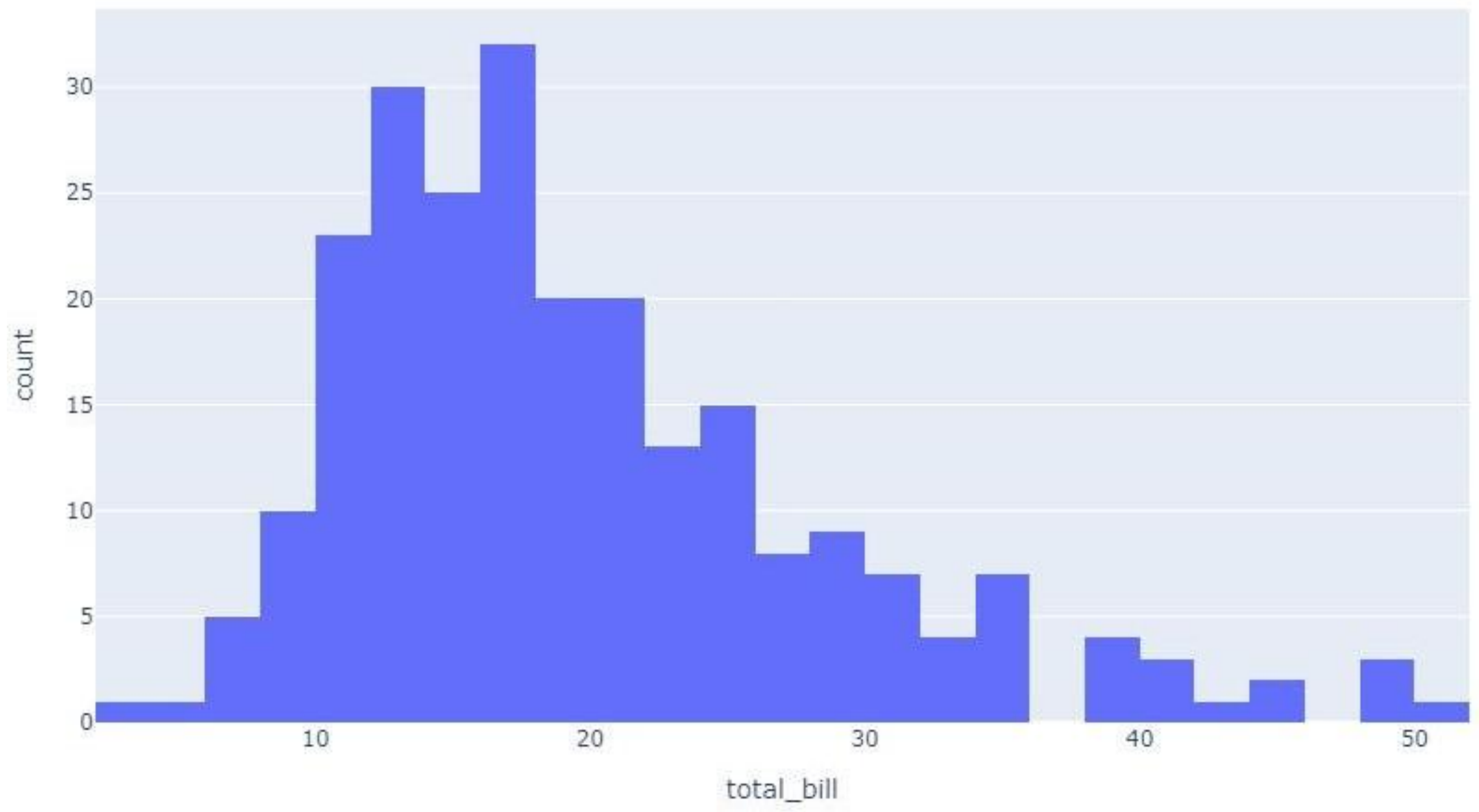


```
df = px.data.tips()
fig = px.bar(df, x="sex", y="total_bill", color="smoker", barmode="stack")
fig.show()
```

1-D destribution

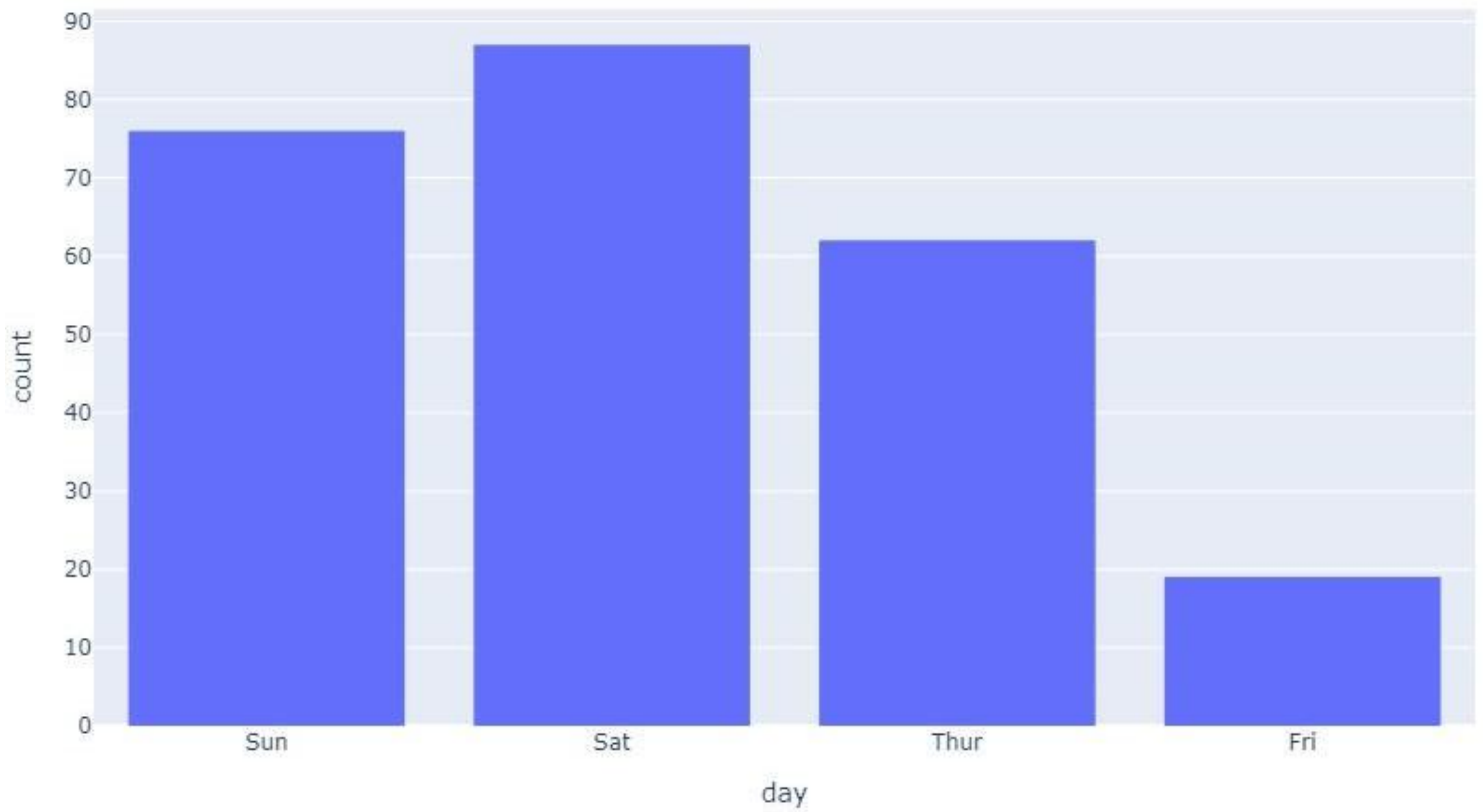
1. px.histogram()

- data_frame
- x
- y
- color
- barmode
- nbins
- title
- labels
- template
- height
- width
- range_x
- range_y



```
df = px.data.tips()
fig = px.histogram(df,
x="total_bill") fig.show()
```

Numerical data

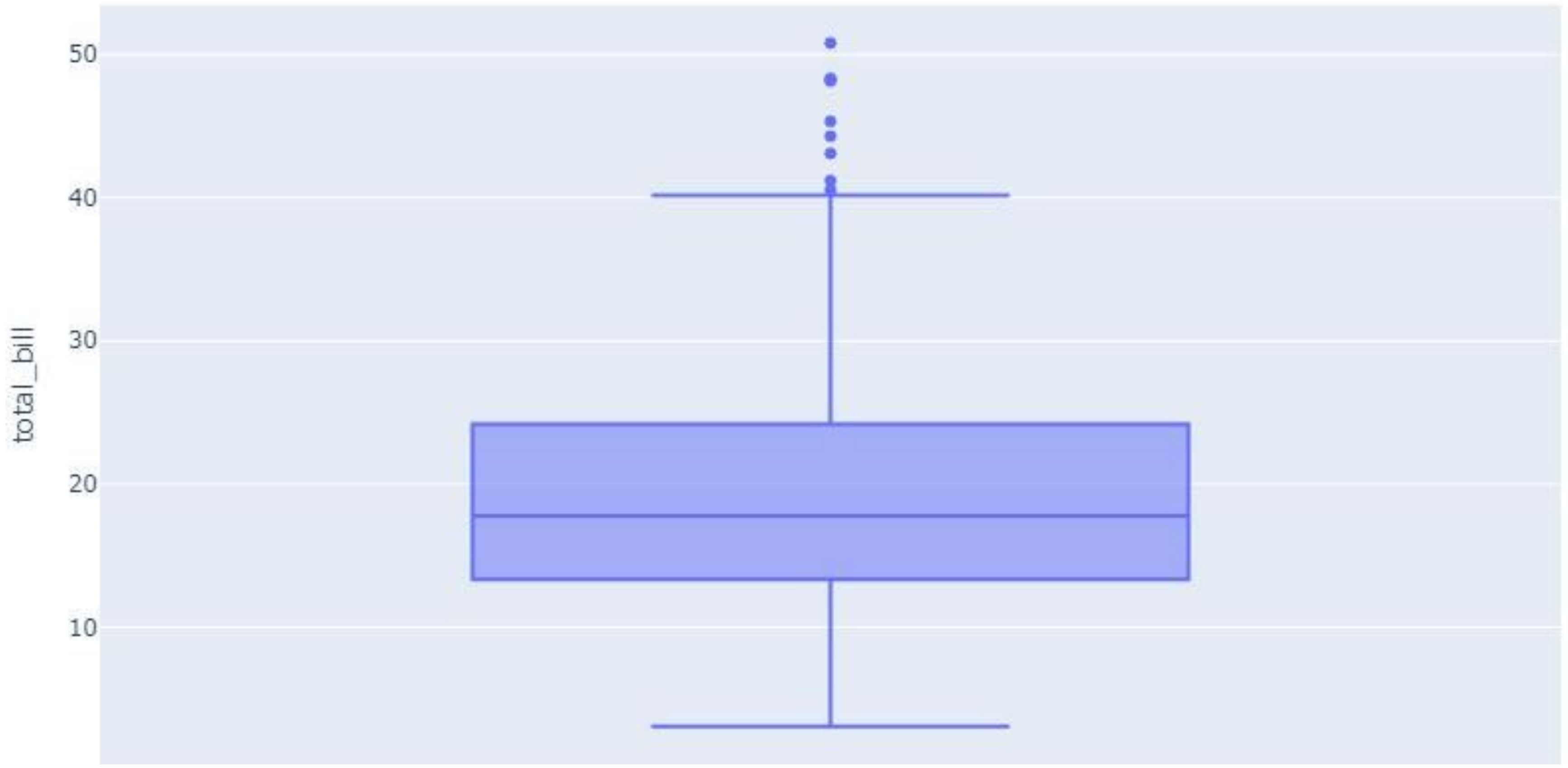


Categorial data

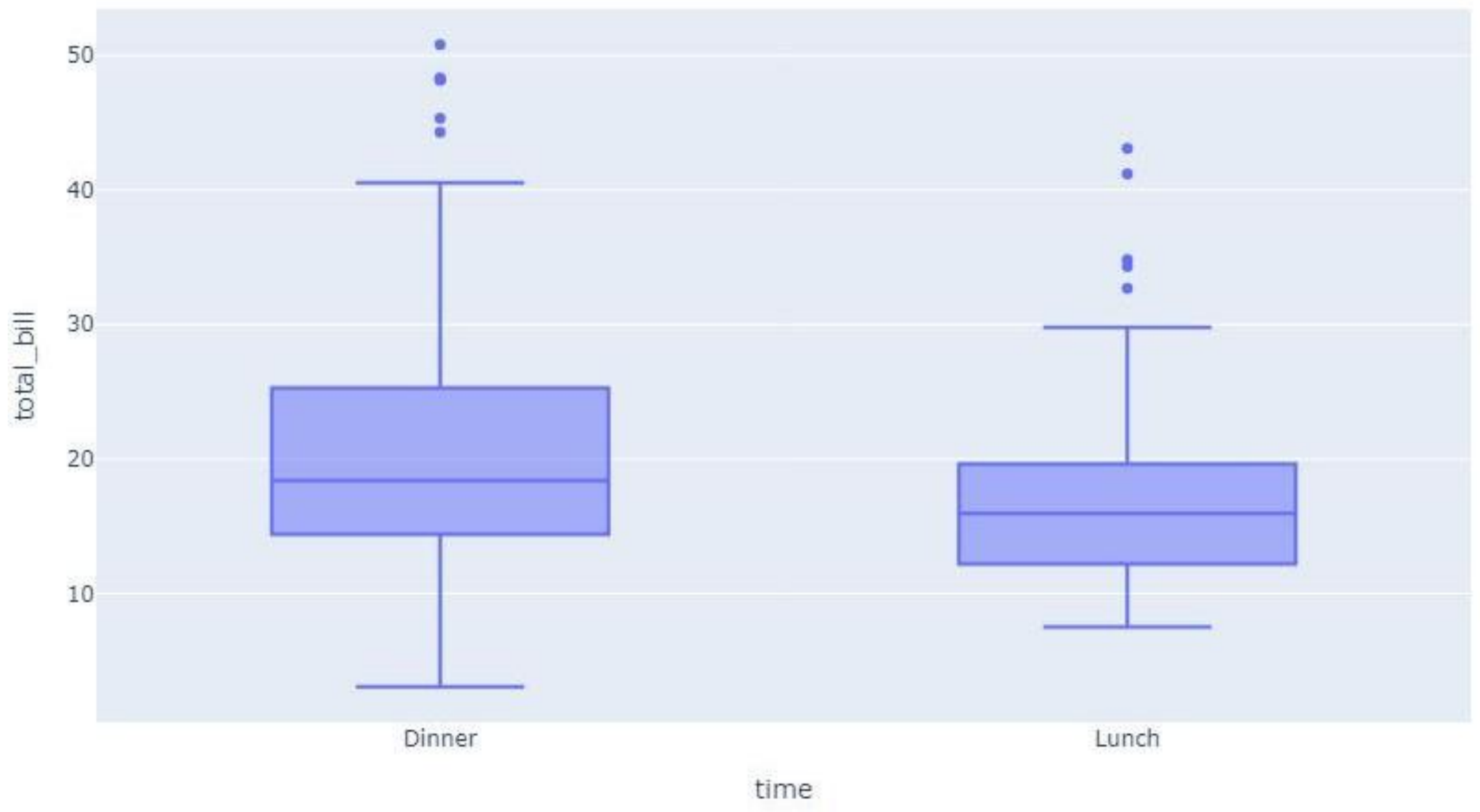
```
df = px.data.tips()
fig = px.histogram(df,
x="day") fig.show()
```

2. px.box()

- data_frame
- x
- y
- color
- size
- facet_row
- facet_col
- title
- labels
- template
- height
- width
- animation_frame



```
df = px.data.tips()
fig = px.box(df,
y="total_bill") fig.show()
```



```
df = px.data.tips()
fig = px.box(df, x="time", y="total_bill")
fig.show()
```

Practical Work in:

- 1_Plotly_Basics.ipynb
- 2_Plotly_Basics.ipynb

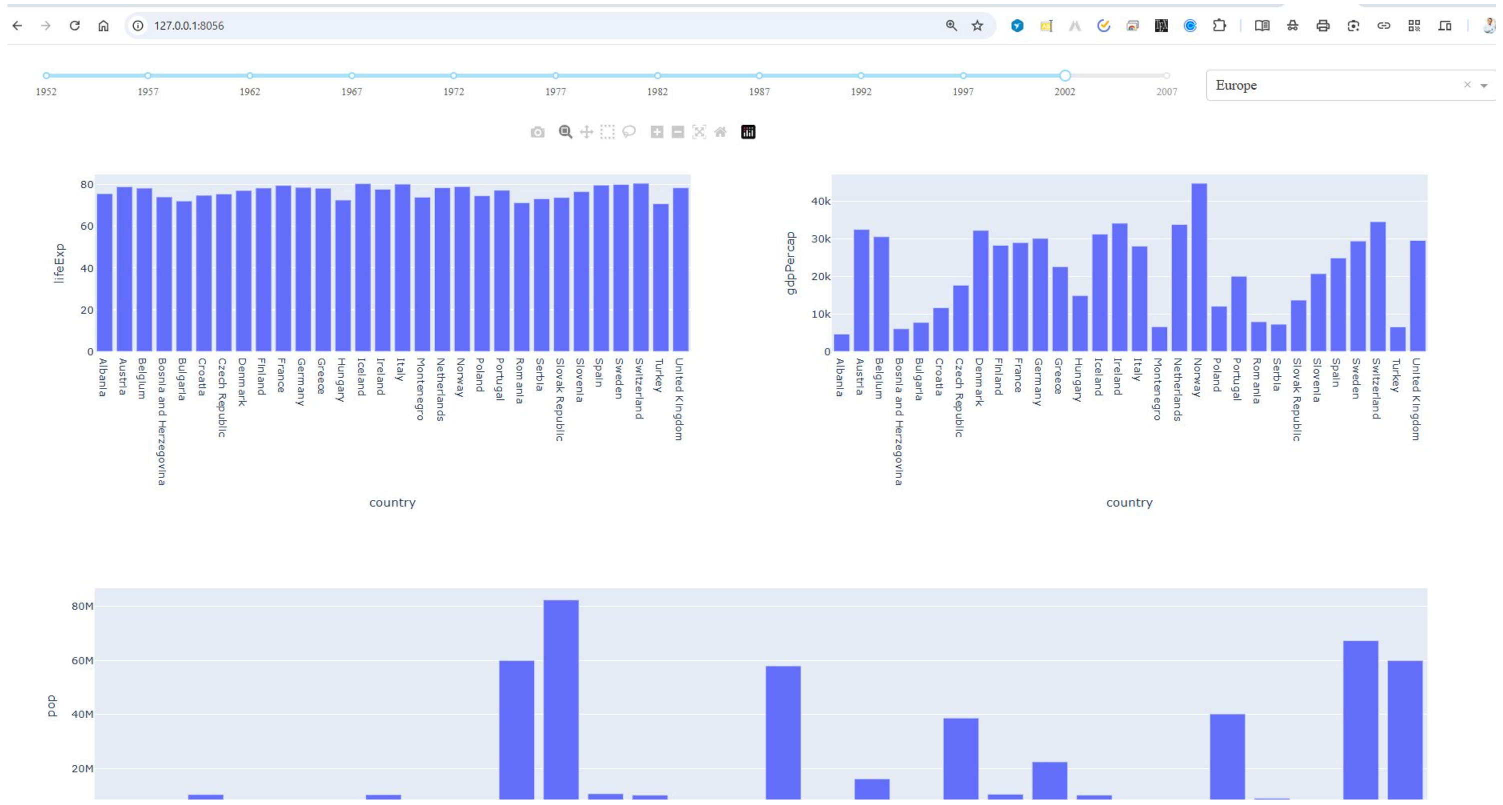
Dash Framework

What is Dash?

- Dash is a Python framework for building interactive web applications, particularly for data visualization and analytics
- No need for deep knowledge of HTML, CSS, or JavaScript.
- Ideal for data visualization and dashboards.

Key Technologies Behind Dash

- **Flask**: Handles the backend server.
- **React.js**: Manages the interactive frontend.
- **Plotly.js**: Enables advanced data visualization.



Why Use Dash?

- Simplifies web development for Python users.
- Supports dynamic and interactive UI components.
- Easily integrates with data science workflows.

https://plotly.com/python/plotly-express/#plotly-express-in-dash

App structure in Dash

Every Dash application has four prime components:

1. App instance
2. Layout
3. Callback function (Optional)
4. app.runserver()

Layout

- Core components

    `from dash import dcc`

- HTML components

    `from dash import html`

Core components :

- Higher-level interactive components that are generated JavaScript , CSS and HTML through react.js
- Slider , input area , check items , datepicker and more

HTML components :

- Component for each html tag
- Key-word arguments describe the HTML attributes like Style , ClassName and id

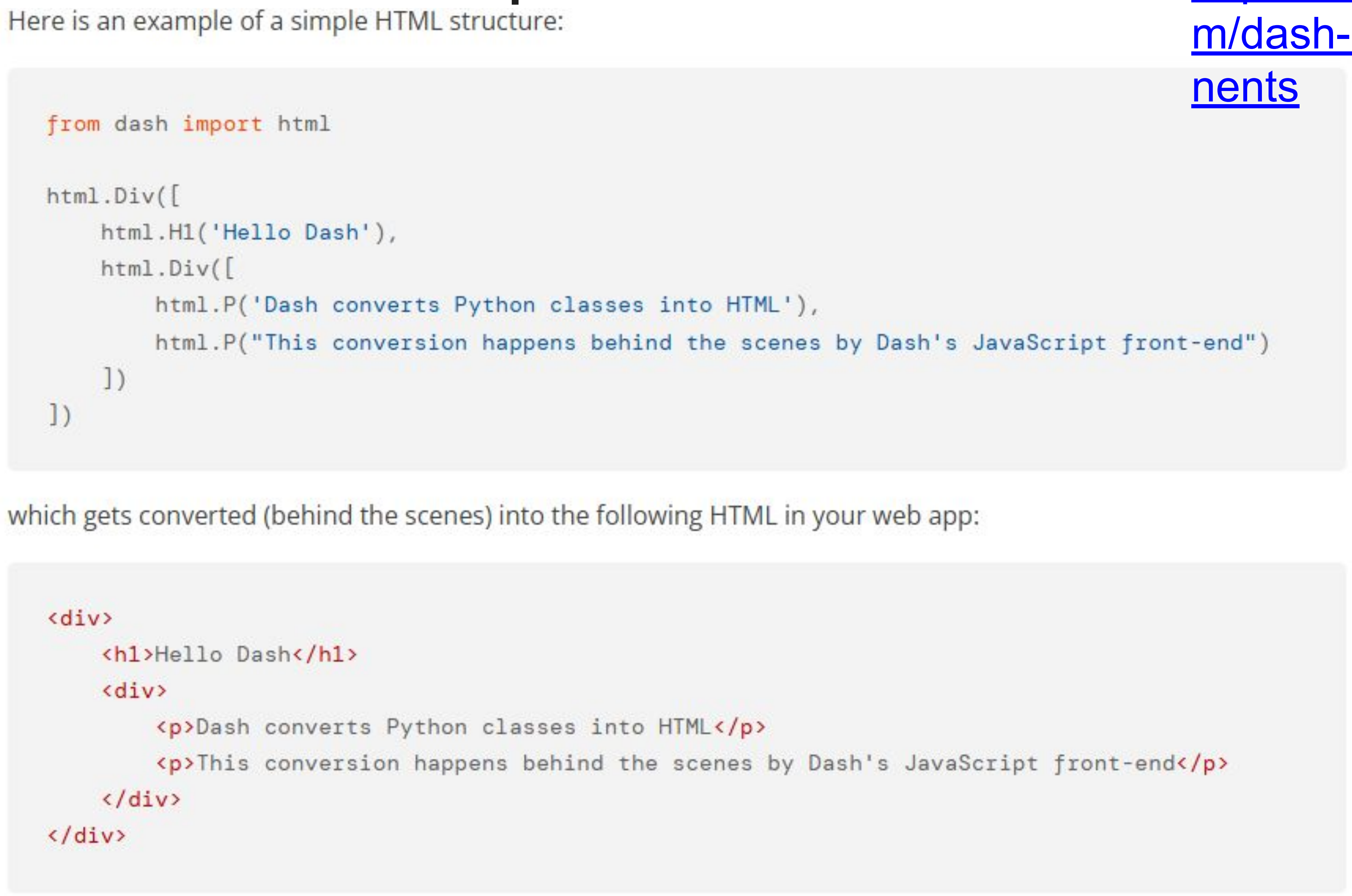
HTML components :

- Dash is a web app framework that provides pure Python abstraction around HTML, CSS, and JavaScript.

- Instead of writing HTML or using an HTML templating engine, you compose your layout using Python with the Dash HTML Components module (dash.html).

Dash HTML Components

https://dash.plotly.com/dash-html-components



Here is an example of a simple HTML structure:

```python
from dash import html

html.Div([
    html.H1('Hello Dash'),
    html.Div([
        html.P('Dash converts Python classes into HTML'),
        html.P("This conversion happens behind the scenes by Dash's JavaScript front-end")
    ])
])
```

which gets converted (behind the scenes) into the following HTML in your web app:

```html
<div>
    <h1>Hello Dash</h1>
    <div>
        <p>Dash converts Python classes into HTML</p>
        <p>This conversion happens behind the scenes by Dash's JavaScript front-end</p>
    </div>
</div>
```

HTML Component Properties

If you're using HTML components, then you also have access to properties like `style`, `class`, and `id`. All of these attributes are available in the Python classes.

The HTML elements and Dash classes are mostly the same but there are a few key differences:

- The `style` property is a dictionary
- Properties in the `style` dictionary are camelCased
- The `class` key is renamed as `className`
- Style properties in pixel units can be supplied as just numbers without the `px` unit

Let's take a look at an example.

```
from dash import html

html.Div([
    html.Div('Example Div', style={'color': 'blue', 'fontSize': 14}),
    html.P('Example P', className='my-class', id='my-p-element')
], style={'marginBottom': 50, 'marginTop': 25})
```

That Dash code will render this HTML markup:

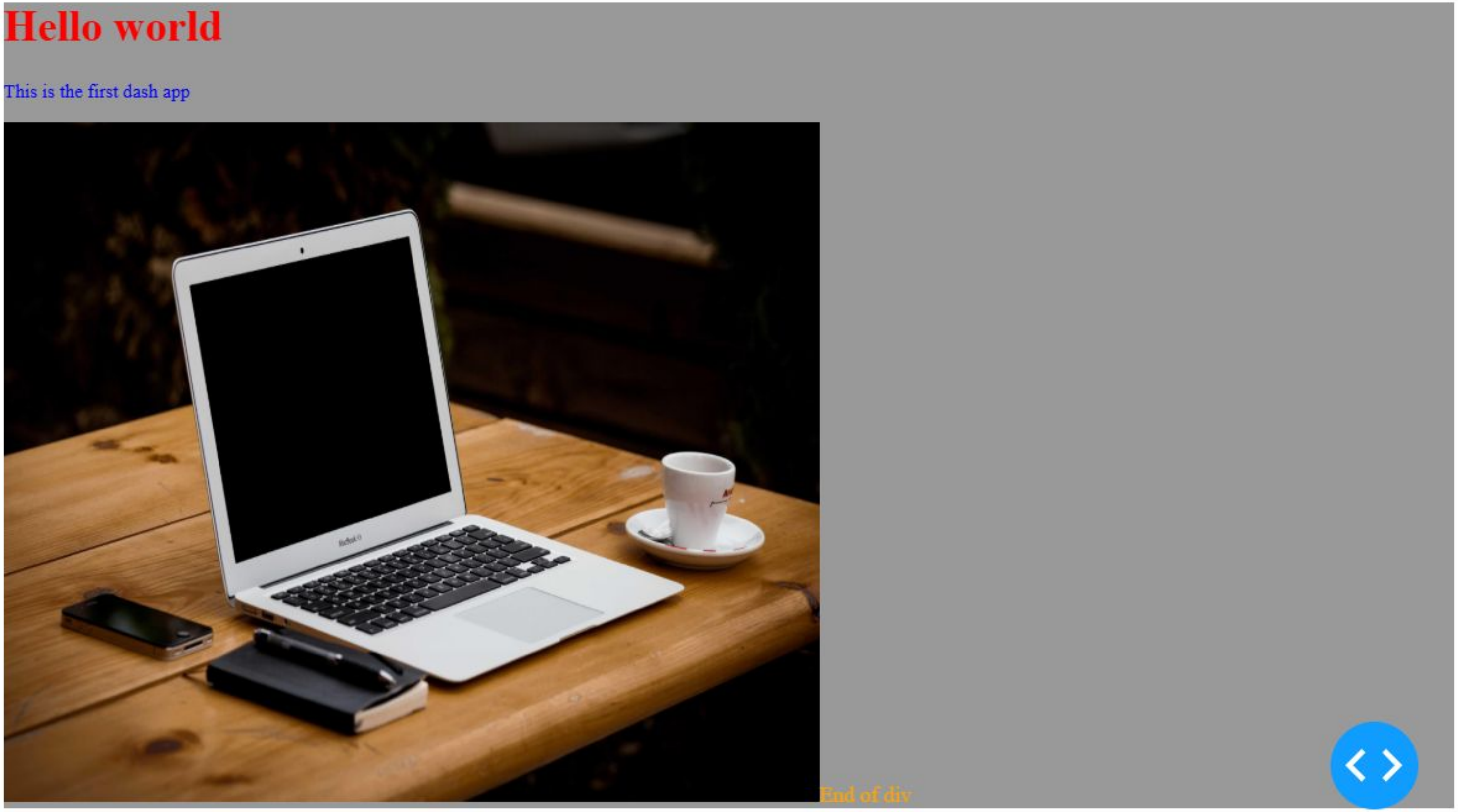
```
<div style="margin-bottom: 50px; margin-top: 25px;">

    <div style="color: blue; font-size: 14px">
        Example Div
    </div>

    <p class="my-class", id="my-p-element">
        Example P
    </p>

</div>
```

1 - HTML components

•[3]:

```python
# 1 - App instance

app1 = Dash(__name__)

# 2 - App layout

app1.layout = html.Div(children = [
    html.H1('Hello world', style={'color': 'red'}),
    html.P('This is the first dash app', style={'color': 'blue', 'fontSize': 14}),
    html.Img(src='https://fastly.picsum.photos/id/0/5000/3333.jpg?hmac=_j6ghY5fCfSD6tvtcV74zXivkJSPIfR9B8w34XeQmvU' ,
            style={'height':500 , 'width':600}),
    'End of div'

], style = {'background-color' :'#999' , 'color':'orange'})

# 3 - Run application on http://127.0.0.1:8050/

if __name__ == '__main__':
    app1.run_server(debug=True , port = 8050)
```

```python
if __name__ == '__main__':
    app1.run_server(debug=True , port = 8050)
```



Hello world

This is the first dash app

End of div

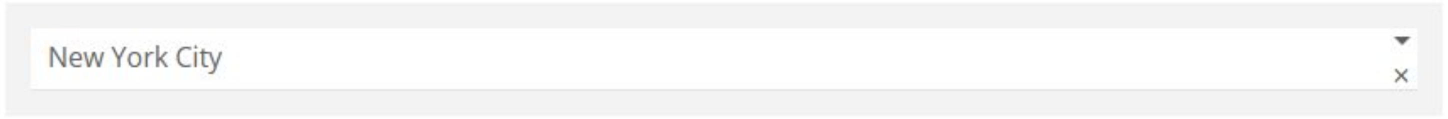
Core components :

- Dash ships with supercharged components for interactive user interfaces.

- The Dash Core Components module (dash.dcc) gives you access to many interactive components, including dropdowns, checklists, and sliders.

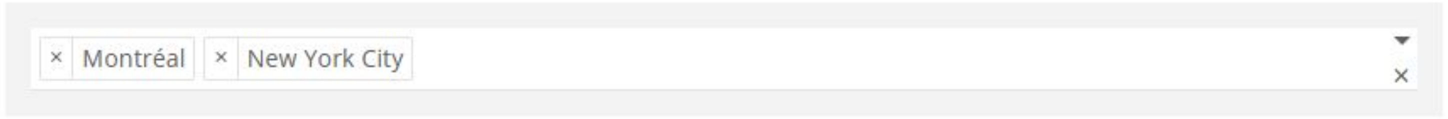
Dropdown

```python
from dash import Dash, html, dcc

app = Dash()

app.layout = html.Div([
    dcc.Dropdown(['New York City', 'Montréal', 'San Francisco'], 'Montréal')
])

if __name__ == '__main__':
    app.run(debug=True)
```



New York City

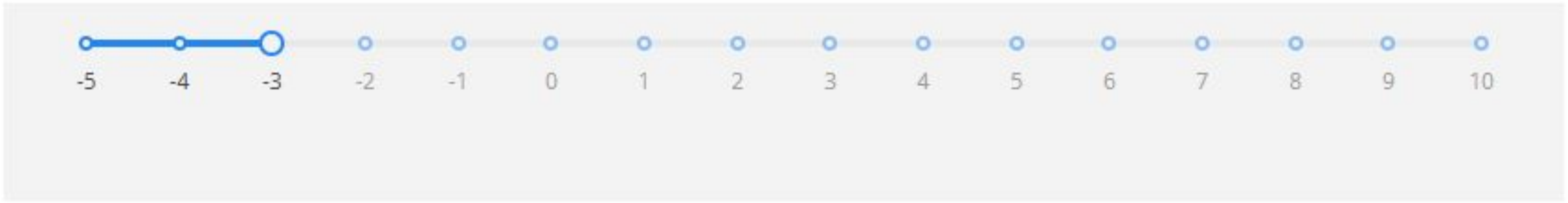
```python
from dash import Dash, dcc, html

app = Dash()

app.layout = html.Div([
    dcc.Dropdown(['New York City', 'Montréal', 'San Francisco'], 'Montréal', multi=True)
])

if __name__ == '__main__':
    app.run(debug=True)
```



× Montréal × New York City

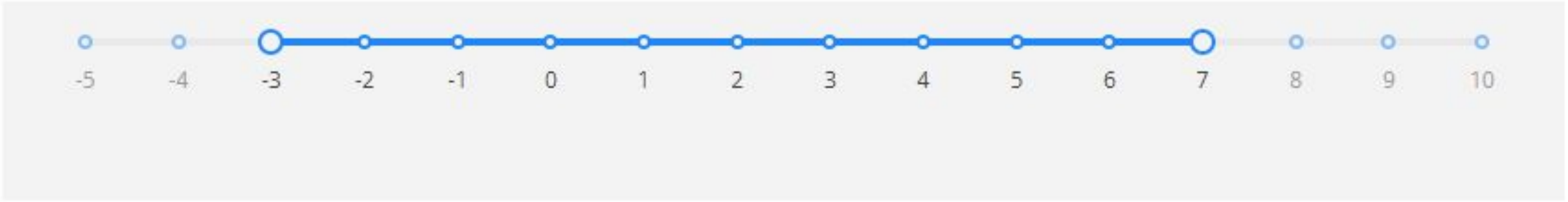
Slider

```
from dash import Dash, dcc, html

app = Dash()

app.layout = html.Div([
    dcc.Slider(-5, 10, 1, value=-3)
])

if __name__ == '__main__':
    app.run(debug=True)
```



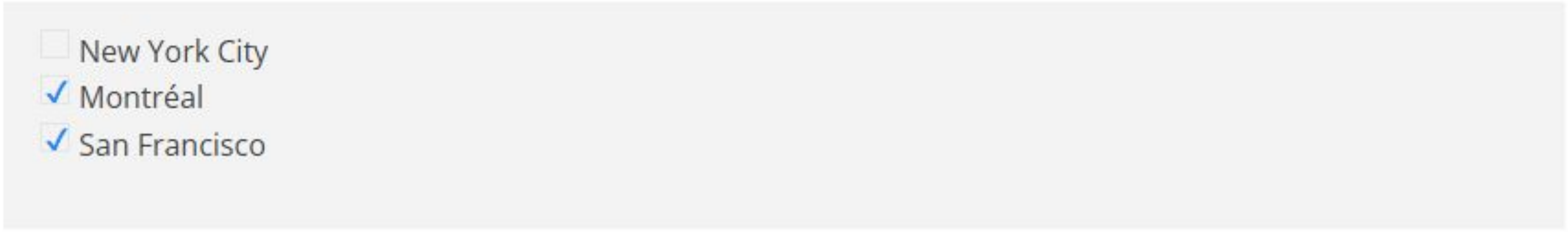
RangeSlider

```python
from dash import Dash, dcc, html

app = Dash()

app.layout = html.Div([
    dcc.RangeSlider(-5, 10, 1, count=1, value=[-3, 7])
])

if __name__ == '__main__':
    app.run(debug=True)
```



Checkboxes

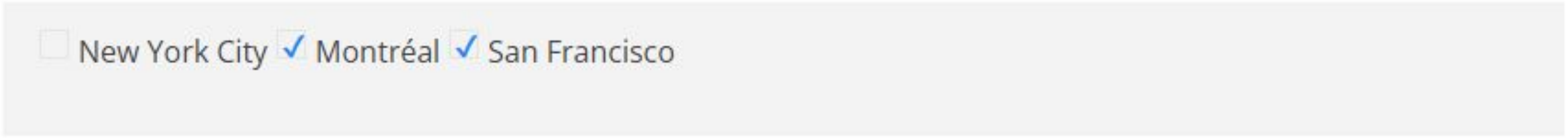
```python
from dash import Dash, dcc, html

app = Dash()

app.layout = html.Div([
    dcc.Checklist(['New York City', 'Montréal', 'San Francisco'],
                  ['Montréal', 'San Francisco'])
])

if __name__ == '__main__':
    app.run(debug=True)
```



☐ New York City
✓ Montréal
✓ San Francisco

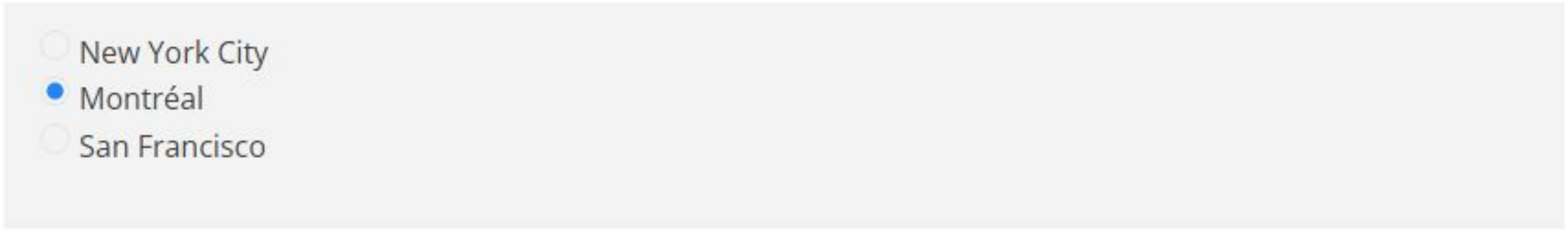
Checkboxes

```python
from dash import Dash, dcc, html

app = Dash()

app.layout = html.Div([
    dcc.Checklist(
        ['New York City', 'Montréal', 'San Francisco'],
        ['Montréal', 'San Francisco'],
        inline=True
    )
])

if __name__ == '__main__':
    app.run(debug=True)
```


☐ New York City ✓ Montréal ✓ San Francisco

Radio Items

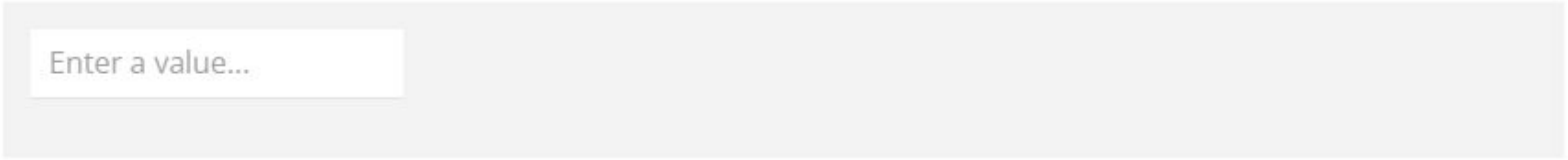
```python
from dash import Dash, dcc, html

app = Dash()

app.layout = html.Div([
    dcc.RadioItems(['New York City', 'Montréal', 'San Francisco'], 'Montréal')
])

if __name__ == '__main__':
    app.run(debug=True)
```



New York City
[✓ Montréal]
San Francisco

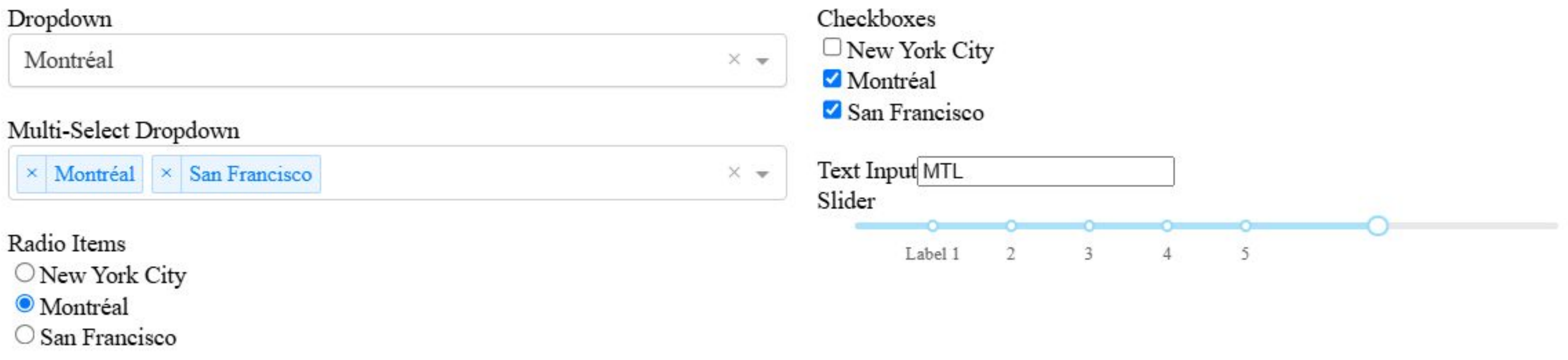
Input

```python
from dash import Dash, dcc, html

app = Dash()

app.layout = html.Div([
    dcc.Input(
        placeholder='Enter a value...',
        type='text',
        value=''
    )
])

if __name__ == '__main__':
    app.run(debug=True)
```



```python
# 2 - App layout

app2.layout = html.Div([
    html.Div(children=[
        html.Label('Dropdown'),
        dcc.Dropdown(['New York City', 'Montréal', 'San Francisco'], 'Montréal'),

        html.Br(),
        html.Label('Multi-Select Dropdown'),
        dcc.Dropdown(['New York City', 'Montréal', 'San Francisco'],
                     ['Montréal', 'San Francisco'],
                     multi=True),

        html.Br(),
        html.Label('Radio Items'),
```

```python
if __name__ == '__main__':
    app2.run_server(debug=True , port = 8051)
```



Dropdown
Montréal

Multi-Select Dropdown
× Montréal × San Francisco

Radio Items
○ New York City
● Montréal
○ San Francisco

Checkboxes
☐ New York City
☑ Montréal
☑ San Francisco

Text Input MTL

Slider

Interactivity with Callback function

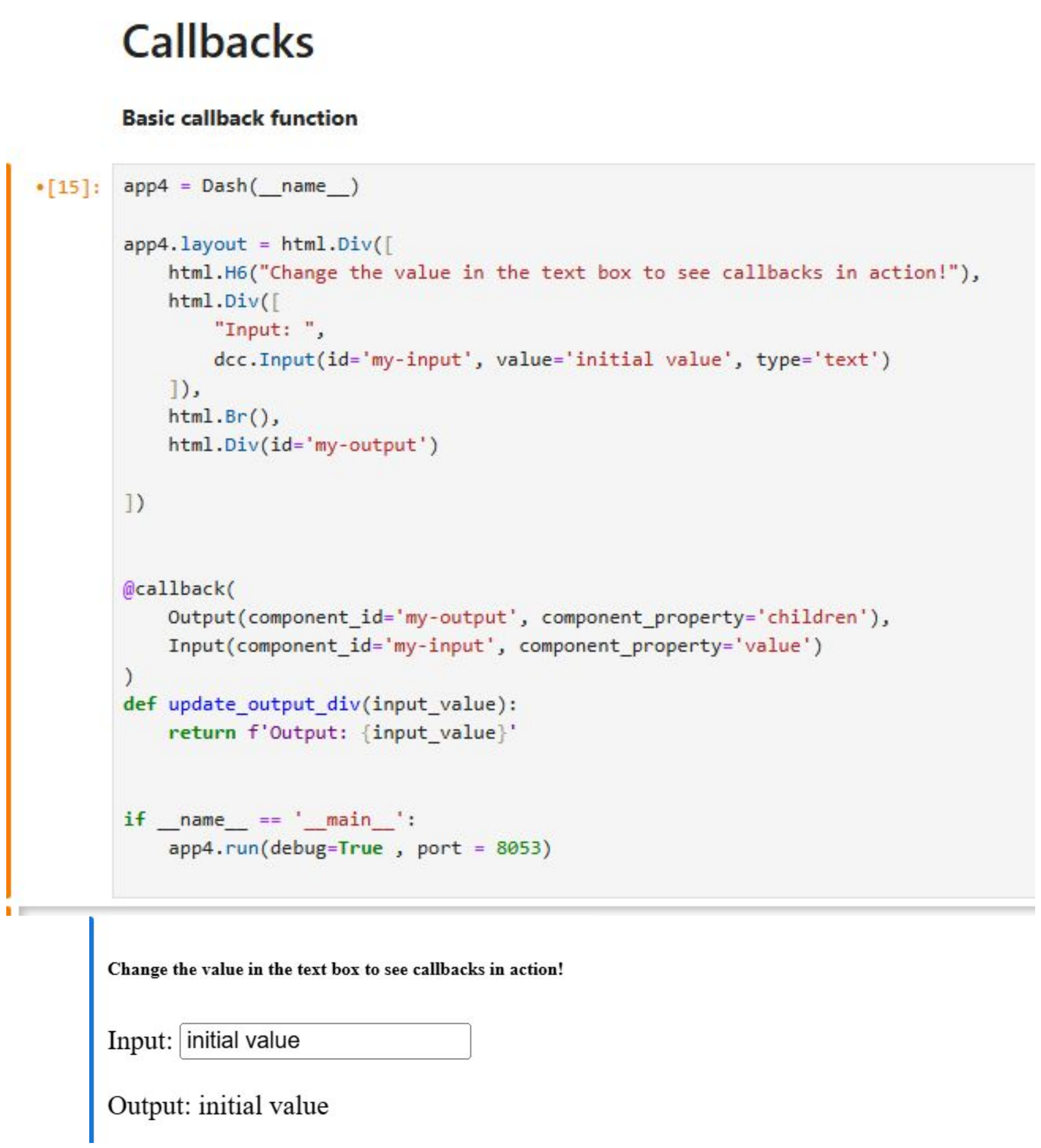
- Python function that is automatically called by Dash whenever an input component’s property changes
- Decorated with @app.callback Decorator
- Takes parameters as many as inputs
- Perform operations to return the desired result for the output component
- Return values as many as inputs

Callback function structure :

```
@app.callback(Output , Input , State)

def callback_function :
    ...
    return result
```

Output : Sets results returned from callback to a component id
Input : Sets input that is provided to a callback function to a component id
State : Like input but doesn’t trigger the function

Update text when write immediately



Callbacks

Basic callback function

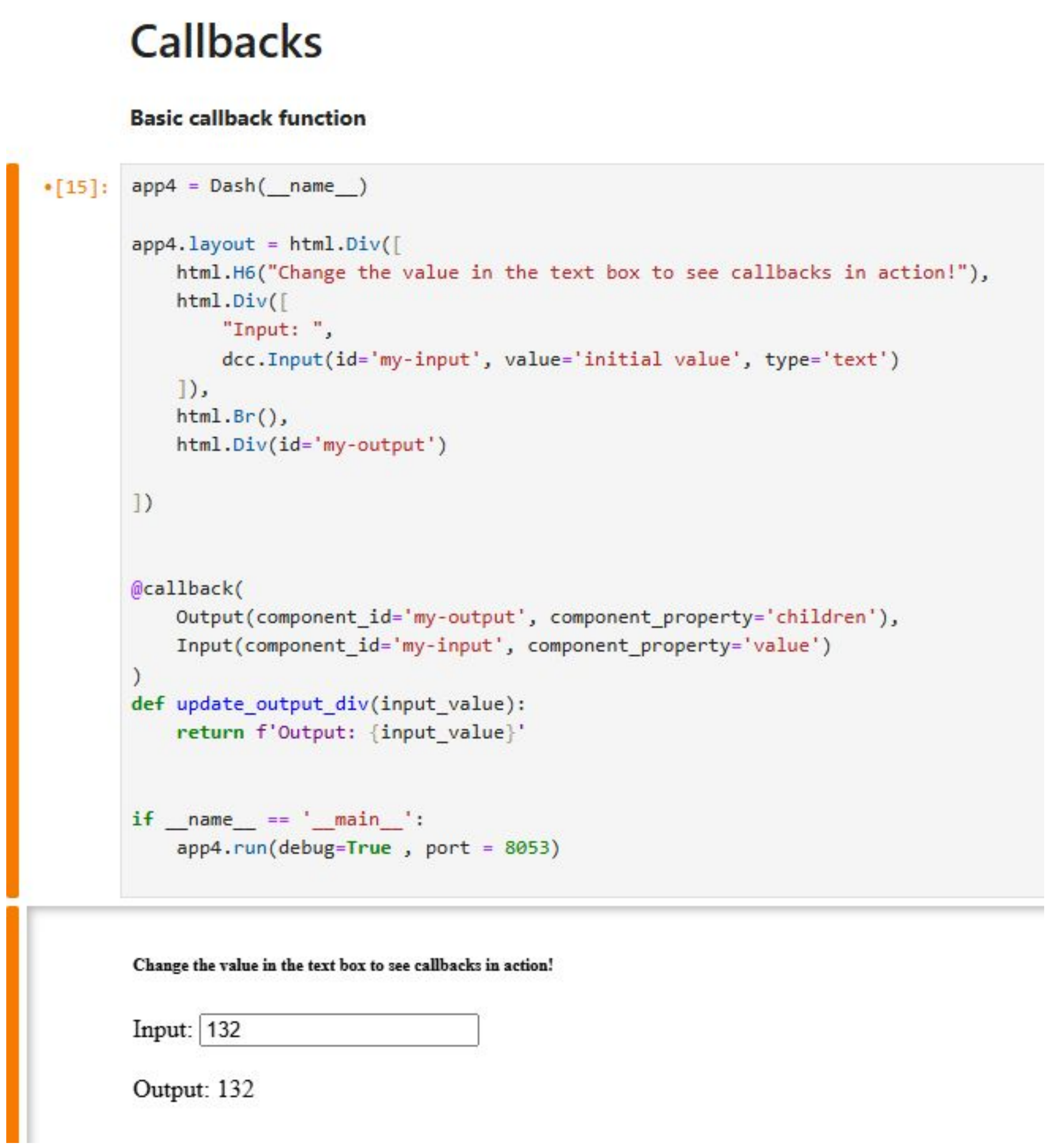
```
•[15]: app4 = Dash(__name__)

       app4.layout = html.Div([
           html.H6("Change the value in the text box to see callbacks in action!"),
           html.Div([
               "Input: ",
               dcc.Input(id='my-input', value='initial value', type='text')
           ]),
           html.Br(),
           html.Div(id='my-output')

       ])


       @callback(
           Output(component_id='my-output', component_property='children'),
           Input(component_id='my-input', component_property='value')
       )
       def update_output_div(input_value):
           return f'Output: {input_value}'


       if __name__ == '__main__':
           app4.run(debug=True , port = 8053)
```

Change the value in the text box to see callbacks in action!

Input: initial value

Output: initial value

Update text when write immediately



Callbacks

Basic callback function

•[15]:
```
app4 = Dash(__name__)

app4.layout = html.Div([
    html.H6("Change the value in the text box to see callbacks in action!"),
    html.Div([
        "Input: ",
        dcc.Input(id='my-input', value='initial value', type='text')
    ]),
    html.Br(),
    html.Div(id='my-output')

])


@callback(
    Output(component_id='my-output', component_property='children'),
    Input(component_id='my-input', component_property='value')
)
def update_output_div(input_value):
    return f'Output: {input_value}'


if __name__ == '__main__':
    app4.run(debug=True , port = 8053)
```

Change the value in the text box to see callbacks in action!

Input: 132

Output: 132

[64]:

```python
app4 = Dash(__name__)

app4.layout = html.Div([
    html.H6("Change the value in the text box to see callbacks in action!"),
    html.Div([
        "Input: ",
        dcc.Input(id='my-input', value='initial value', type='text')
    ]),
    html.Br(),
    html.Div(
        id='my-output')
])


@callback(
    Output(component_id='my-output', component_property='children'),
    Input(component_id='my-input', component_property='value')
)
def update_output_div(input_value):
    return f'Output: {input_value}'


if __name__ == '__main__':
    app4.run(debug=True , port = 8053)
```

Change the value in the text box to see callbacks in action!

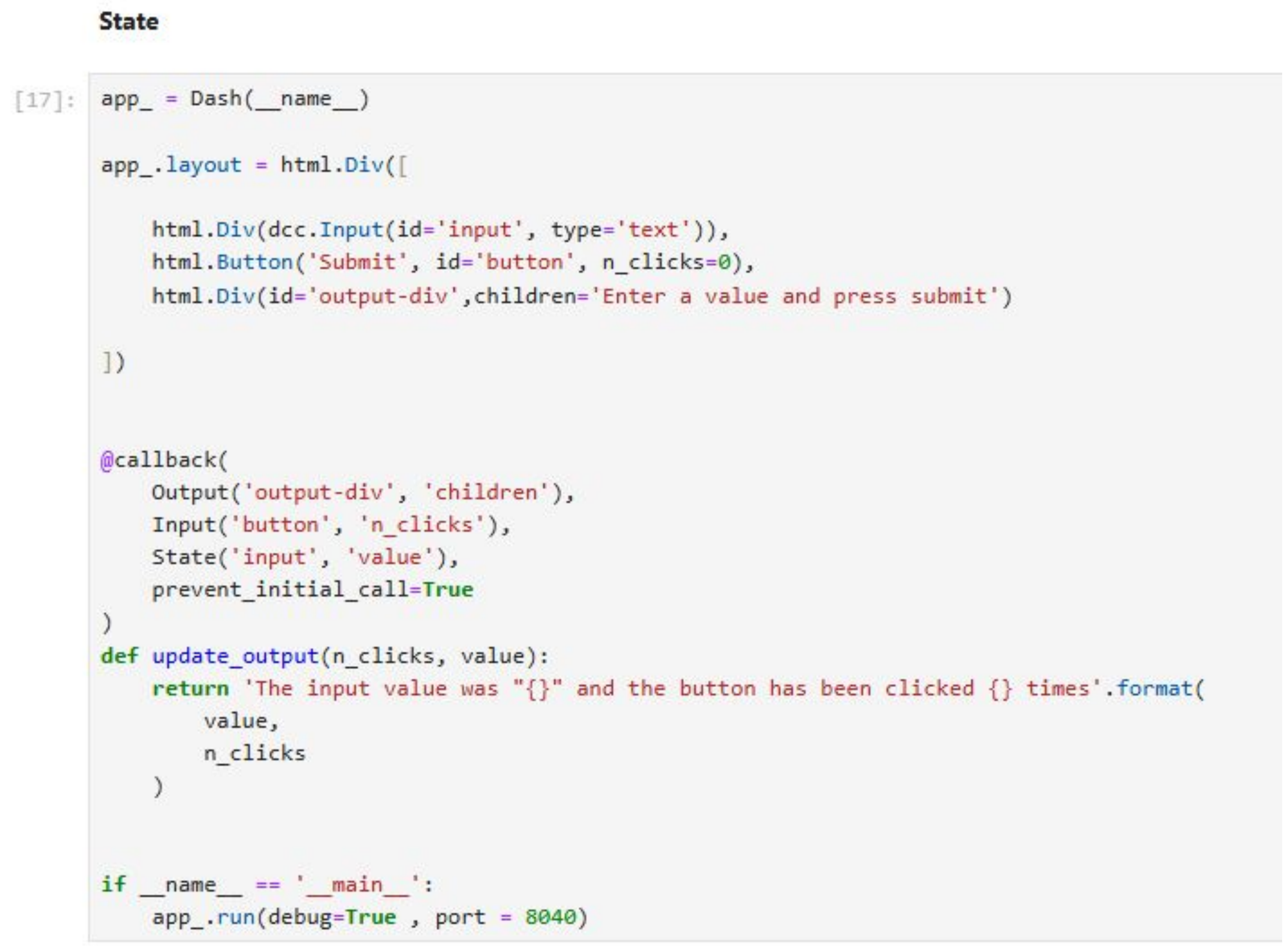
Input:

⚡ Waiting for input...

Use State: Prevent immediate update

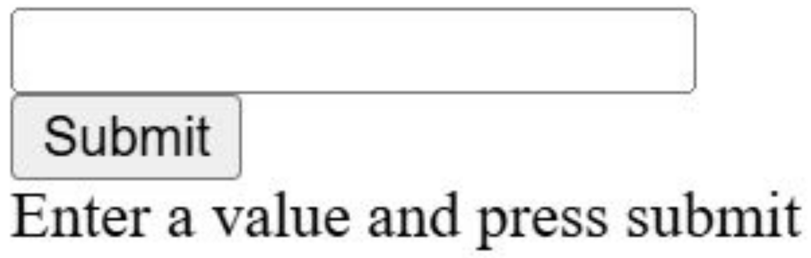
In some cases, we want to take user input but **not trigger the update immediately**.

Instead, we update the output **only when a button is clicked**. This can be achieved using `State` in Dash.



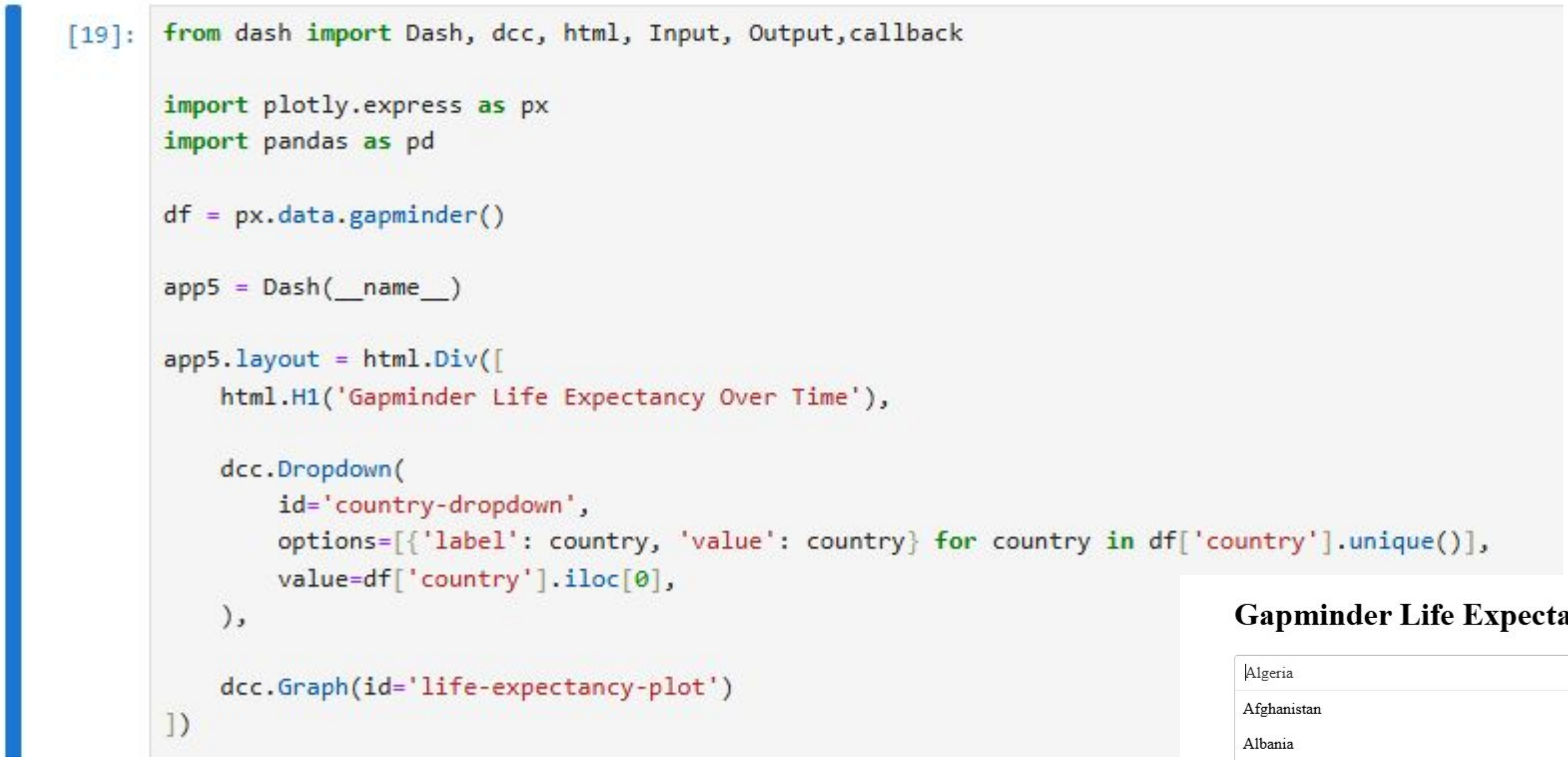
State

```
[17]: app_ = Dash(__name__)

      app_.layout = html.Div([

          html.Div(dcc.Input(id='input', type='text')),
          html.Button('Submit', id='button', n_clicks=0),
          html.Div(id='output-div',children='Enter a value and press submit')

      ])


      @callback(
          Output('output-div', 'children'),
          Input('button', 'n_clicks'),
          State('input', 'value'),
          prevent_initial_call=True
      )
      def update_output(n_clicks, value):
          return 'The input value was "{}" and the button has been clicked {} times'.format(
              value,
              n_clicks
          )


      if __name__ == '__main__':
          app_.run(debug=True , port = 8040)
```

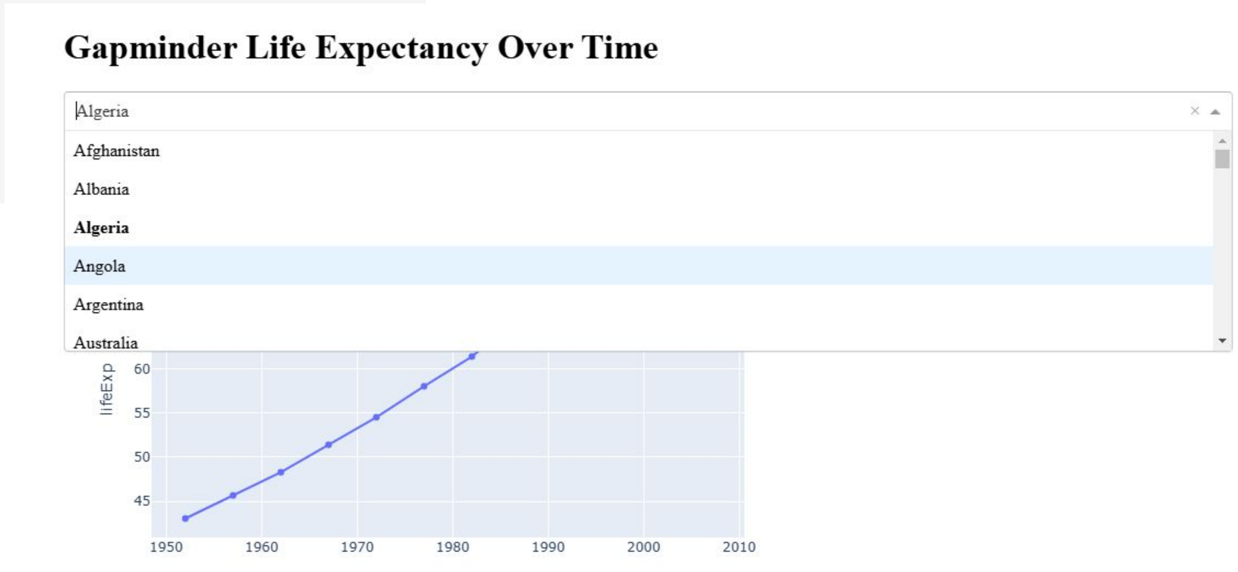


Enter a value and press submit

One input



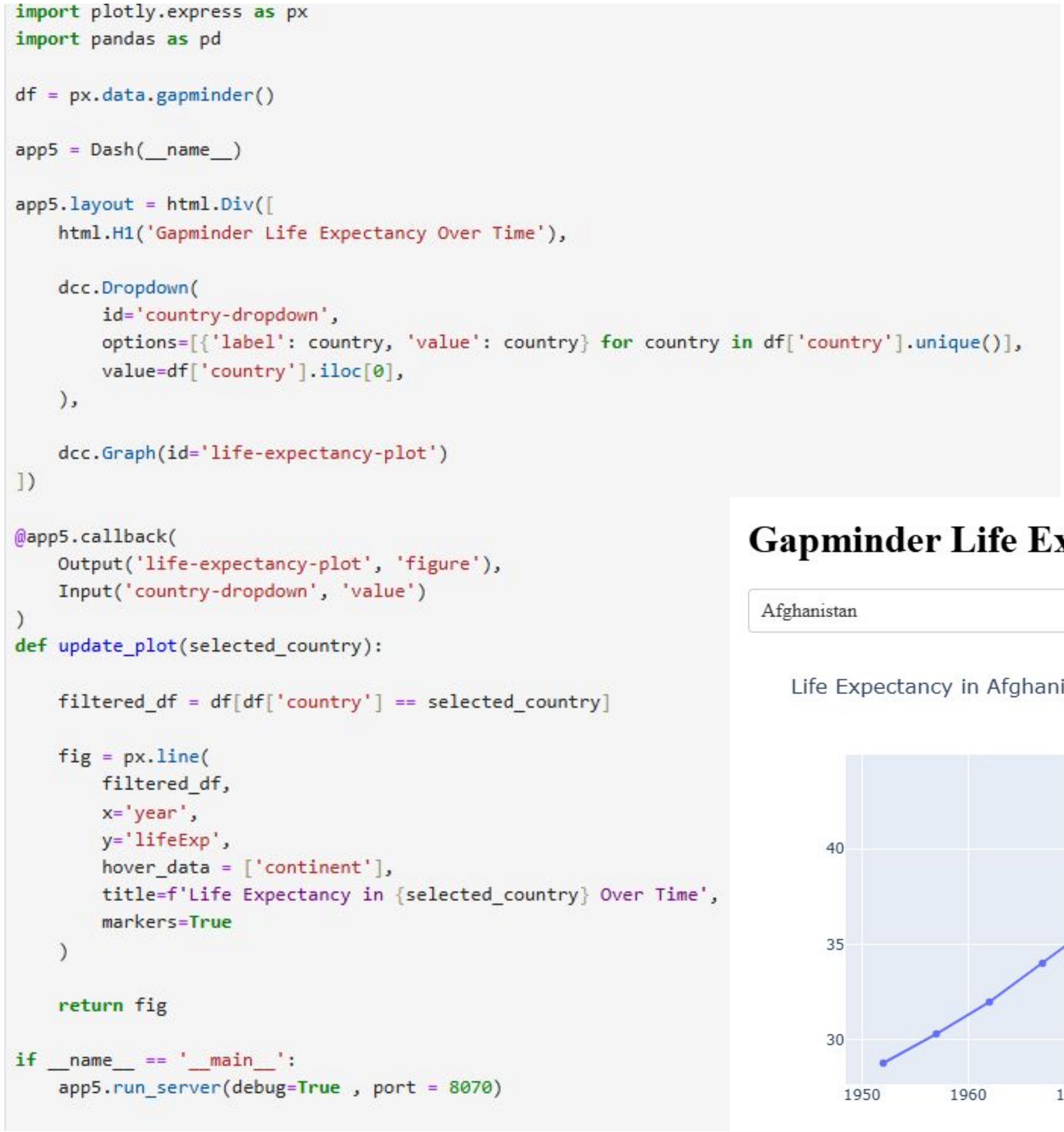
```
[19]: from dash import Dash, dcc, html, Input, Output,callback

      import plotly.express as px
      import pandas as pd

      df = px.data.gapminder()

      app5 = Dash(__name__)

      app5.layout = html.Div([
          html.H1('Gapminder Life Expectancy Over Time'),

          dcc.Dropdown(
              id='country-dropdown',
              options=[{'label': country, 'value': country} for country in df['country'].unique()],
              value=df['country'].iloc[0],
          ),

          dcc.Graph(id='life-expectancy-plot')
      ])
```



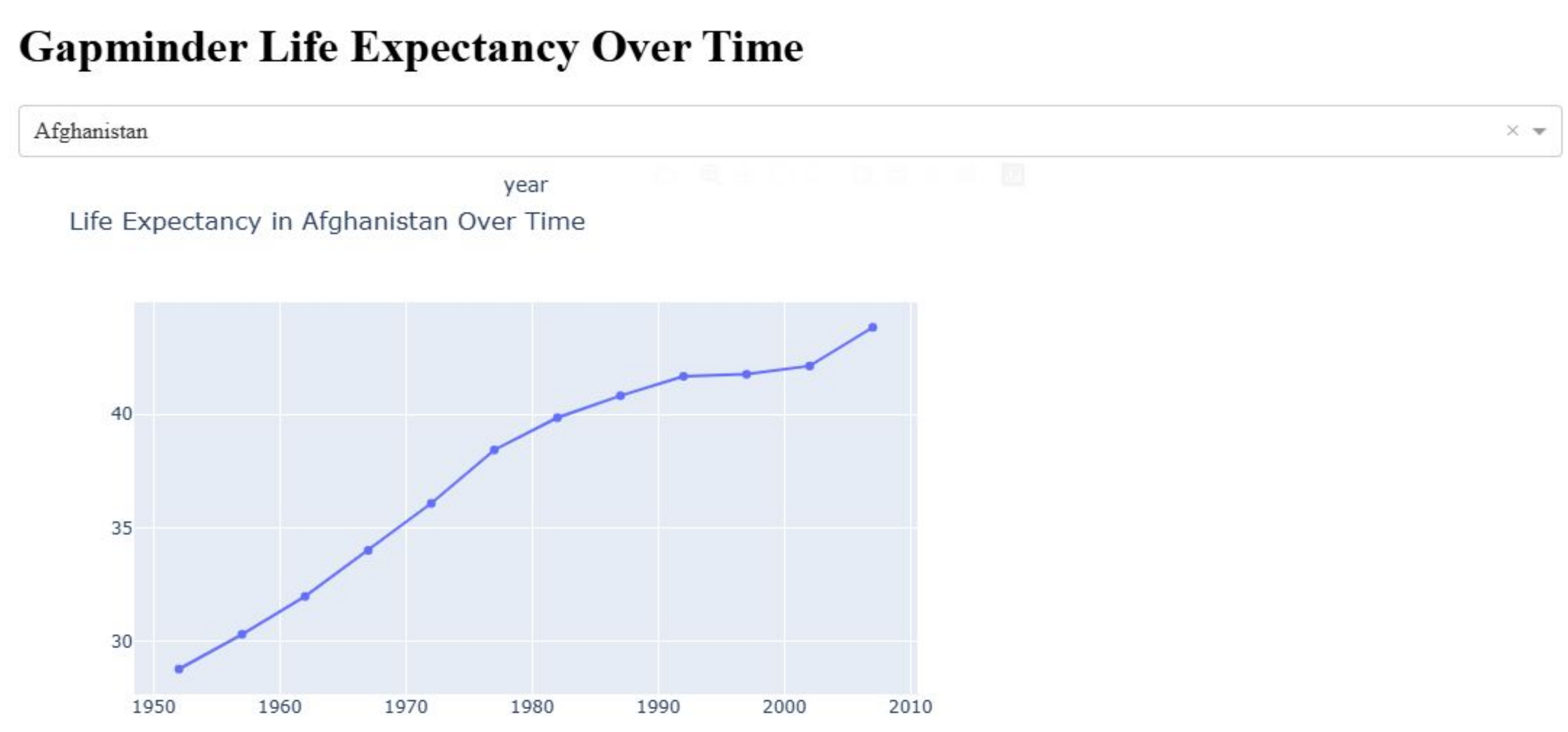
Gapminder Life Expectancy Over Time

Algeria

Afghanistan
Albania
Algeria
Angola
Argentina
Australia



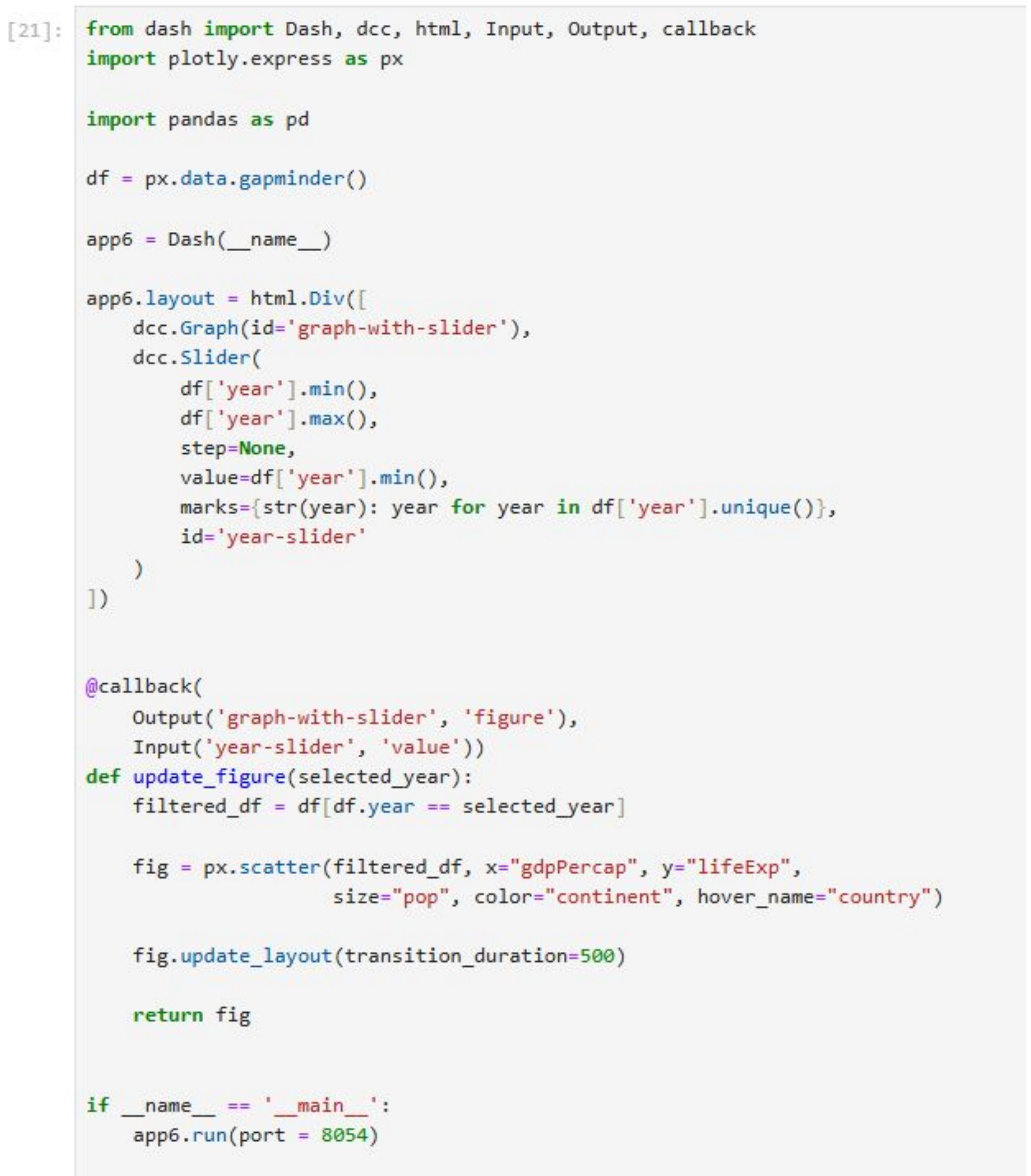
```
import plotly.express as px
import pandas as pd

df = px.data.gapminder()

app5 = Dash(__name__)

app5.layout = html.Div([
    html.H1('Gapminder Life Expectancy Over Time'),

    dcc.Dropdown(
        id='country-dropdown',
        options=[{'label': country, 'value': country} for country in df['country'].unique()],
        value=df['country'].iloc[0],
    ),

    dcc.Graph(id='life-expectancy-plot')
])

@app5.callback(
    Output('life-expectancy-plot', 'figure'),
    Input('country-dropdown', 'value')
)
def update_plot(selected_country):

    filtered_df = df[df['country'] == selected_country]

    fig = px.line(
        filtered_df,
        x='year',
        y='lifeExp',
        hover_data = ['continent'],
        title=f'Life Expectancy in {selected_country} Over Time',
        markers=True
    )

    return fig

if __name__ == '__main__':
    app5.run_server(debug=True , port = 8070)
```



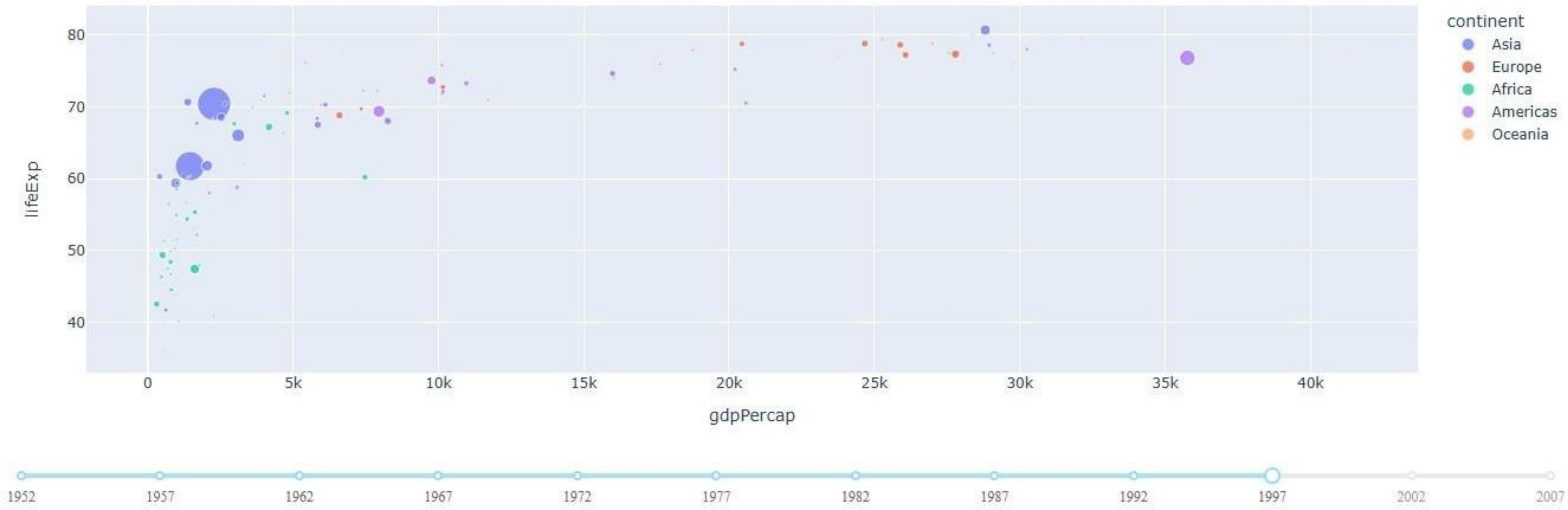
Gapminder Life Expectancy Over Time

Afghanistan

One input



```
[21]:
from dash import Dash, dcc, html, Input, Output, callback
import plotly.express as px

import pandas as pd

df = px.data.gapminder()

app6 = Dash(__name__)

app6.layout = html.Div([
    dcc.Graph(id='graph-with-slider'),
    dcc.Slider(
        df['year'].min(),
        df['year'].max(),
        step=None,
        value=df['year'].min(),
        marks={str(year): year for year in df['year'].unique()},
        id='year-slider'
    )
])


@callback(
    Output('graph-with-slider', 'figure'),
    Input('year-slider', 'value'))
def update_figure(selected_year):
    filtered_df = df[df.year == selected_year]

    fig = px.scatter(filtered_df, x="gdpPercap", y="lifeExp",
                     size="pop", color="continent", hover_name="country")

    fig.update_layout(transition_duration=500)

    return fig


if __name__ == '__main__':
    app6.run(port = 8054)
```

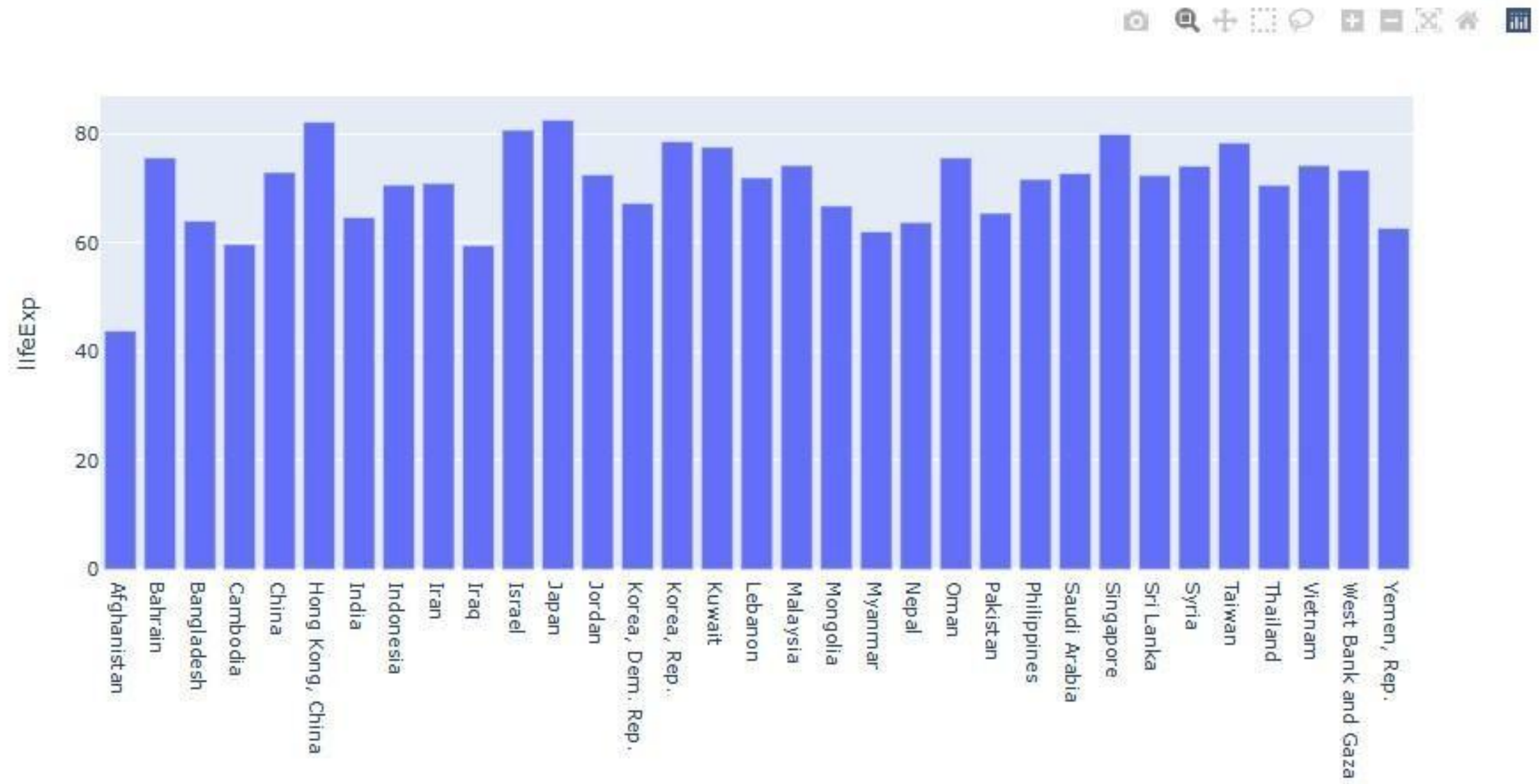


continent
Asia
Europe
Africa
Americas
Oceania
lifeExp
40
50
60
70
80
gdpPercap
0
5k
10k
15k
20k
25k
30k
35k
40k
1952
1957
1962
1967
1972
1977
1982
1987
1992
1997
2002
2007

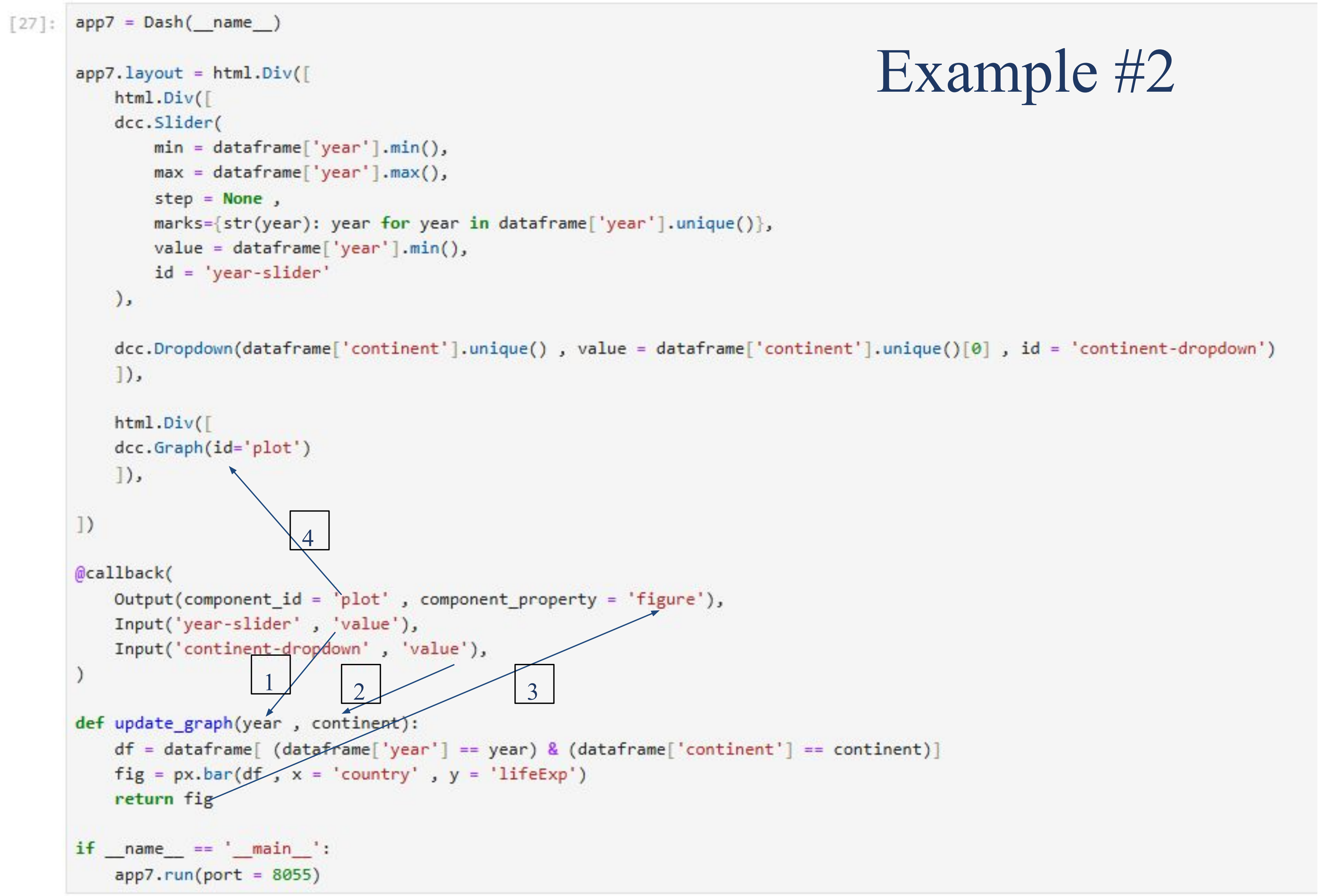
Using Multiple Inputs to Update Multiple Data Points

Multiple inputs

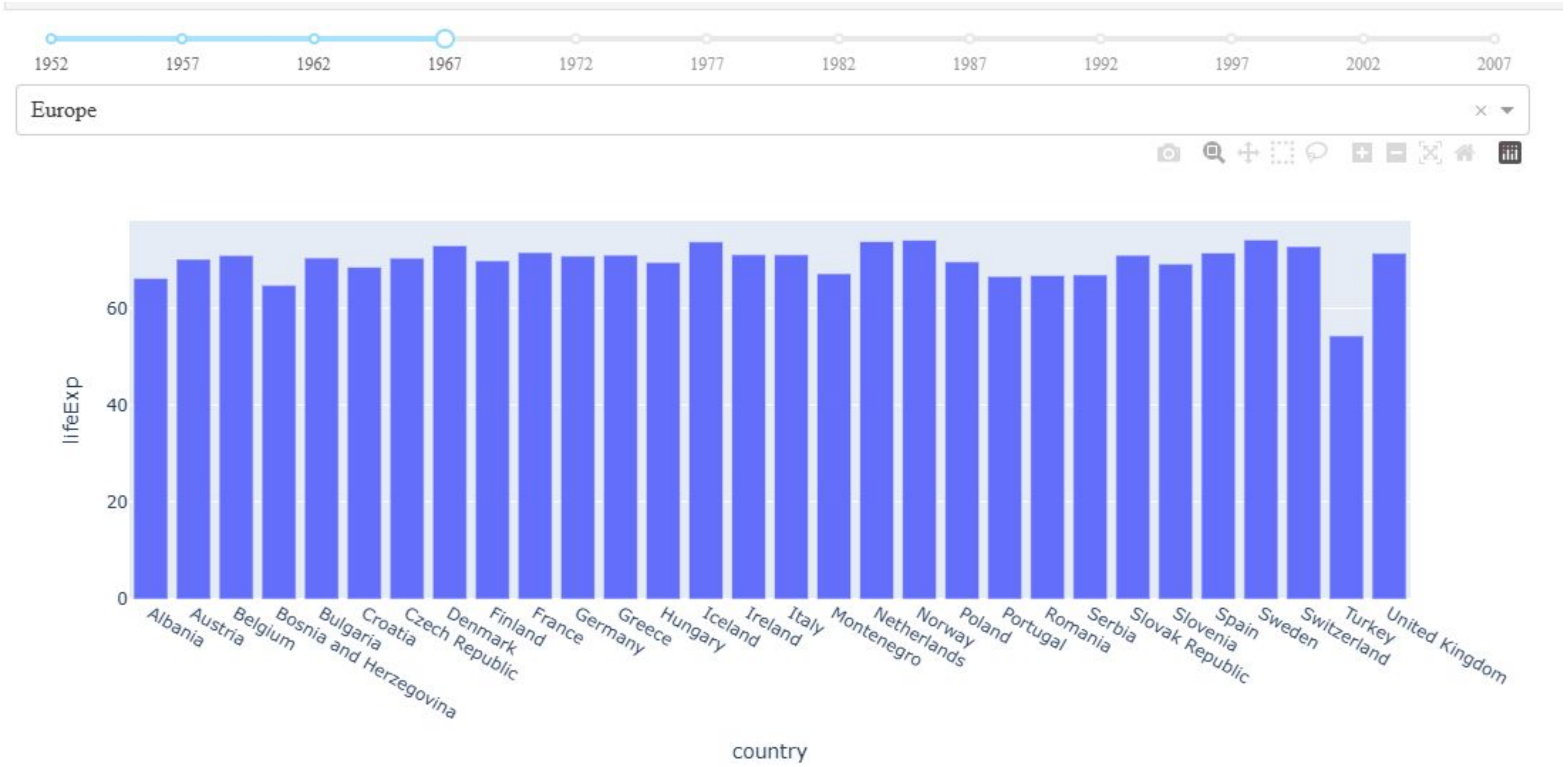
```
dataframe = px.data.gapminder()
figure = px.bar(dataframe[(dataframe['continent'] == 'Asia') & (dataframe['year'] == 2007)] , x ='country' , y = 'lifeExp')
figure
```



Example #2

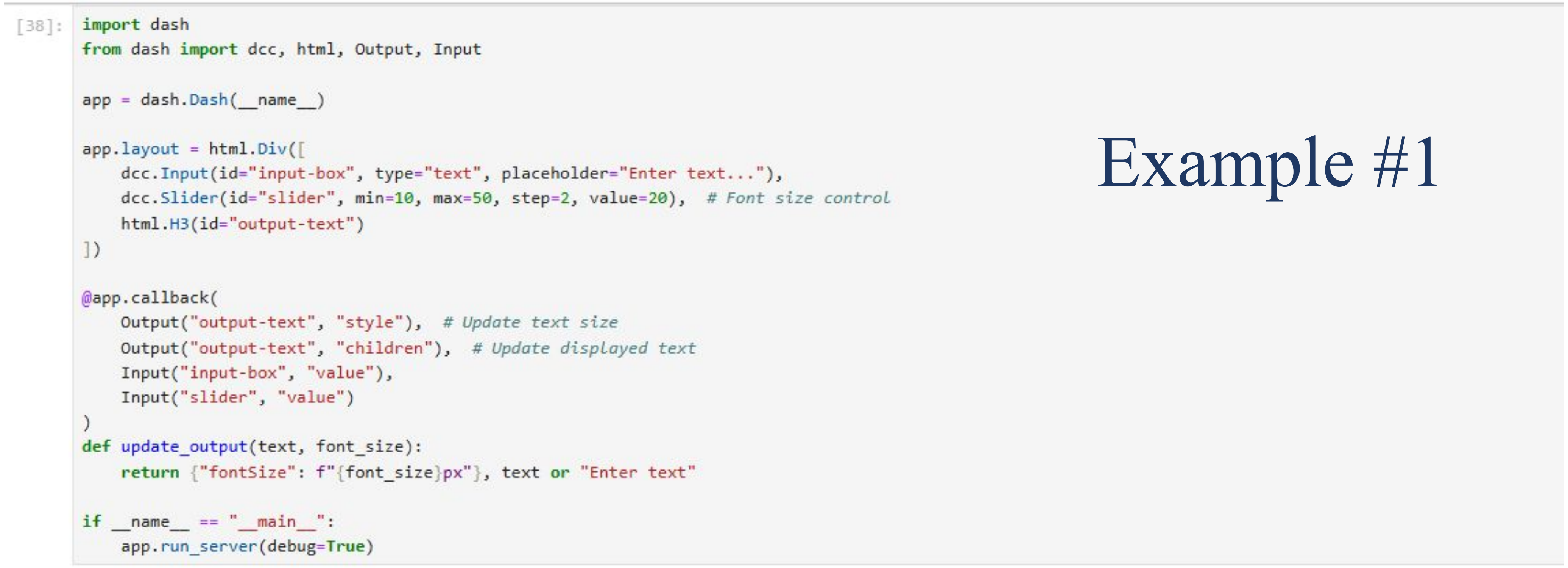


```
[27]: app7 = Dash(__name__)

      app7.layout = html.Div([
          html.Div([
          dcc.Slider(
              min = dataframe['year'].min(),
              max = dataframe['year'].max(),
              step = None ,
              marks={str(year): year for year in dataframe['year'].unique()},
              value = dataframe['year'].min(),
              id = 'year-slider'
          ),

          dcc.Dropdown(dataframe['continent'].unique() , value = dataframe['continent'].unique()[0] , id = 'continent-dropdown')
          ]),

          html.Div([
          dcc.Graph(id='plot')
          ]),

      ])

      @callback(
          Output(component_id = 'plot' , component_property = 'figure'),
          Input('year-slider' , 'value'),
          Input('continent-dropdown' , 'value'),
      )

      def update_graph(year , continent):
          df = dataframe[ (dataframe['year'] == year) & (dataframe['continent'] == continent)]
          fig = px.bar(df , x = 'country' , y = 'lifeExp')
          return fig

      if __name__ == '__main__':
          app7.run(port = 8055)
```


1952
1957
1962
1967
1972
1977
1982
1987
1992
1997
2002
2007
Europe
lifeExp
0
20
40
60
country
Albania
Austria
Belgium
Bosnia and Herzegovina
Bulgaria
Croatia
Czech Republic
Denmark
Finland
France
Germany
Greece
Hungary
Iceland
Ireland
Italy
Montenegro
Netherlands
Norway
Poland
Portugal
Romania
Serbia
Slovak Republic
Slovenia
Spain
Sweden
Switzerland
Turkey
United Kingdom

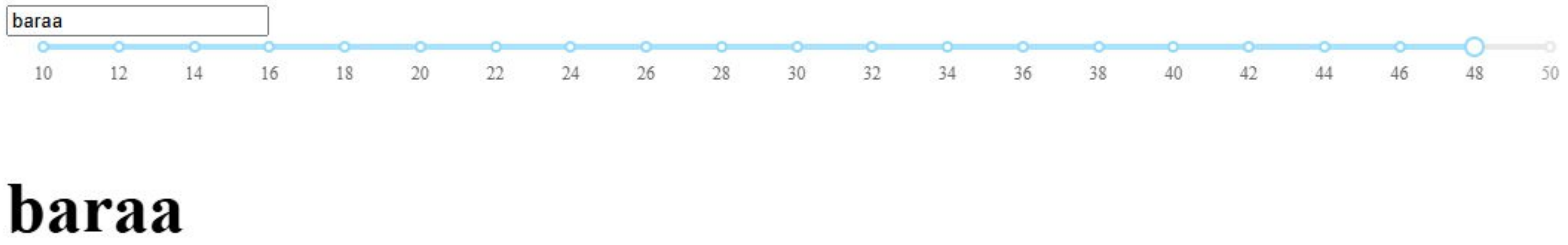
Using Multiple Inputs to Update Multiple Data Points

A callback function in Dash can take **multiple inputs** and update multiple elements in the UI.



```
[38]: import dash
      from dash import dcc, html, Output, Input

      app = dash.Dash(__name__)

      app.layout = html.Div([
          dcc.Input(id="input-box", type="text", placeholder="Enter text..."),
          dcc.Slider(id="slider", min=10, max=50, step=2, value=20),  # Font size control
          html.H3(id="output-text")
      ])

      @app.callback(
          Output("output-text", "style"),  # Update text size
          Output("output-text", "children"),  # Update displayed text
          Input("input-box", "value"),
          Input("slider", "value")
      )
      def update_output(text, font_size):
          return {"fontSize": f"{font_size}px"}, text or "Enter text"

      if __name__ == "__main__":
          app.run_server(debug=True)
```

Example #1

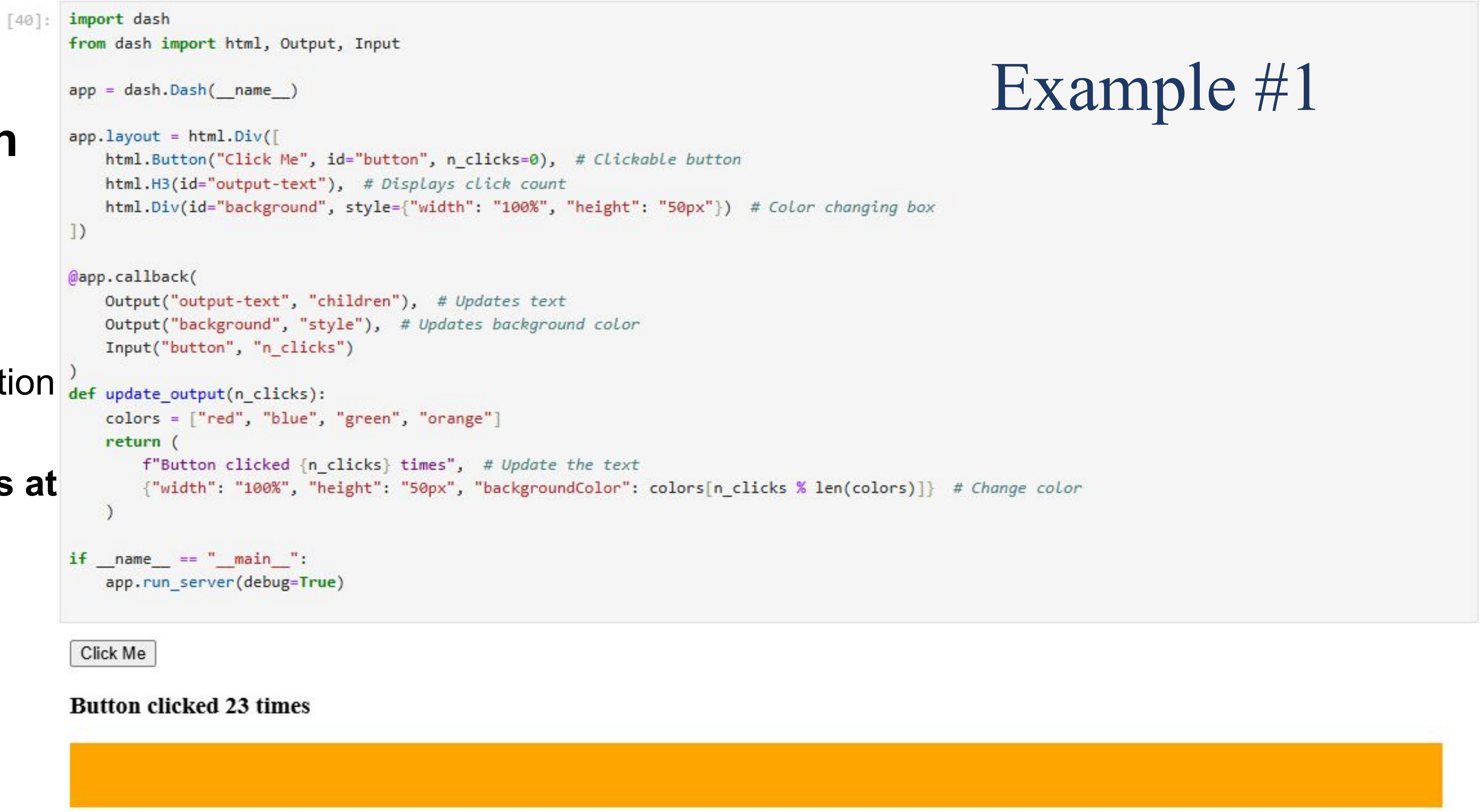


Using Multiple Outputs in Dash Callbacks

Using **Multiple Outputs** in Dash Callbacks

Example #1

A single callback function in Dash can **update multiple components at the same time.**



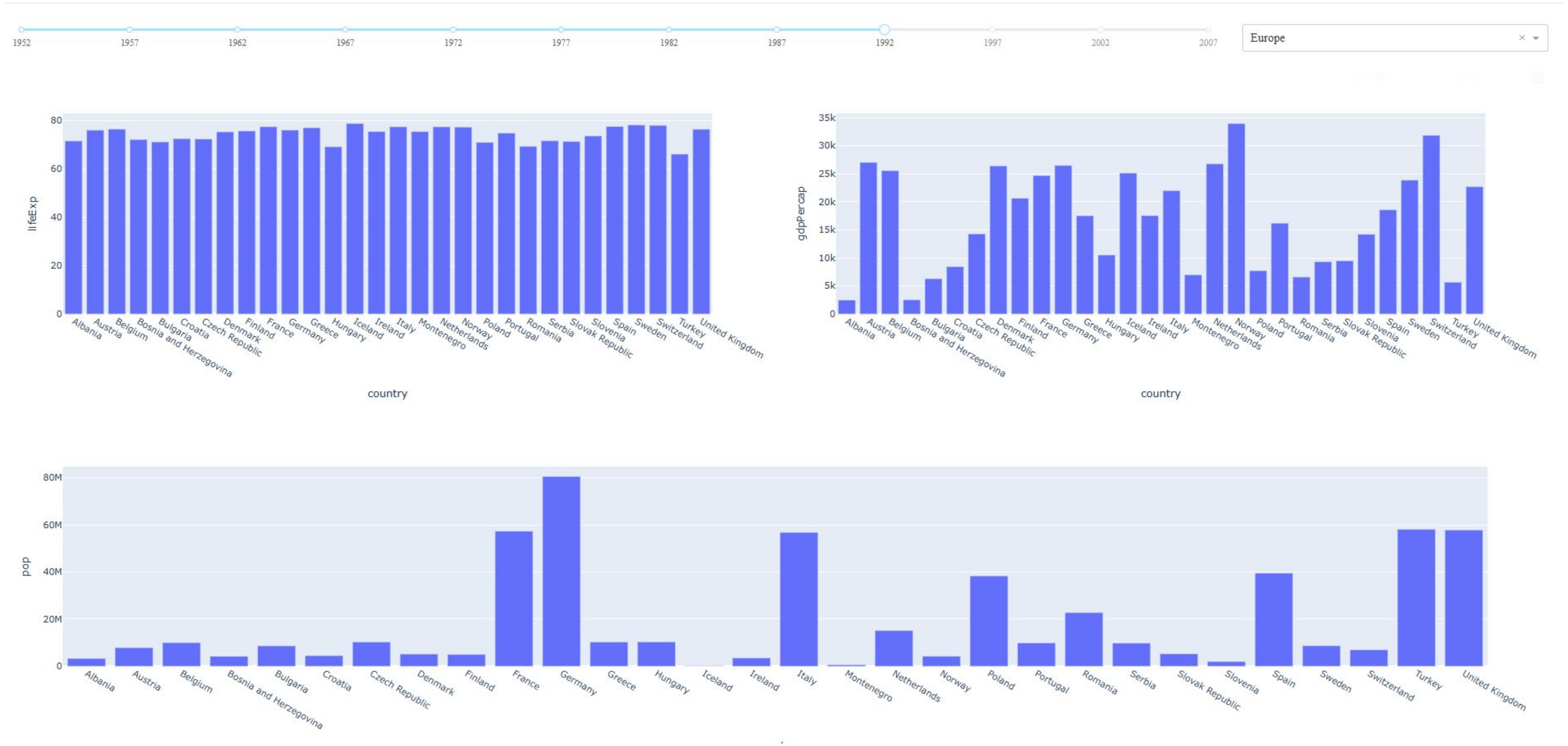
[40]:

```python
import dash
from dash import html, Output, Input

app = dash.Dash(__name__)

app.layout = html.Div([
    html.Button("Click Me", id="button", n_clicks=0),  # Clickable button
    html.H3(id="output-text"),  # Displays click count
    html.Div(id="background", style={"width": "100%", "height": "50px"})  # Color changing box
])

@app.callback(
    Output("output-text", "children"),  # Updates text
    Output("background", "style"),  # Updates background color
    Input("button", "n_clicks")
)
def update_output(n_clicks):
    colors = ["red", "blue", "green", "orange"]
    return (
        f"Button clicked {n_clicks} times",  # Update the text
        {"width": "100%", "height": "50px", "backgroundColor": colors[n_clicks % len(colors)]}  # Change color
    )

if __name__ == "__main__":
    app.run_server(debug=True)
```

Click Me

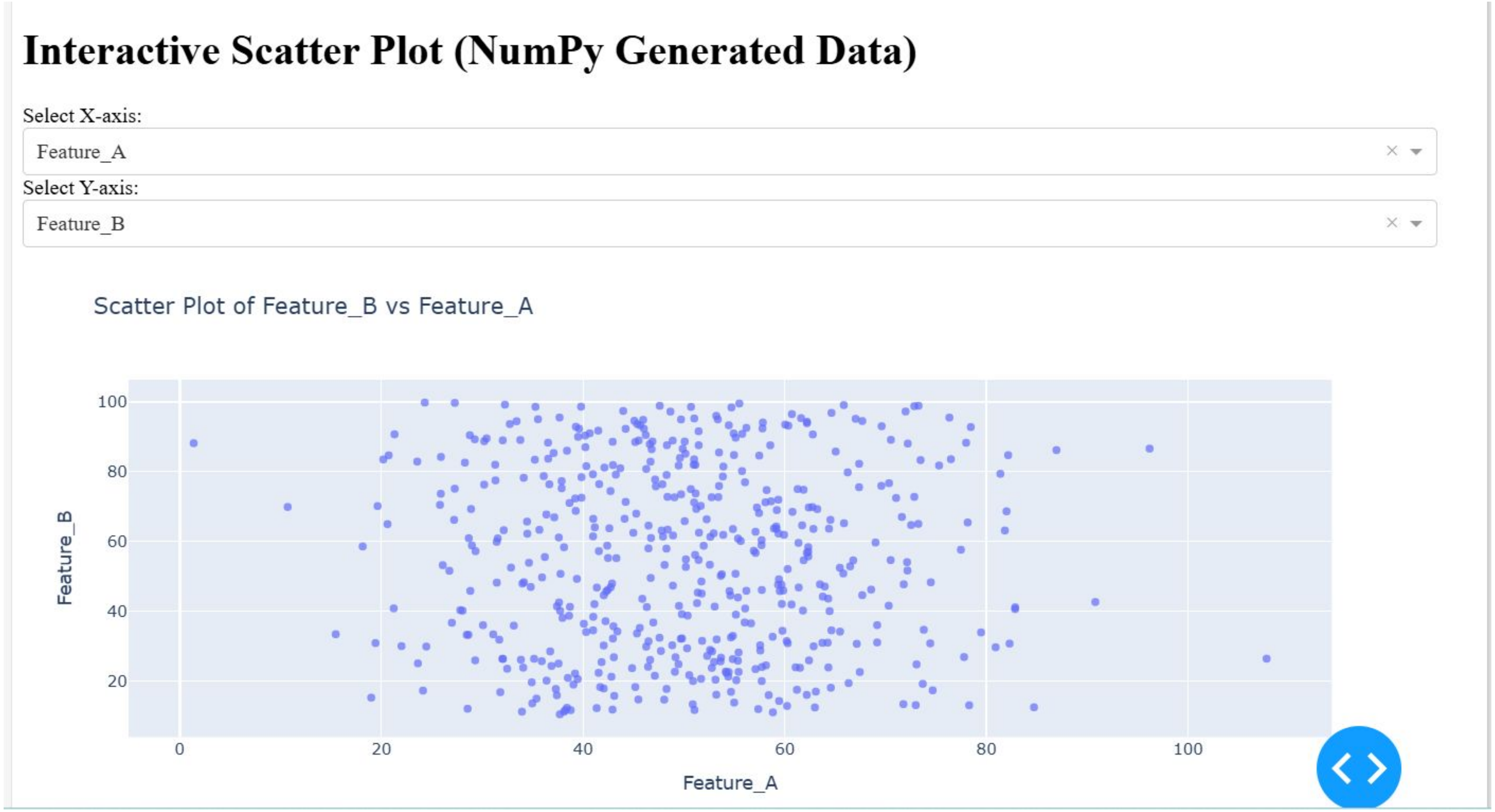
Button clicked 23 times

```
app7.layout = html.Div([
    html.Div([
    dcc.Slider(
        min = dataframe['year'].min(),
        max = dataframe['year'].max(),
        step = None ,
        marks={str(year): year for year in dataframe['year'].unique()},
        value = dataframe['year'].min(),
        id = 'year-slider'
    ),

    dcc.Dropdown(dataframe['continent'].unique() , value = dataframe['continent'].unique()[0] , id = 'continent-dropdown')
    ]),

    html.Div([
    dcc.Graph(id='plot')
    ]),

])

@callback(
    Output(component_id = 'plot' , component_property = 'figure'),
    Input('year-slider' , 'value'),
    Input('continent-dropdown' , 'value'),
)

def update_graph(year , continent):
    df = dataframe[ (dataframe['year'] == year) & (dataframe['continent'] == continent)]
    fig = px.bar(df , x = 'country' , y = 'lifeExp')
    return fig
```


1952
1957
1962
1967
1972
1977
1982
1987
1992
1997
2002
2007
Europe
lifeExp
0
20
40
60
80
country
gdpPercap
0
5k
10k
15k
20k
25k
30k
35k
country
pop
0
20M
40M
60M
80M
Albania
Austria
Belgium
Bosnia and Herzegovina
Bulgaria
Croatia
Czech Republic
Denmark
Finland
France
Germany
Greece
Hungary
Iceland
Ireland
Italy
Montenegro
Netherlands
Norway
Poland
Portugal
Romania
Serbia
Slovak Republic
Slovenia
Spain
Sweden
Switzerland
Turkey
United Kingdom

Tasks in classroom

Task 4 output: Interactive Scatter Plot with User Controls using NumPy & Dash



Interactive Scatter Plot (NumPy Generated Data)

Select X-axis:

Feature_A

Select Y-axis:

Feature_B