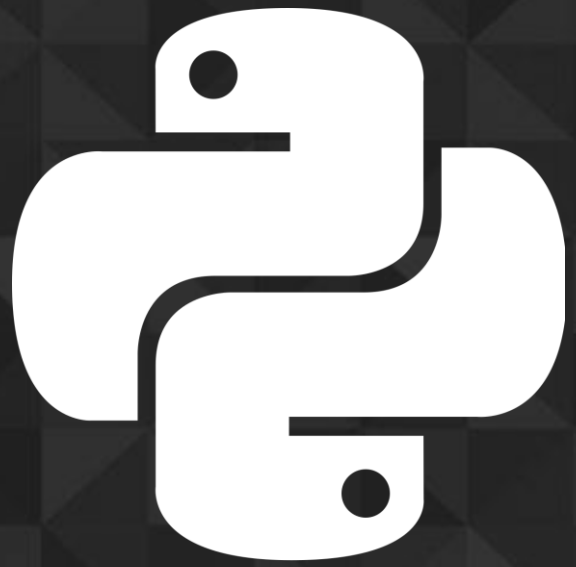


# DASHBOARDS WITH PLOTLY & DASH

# COURSE OUTLINE

| | | |
|---|---|---|
| 1 | **Intro to Plotly & Dash** | *Introduce the Plotly & Dash libraries, and cover the key steps & components for creating a basic Dash application with interactive Plotly visuals* |
| 2 | **Plotly Figures & Chart Types** | *Dive deep into the Plotly library and use it to build & customize several chart types, including line charts, bar charts, pie charts, scatterplots, and histograms* |
| 3 | **Interactive Elements** | *Get comfortable embedding Dash's interactive elements into your application, and using them to manipulate Plotly Visualizations* |

**PROJECT:** Interactive Ski Resort Visualizations

| | | |
|---|---|---|
| 4 | **Dashboard Layouts** | *Learn how to organize your visualizations and interactive components into a visually appealing and logical structure* |
| 5 | **Advanced Functionality** | *Take your applications to the next level by learning how to update your application with real-time data, develop chained-callback functions, and more!* |

**PROJECT:** Building A Dashboard to Help Agents Find The Best Ski resorts For Customers

# THE COURSE PROJECT

## THE SITUATION

You've just been hired to join the Analytics team at **Maveluxe Travel**, a high-end travel agency with best-in-class services for helping customers find the perfect resorts based on their preferences

## THE ASSIGNMENT

Your task is to **build interactive visuals and dashboards** to help Maveluxe's travel agents pick the best ski resorts for their customers based on their very picky tastes. We will build a dashboard that includes interactive maps and report cards for ski resorts around the world.

## THE OBJECTIVES

- Use Pandas to read & manipulate multiple datasets
- Use Plotly to visualize data, communicate insights, and help users get the information they need
- Use Dash to add interactivity to the Plotly visuals, then build and deploy comprehensive dashboards

MAVELUXE

# SETTING EXPECTATIONS

## This course covers the **core functionality** for Plotly & Dash

- *We'll focus on adding interactivity to charts and constructing dashboard applications*

## We'll focus on creating **interactive** visuals & dashboards

- *We will cover Plotly chart types enough to get you comfortable with them, but we won't dive into extreme customization. Instead, our focus is interactivity.*

## We'll use **Jupyter Notebooks** as our primary coding environment

- *Jupyter Notebooks are free to use, and the industry standard for conducting data analysis with Python (we'll introduce Google Colab as an alternative, cloud-based environment as well)*

## This course **requires previous knowledge** of Python and Pandas

- *It is strongly recommended that you complete the 3 previous courses in our Python specialist path, or have a solid understanding of base Python, DataFrame manipulation with Pandas, and visualization with Matplotlib or Seaborn*

# INSTALLATION & SETUP

# INSTALLING ANACONDA (MAC)

Installing Anaconda

Launching Jupyter

Google Colab

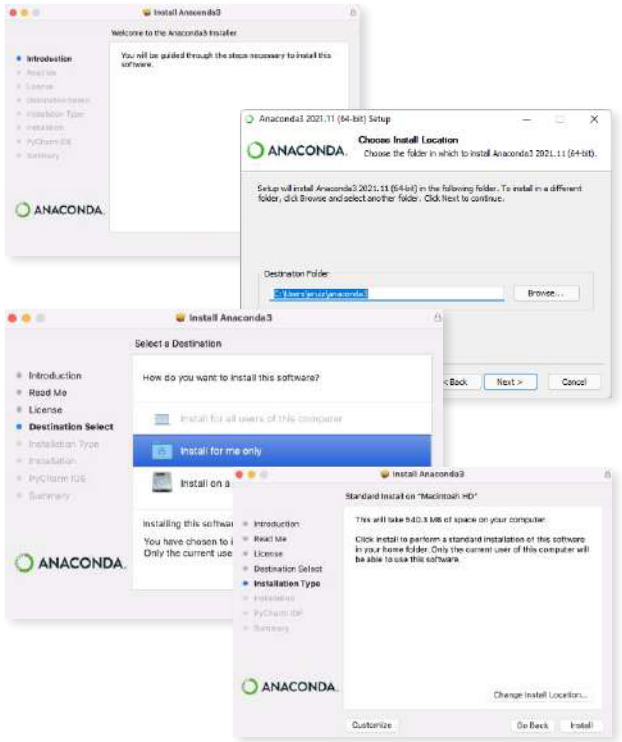
1) Go to **anaconda.com/products/distribution** and click 



2) Click **X** on the Anaconda Nucleus pop-up
*(no need to launch)*


3) Launch the downloaded Anaconda **pkg** file

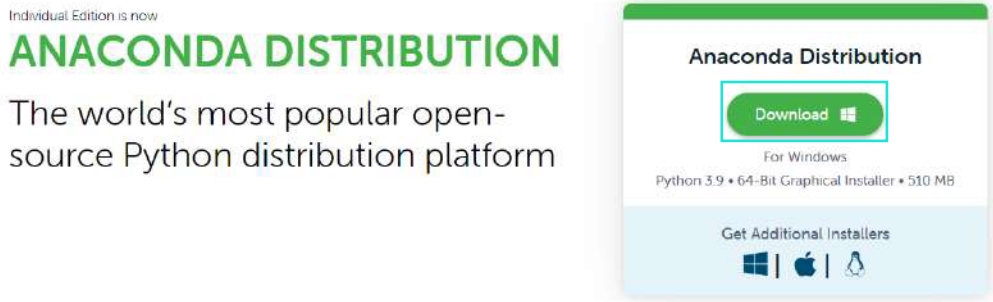

4) Follow the **installation steps**
*(default settings are OK)*



# INSTALLING ANACONDA (PC)

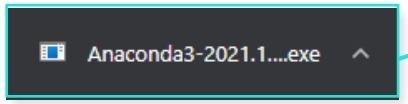


1) Go to **anaconda.com/products/distribution** and click Download

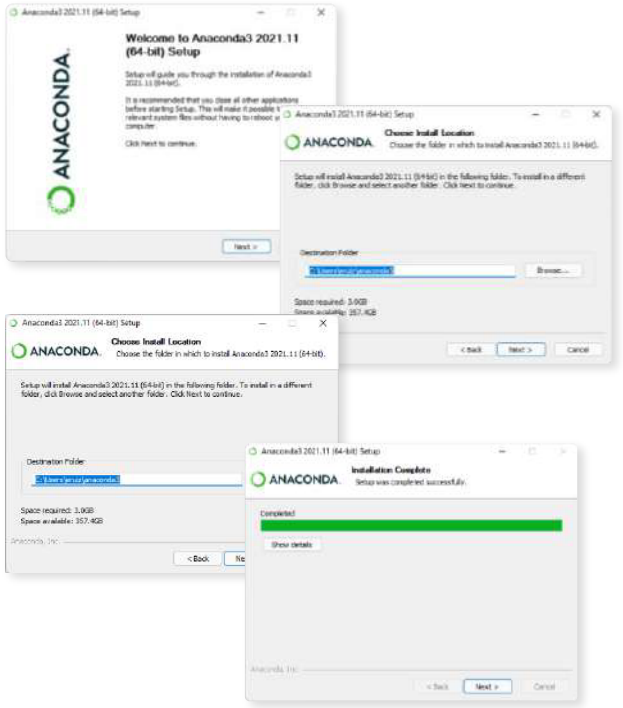



2) Click **X** on the Anaconda Nucleus pop-up
*(no need to launch)*


3) Launch the downloaded Anaconda **exe** file




4) Follow the **installation steps**
*(default settings are OK)*



# LAUNCHING JUPYTER

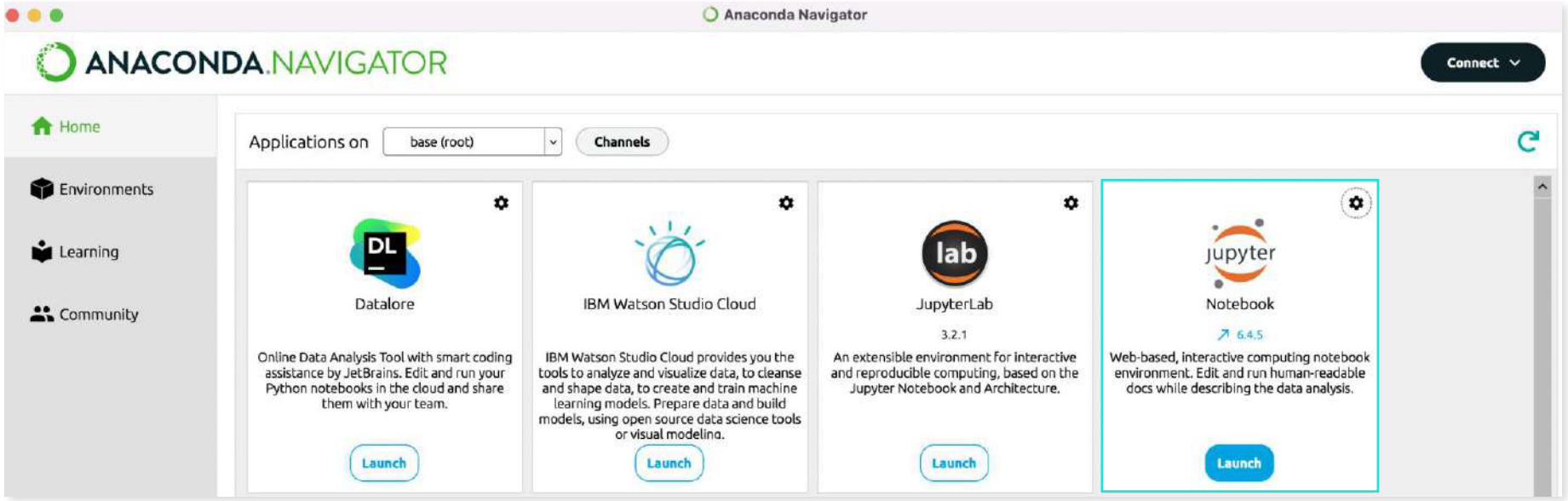
Installing Anaconda

Launching Jupyter

Google Colab

1) Launch **Anaconda Navigator**

2) Find **Jupyter Notebook** and click Launch



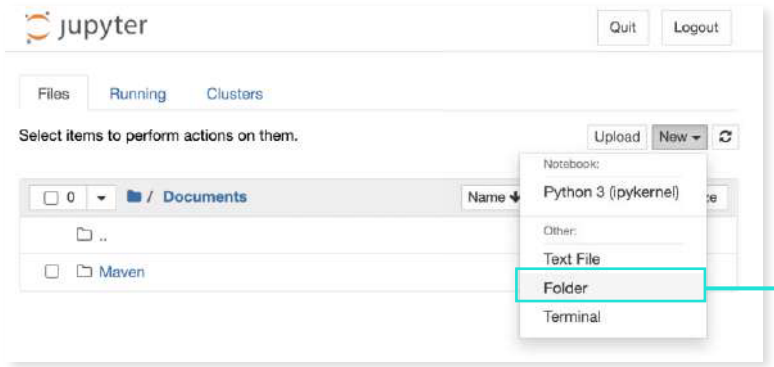
# YOUR FIRST JUPYTER NOTEBOOK

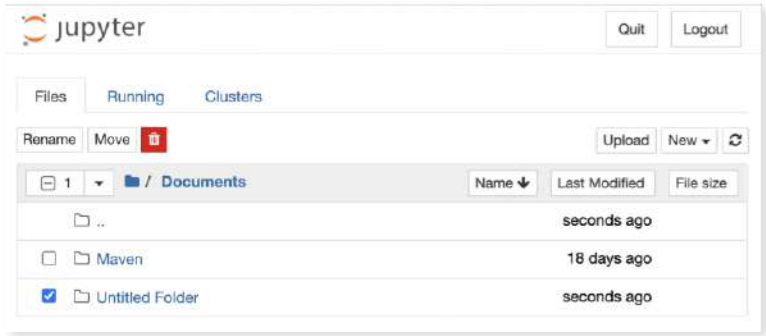
Installing Anaconda

Launching Jupyter

Google Colab

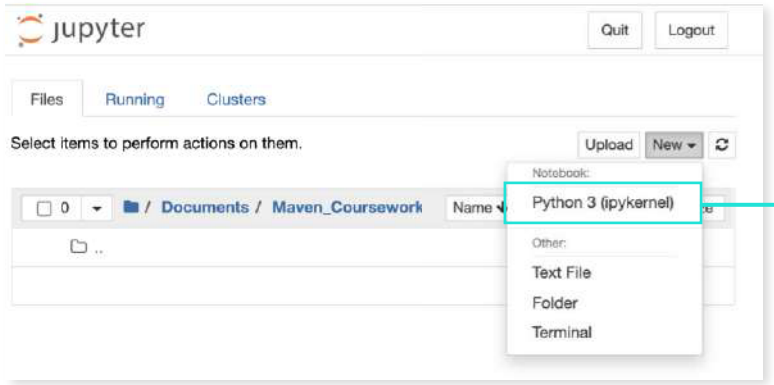
1) Once inside the Jupyter interface, **create a folder** to store your notebooks for the course

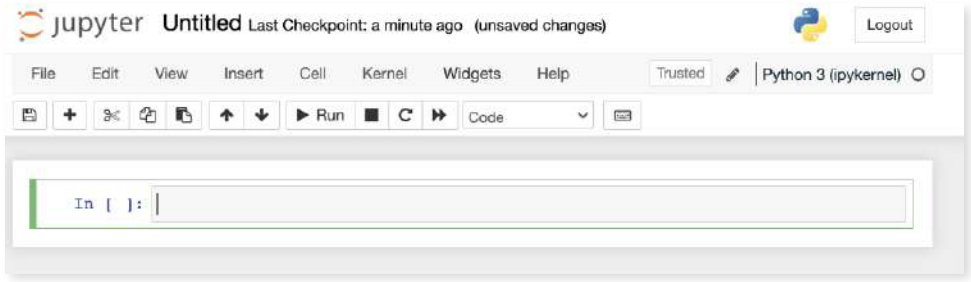



***NOTE:*** *You can rename your folder by clicking "Rename" in the top left corner*

2) Open your new coursework folder and **launch your first Jupyter notebook**!



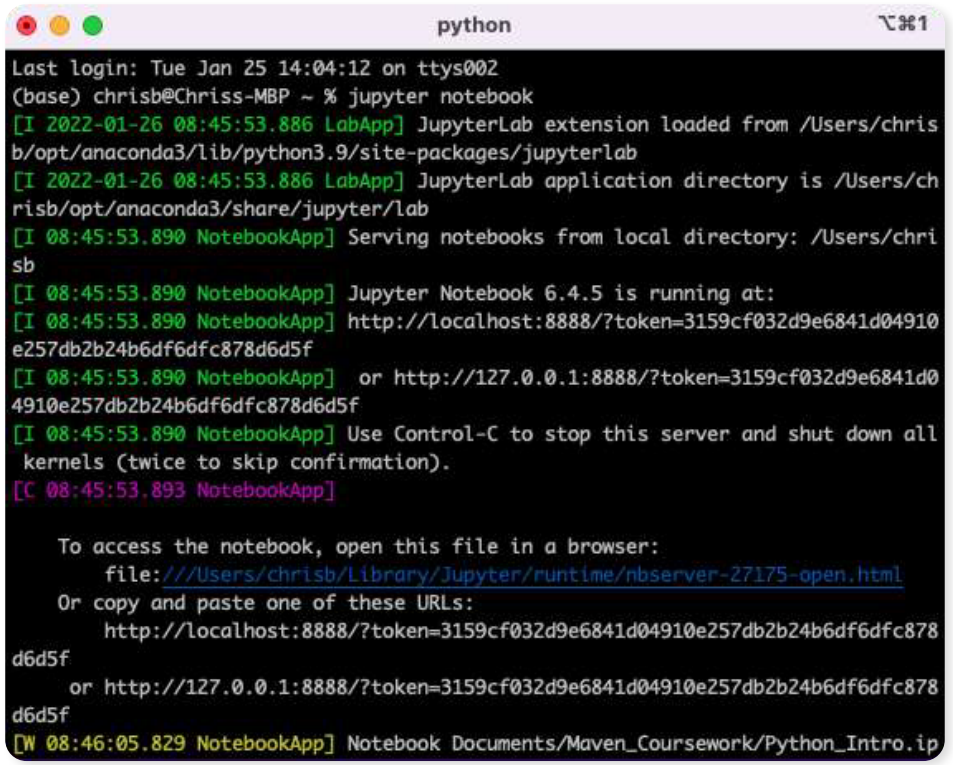




***NOTE:*** *You can rename your notebook by clicking on the title at the top of the screen*

# THE NOTEBOOK SERVER

**NOTE:** When you launch a Jupyter notebook, a terminal window may pop up as well; this is called a **notebook server**, and it powers the notebook interface

- Installing Anaconda
- Launching Jupyter
- Google Colab




If you close the server window, **your notebooks will not run!**

Depending on your OS, and method of launching Jupyter, one may not open. As long as you can run your notebooks, don't worry!

# ALTERNATIVE: GOOGLE COLAB

Installing Anaconda

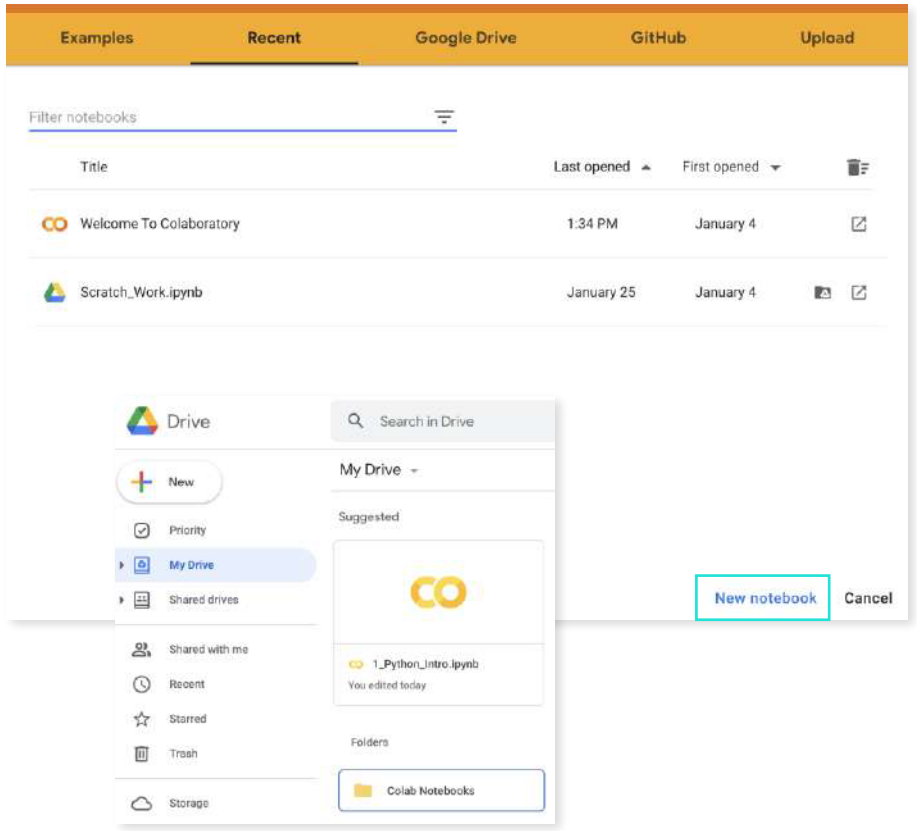
Launching Jupyter

Google Colab

**Google Colab** is Google's cloud-based version of Jupyter Notebooks

## To create a Colab notebook:

1. Log in to a Gmail account
2. Go to **colab.research.google.com**
3. Click "**new notebook**"




Colab is very similar to Jupyter Notebooks *(they even share the same file extension)*; the main difference is that you are connecting to **Google Drive** rather than your machine, so files will be stored in Google's cloud

# INTRO TO PLOTLY & DASH

# INTRO TO PLOTLY & DASH

In this section we'll introduce the **Plotly** & **Dash** libraries, and cover the steps & key components for creating a basic Dash application with interactive Plotly visuals

## TOPICS WE'LL COVER:

- Why Interactivity?
- The Plotly Ecosystem
- Dash Applications
- Layouts & Interactivity
- Callback Functions
- Application Run Options
- Interactive Plotly Visuals

## GOALS FOR THIS SECTION:

- Recognize the value of interactive charts & graphs
- Understand the anatomy of a Dash application, and the essential steps for creating one
- Learn to add interactive elements and process them with callback functions to update the app
- Create your first interactive visual with Plotly & Dash

# WHY INTERACTIVITY?



Adding **interactivity** to charts lets your stakeholders explore the data and conduct their own analysis, freeing up your time to tackle more problems

**EXAMPLE** | *Visualizing store transactions over time **(static)***

*"Can you build me a line chart showing transactions by store over time?"*

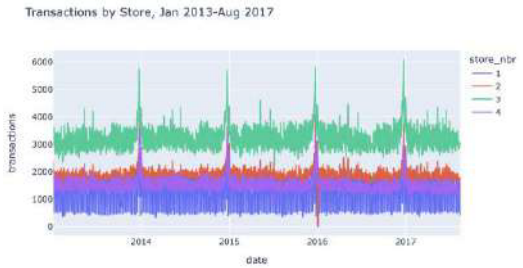
*"Great, now can you make another chart but zoom in for December 2013?"*

*"Thank you! What is the exact value for store 3 on Dec 3? I was also curious about 2015…"*

*"Sure thing!"*

*"Yeah, here you go."*

*"Uh..."*




# WHY INTERACTIVITY?

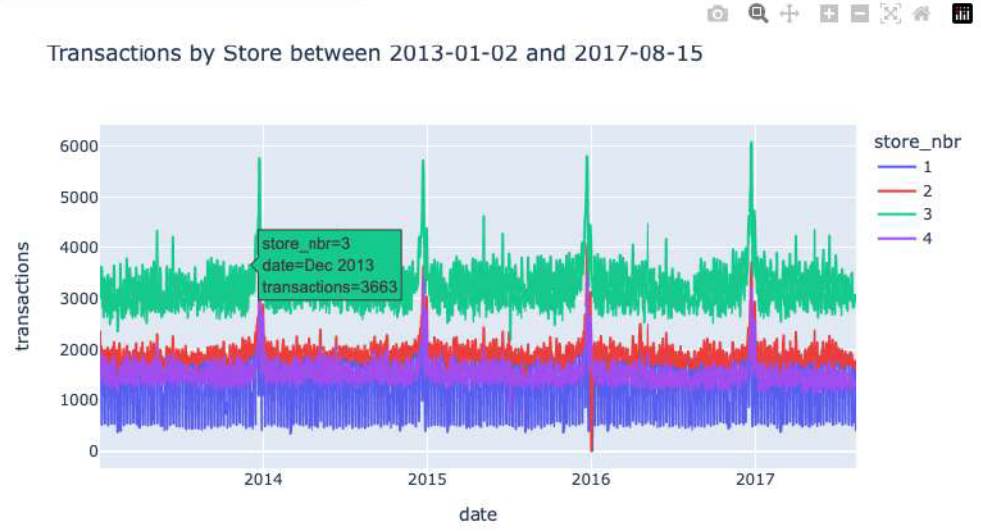


Adding **interactivity** to charts lets your stakeholders explore the data and conduct their own analysis, freeing up your time to tackle more problems

**EXAMPLE** | *Visualizing store transactions over time **(interactive)***

*"Can you build me a line chart showing transactions by store over time?"*

*"Sure thing, I set up an interactive chart that allows you to zoom in on selected date periods and hover over each point for more detail!"*




Adding **interactive elements** like dropdowns to filter data or tooltips to display metrics of interest can greatly improve the value of a single chart to its stakeholders

# MEET PLOTLY & DASH



plotly | Dash

**Plotly** is an open-source JavaScript based library that focuses on creating *interactive* visualizations with the Python, R, or Julia programming languages

**Dash** is Plotly's sister library that focuses on building *dashboards* with Plotly visuals and deploying them as interactive web applications

# INSTALLING PLOTLY & DASH



## 1) Install plotly and dash via conda

```
import sys
!conda install --yes --prefix {sys.prefix} plotly
```

```
!conda install --yes --prefix {sys.prefix} dash
```

## 2) Install jupyter-dash via pip

*This helps develop apps in Jupyter before Application Run*

```
!pip install --prefix {sys.prefix} jupyter-dash
```

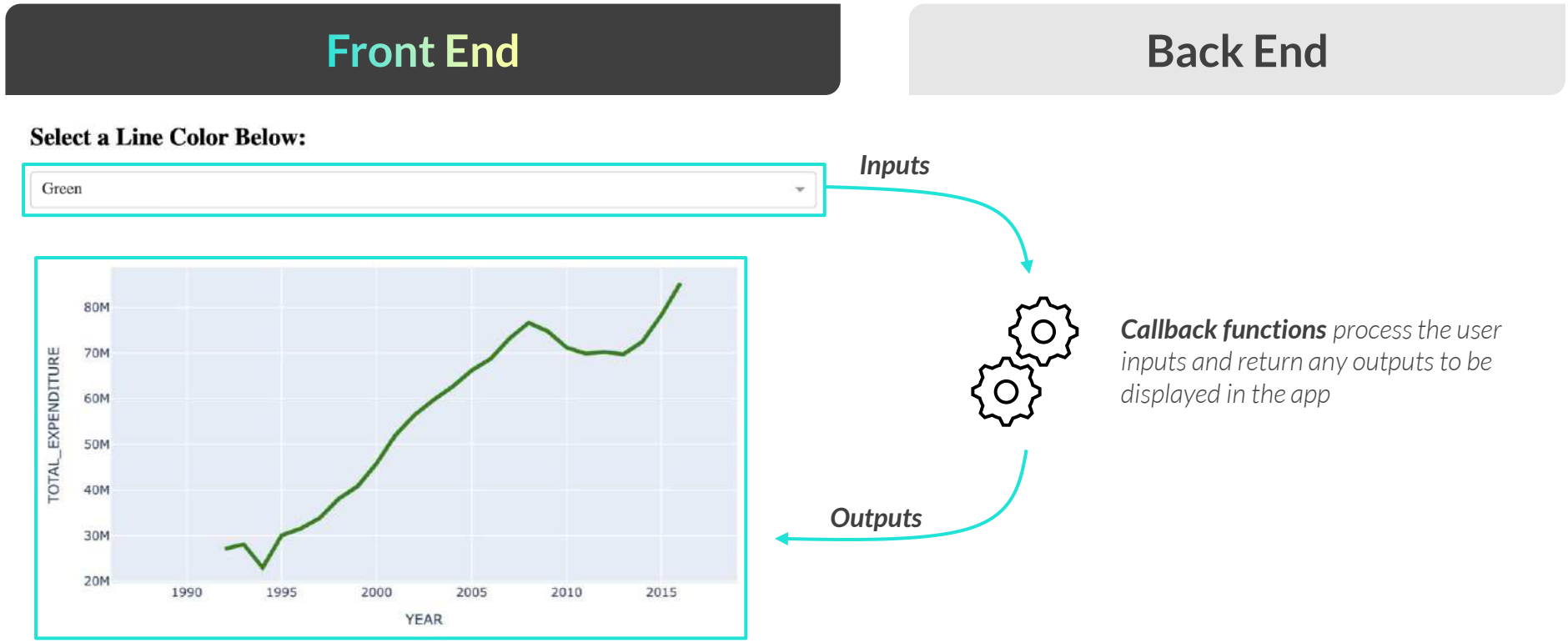
If you get a warning that "a new version of conda exists" during the installation, you can ignore it or run the code specified in the output

# THE ANATOMY OF A DASH APPLICATION



**Dash applications** are web apps with two main components:

- The **front end** displays interactive elements for user input (and any visuals affected by them)
- The **back end** processes the inputs and returns the outputs to be displayed in the visuals



# CREATING A DASH APPLICATION



You can **create a dash application** by following these steps:

1. Import the necessary libraries
2. Create the Dash application
3. Set up the HTML layout
4. Add the callback functions
5. Run the application

*At this point you just have an **empty app***

*This is the **front end** of the web application*

*This is the **back end** of the web application*

*This **runs** the app!*

```
from dash import Dash, dcc, html
from dash.dependencies import Input, Output

app = Dash(__name__)

app.layout = html.Div([
    html.H2(), # HTML Header
    dcc.Dropdown(), # An Interactive Dropdown Menu
    dcc.Graph() # A Chart that changes based on Dropdown Menu Value
])

@app.callback(Output(), Input()) # Ties dropdown to chart
def interactive_chart(input): # Create a function accepts dropdown value as argument
    return output # Usually a Plotly chart

app.run_server(debug=True)
```

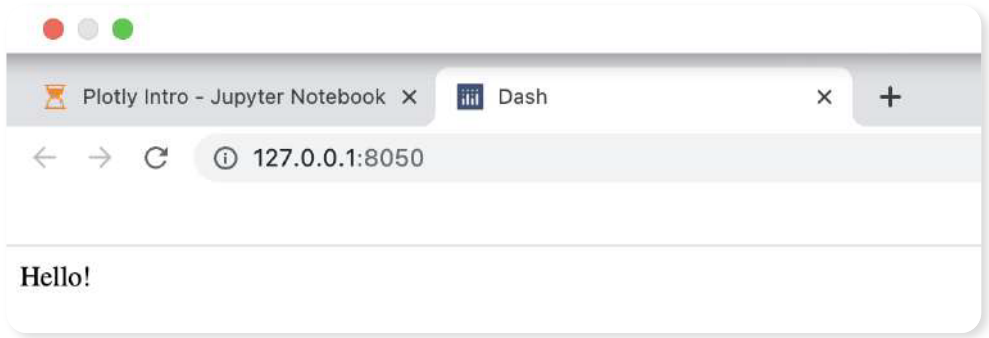
# THE WORLD'S SIMPLEST DASH APP



This is all you need to create the **world's simplest dash app**:

```
from dash import Dash, html

app = Dash(__name__)

app.layout = html.Div("Hello!")

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

```
Dash is running on http://127.0.0.1:8050/
```




1. *Import the Dash function and html module*
2. *Create the application with the Dash function*
3. *Add a single HTML element (div) with the text "Hello!"*
4. *Run the application on your local server*

Dash applications **do not require callback functions** to run, but they are necessary for adding interactivity based on user inputs

**Congrats**, you just became a Python web developer!

# THE WORLD'S SIMPLEST DASH APP



You can use JupyterDash to **run the app in-notebook** as you're designing it

```
from dash import html
from jupyter_dash import JupyterDash

app = JupyterDash(__name__)

app.layout = html.Div("Hello!")


if __name__ == "__main__":
    app.run_server(mode="inline")
```

Hello!

1. *Import the JupyterDash function instead of Dash*
2. *Create the application with the **JupyterDash** function*
3. *Add a single HTML element (div) with the text "Hello!"*
4. *Run the application **inside the Jupyter Notebook**, below the code cell (mode="inline")*

**PRO TIP:** Use JupyterDash to quickly and easily to iterate on your app inside the notebook – you can switch to Dash for Application Run once it's finished *(more later!)*

# IMPORTING LIBRARIES



To build dashboards using Plotly & Dash, you may need to **import these libraries**:

```
from dash import Dash, html, dcc
from dash.dependencies import Output, Input
from jupyter_dash import JupyterDash
import plotly.express as px
import pandas as pd
```

*Functions for creating the app and adding elements*

*Back-end functions for adding interactivity*

*Function for creating & running the app in Jupyter Notebooks*

*Library for data prep & ETL*

*Library for adding Plotly visuals*

Importing specific functions and modules from dash instead of the full library helps when writing the code, as you don't need to reference the library when calling each function

# CREATING THE APP



## The Dash(__name__) function **creates a Dash application**

- This is typically assigned to a variable called “app”

```
import dash

app = dash.Dash(__name__)

app
```

```
<dash.dash.Dash at 0x7feeeb9a8c40>
```

*Note that importing the entire dash library requires calling it before the Dash(__name__) function*

*This is a **Dash application** object*

**__name__** is a special variable used to only run a Dash app if the script is being run directly (not being imported as a module)

# HTML LAYOUTS



Dash uses **HTML layouts** for designing the front-end of the application

- Use the **html** module to specify the visual components and assign it to **app.layout**

```
from jupyter_dash import JupyterDash
from dash import html

app = JupyterDash(__name__)

app.layout = html.Div("Hello!")

if __name__ == "__main__":
    app.run_server(mode="inline")
```

*The **.Div()** method lets you add html sections to store different components*

Hello!

Familiarity with the **HTML** and **CSS** web design languages can be a big help for making beautiful applications – we will conduct a brief crash course on the basics later!

# INTERACTIVE ELEMENTS



Dash has many **interactive elements** that can be added to apps for user input

- These are provided by the Dash Core Components module (dcc)

| Interactive Element | Description |
|---|---|
| Dropdown | Single or multi-select dropdown list of pre-defined options |
| Slider | Slider for selecting a single value from a pre-defined list |
| RangeSlider | Slider for selecting a range of values from a pre-defined list |
| Checklist | Multi-select checkboxes from a pre-defined list |
| RadioItems | Single-select radio buttons from a pre-defined list |
| DatePickerSingle | Dropdown calendar to select a single date |
| DatePickerRange | Dropdown calendar to select a data range |
| Tabs | Tabs for navigating to different views of the app |
| Graphs | Container for Plotly figures |

These interactive elements are useless until they are processed in the back-end with **callback functions**

# INTERACTIVE ELEMENTS



Dash has many **interactive elements** that can be added to apps for user input

- These are provided by the Dash Core Components module (dcc)

**EXAMPLE** | *Adding a dropdown color-picker*

```
app = JupyterDash(__name__)

app.layout = html.Div([
    "Pick a Color!",
    dcc.Dropdown(
        options=["Red", "Green", "Blue"],
        id="color-input",
        value="Red"
    ),
])

if __name__ == "__main__":
    app.run_server(mode="inline")
```


→ This **dcc.Dropdown()** component has three properties:

1. **options**: contains the contents of the dropdown
   - *This is a list of strings that populate the dropdown menu*
   - *The option selected by the user gets passed through the value*
   - *It is possible to have a user facing label that differs from the value processed in the backend(more later!)*
2. **id**: the identifier for the value passed through
   - *This helps tie input values to outputs in the app (like charts!)*
3. **value**: the option selected
   - *This helps set a default "starting" value if needed*
   - *This property changes whenever a user selects a new option*

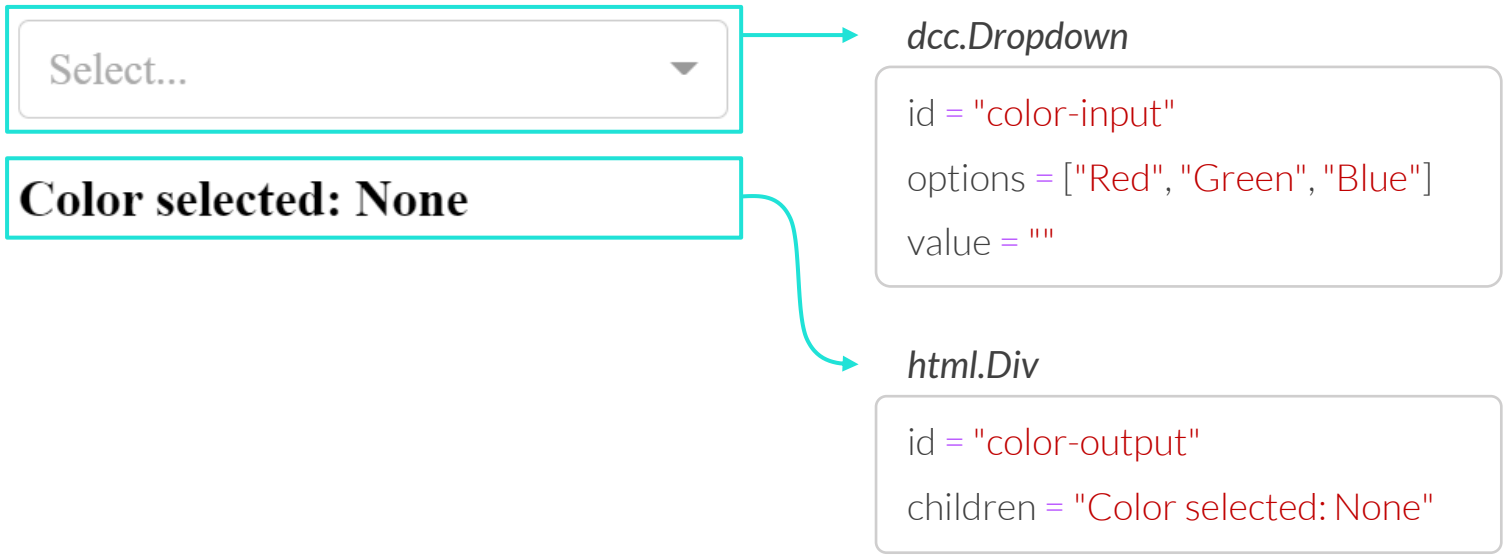
# CALLBACK FUNCTIONS



**Callback functions** process user inputs and update the app accordingly

- They are triggered by a change to a property of an HTML component *(input)*
- They then change the property of another HTML component *(output)*

**EXAMPLE** | *Adding a dropdown color-picker*



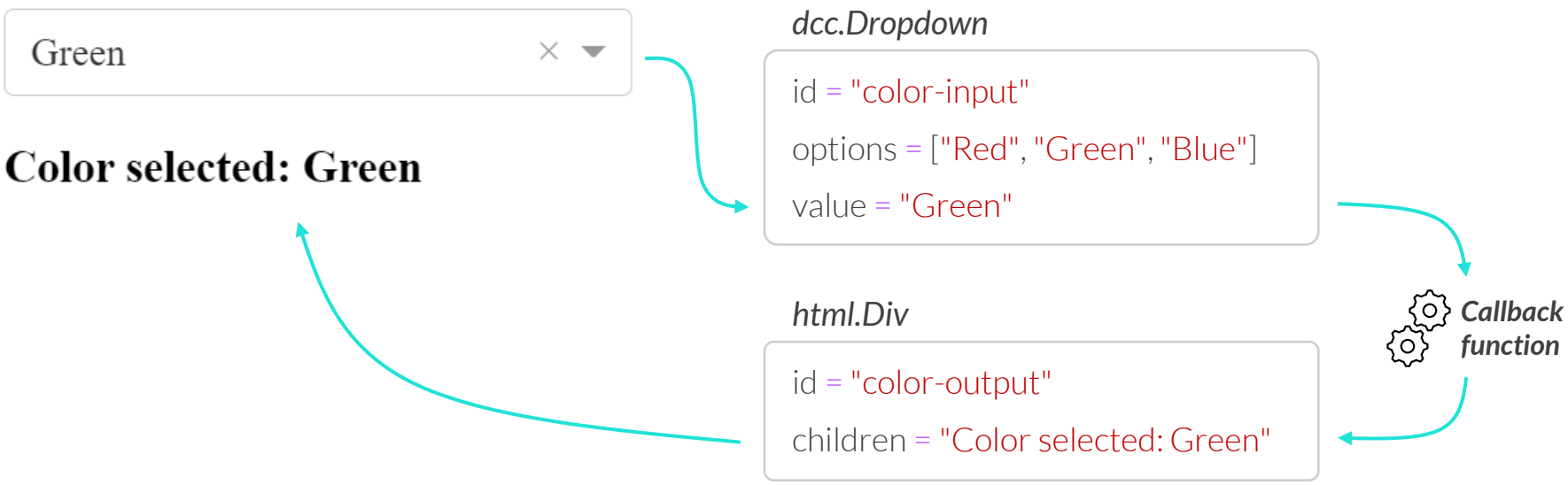
# CALLBACK FUNCTIONS



**Callback functions** process user inputs and update the app accordingly

- They are triggered by a change to a property of an HTML component *(input)*
- They then change the property of another HTML component *(output)*

**EXAMPLE** | *Adding a dropdown color-picker*



# CALLBACK FUNCTIONS



Callback functions are defined by using the **@app.callback** decorator and have at least two arguments *(Output & Input)*, followed by the function itself

```
@app.callback(
    Output(component_id, component_property),
    Input(component_id, component_property))
```

*The Input & Output arguments of the callback decorator*

*(Output goes first!)*

*The id of the html component that triggers (input) or gets modified by (output) the callback function*

*The property of the html component that is passed into (input) or gets modified by (output) the callback function*

***Examples:***

- *"children" (for text)*
- *"value" (for interactive elements)*
- *"figure" (for charts)*

# CALLBACK FUNCTIONS



Callback functions are defined by using the **@app.callback** decorator and have at least two arguments *(Output & Input)*, followed by the function itself

```
@app.callback(
      Output(component_id, component_property),
      Input(component_id, component_property)
)
def function_name(variable):
      #function steps
      return f"Output: {variable}"
```

*Defines the callback function and assigns the value from the property of the input component to a variable*

*New value for the property of the output component*

# CALLBACK FUNCTIONS



Callback functions are defined by using the **@app.callback** decorator and have at least two arguments *(Output & Input)*, followed by the function itself

**EXAMPLE** | *Adding a dropdown color-picker*

```
app.layout = html.Div(
    [dcc.Dropdown(
            options=["Red", "Green", "Blue"],
            id="color-input",
        ),
    html.Div(id="color-output"),
])

@app.callback(Output("color-output", "children"), Input("color-input", "value"))
def update_output_div(color):
    return f"Color Selected: {color}"
```

*Note that the "value" and "children" properties aren't specified when creating the components*

*This callback function is triggered when the "value" property of the "color-input" component changes, which updates the "children" property of the "color-output" component*

*The function must immediately follow the @app.callback decorator*

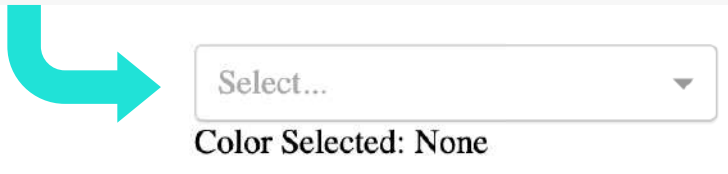
# PREVENTING UPDATES



Callback functions run as soon as the app launches by default, but you can add logic to **prevent updates**

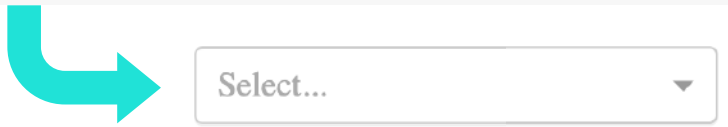
- This can help avoid errors when interactive elements are in an "empty state"

```
@app.callback(Output("color-output", "children"), Input("color-input", "value"))
def update_output_div(color):
    return f"Color Selected: {color}"
```




*Since the dropdown component has no "value" selected when launching the app, "None" gets passed into the callback function and output text*

```
@app.callback(Output("color-output", "children"), Input("color-input", "value"))
def update_output_div(color):
    if not color:
        raise PreventUpdate
    return f"Color Selected: {color}"
```




*This raises a **PreventUpdate** exception if the input is "None" which prevents the app from displaying any output text upon launch*

# APPLICATION RUN OPTIONS



There are several **Application Run options** you can use when running the app:

- **debug=True**: helps with troubleshooting during development *(i.e., better error messages)*
- **host/port**: specify the server address of the app – the default is: http://127.0.0.1:8050/
- **mode="inline"**: runs the app in-notebook when using JupyterDash *(not an option in Dash)*
- **height/width**: set the height or width of the app in pixels or a percentage

```
if __name__ == "__main__":
    app.run_server(debug=True, host="0.0.0.0", port=8051, mode="inline", height=800, width="80%" )
```

**PRO TIP:** The host/port options become more important when deploying your application; and setting width as a percentage is a great way to keep your app proportions consistent!

# **ASSIGNMENT**: SIMPLE DASH APP

**NEW MESSAGE**

*March 2024 ,3*

From: **Ernie Educator** *(Teacher & Friend)*

Subject: **Favor?**

Hey ol' buddy, ol' pal,

Did you catch the game yesterday? Crazy…

Anyways, last time we were together you mentioned you could build dashboards without paid software, and I was wondering if you could help me with a project for my blog. I am passionate about education and want to help others understand data on education funding – could you build a simple app that we can fill in together?

Thanks!

*section01_assignments.ipynb*

Reply Forward

## *Results Preview*

Select a State to Analyze:

Oregon

State Selected: Oregon

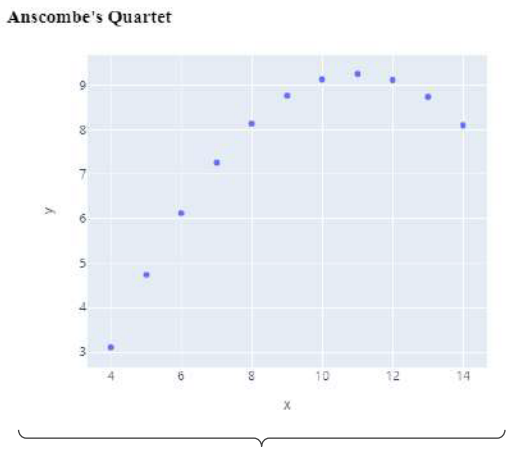
# INTERACTIVE PLOTLY VISUALS



You can add **interactive Plotly visuals** to Dash apps with these 3 steps:

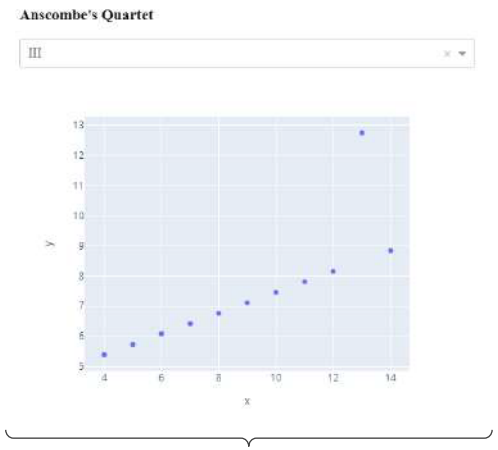
1. Create a "prototype" visual using Pandas & Plotly *without* interactivity
2. Identify the element that changes and define its options in the interactive component
3. Connect the interactive component to the visual using a callback function

**EXAMPLE** | *Visualizing the different datasets from Anscombe's Quartet*




*Create one "view" of the visual as if you had interacted with it*


*Use the values for the different "views" as the interactive options*




*Create the chart in the callback function to bring it all together*

# INTERACTIVE PLOTLY VISUALS



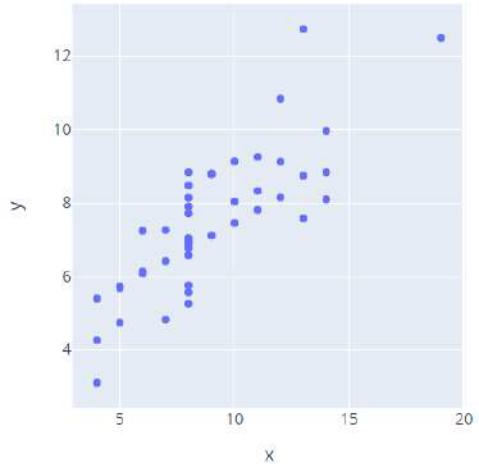
**STEP 1** *Create a "prototype" visual using Pandas & Plotly without interactivity*

```
df.head()
```

| | dataset | x | y |
|---|---|---|---|
| 0 | I | 10.0 | 8.04 |
| 1 | I | 8.0 | 6.95 |
| 2 | I | 13.0 | 7.58 |
| 3 | I | 9.0 | 8.81 |
| 4 | I | 11.0 | 8.33 |

*Anscombe's quartet is a group of four datasets with nearly identical regression lines but significantly different visual relationships*

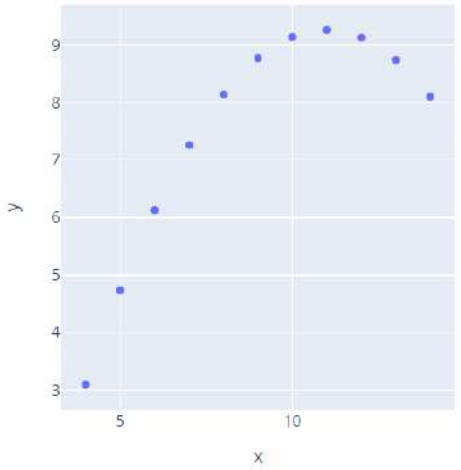
```
px.scatter(df, x="x", y="y")
```




*You can use Plotly Express and the* ***px.scatter*** *function to plot the "x" and "y" columns in the DataFrame*

*Note that this plots all four datasets in Anscombe's quartet*

```
px.scatter(df.loc[df["dataset"]=="II"], x="x", y="y")
```




*You can then filter the DataFrame to only plot one dataset at a time*

*This will be the element that the user can change to filter the chart!*

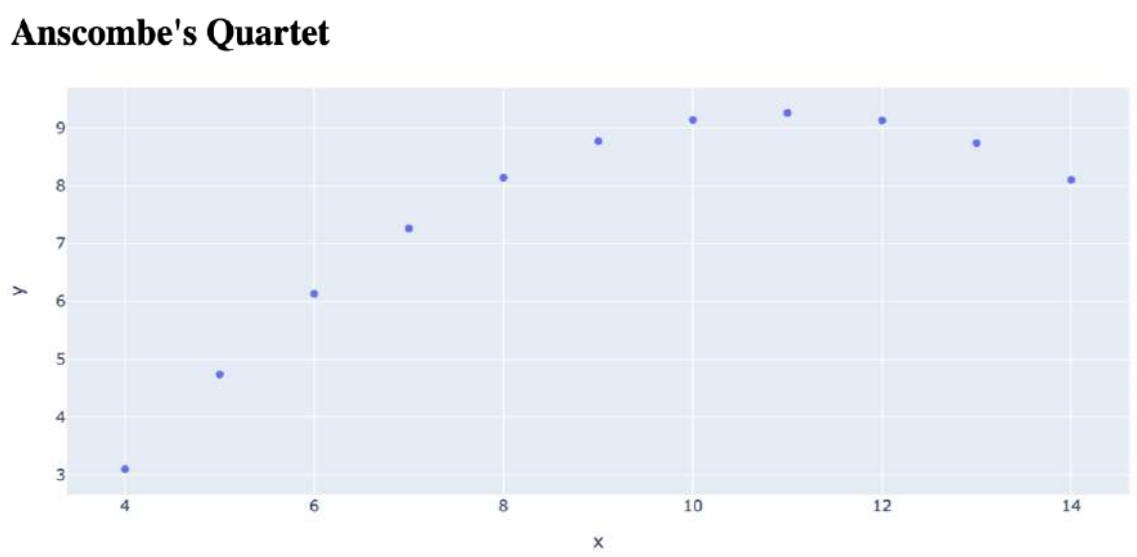
# INTERACTIVE PLOTLY VISUALS



**STEP 1** *Create a "prototype" visual using Pandas & Plotly without interactivity*

```
app.layout = html.Div([
    html.H3("Anscombe's Quartet"),
    dcc.Graph(
        id="visual",
        figure=px.scatter(df.loc[df["dataset"]=="II"], x="x", y="y")
    )
])
```

*You can add a Plotly visual to your app with the **figure** property of the **dcc.Graph** component*



# INTERACTIVE PLOTLY VISUALS



**STEP 2** *Identify the element that changes and define its options in the interactive component*

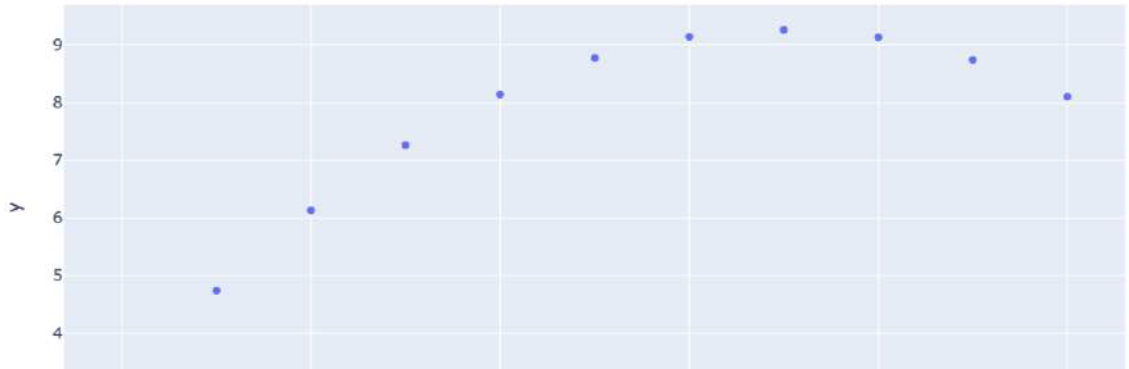
```
app.layout = html.Div([
    html.H3("Anscombe's Quartet"),
    dcc.Dropdown(id="dropdown", options=df["dataset"].unique()),
    dcc.Graph(
        id="visual",
        figure = px.scatter(
            df.query(f"dataset == 'II'"),
            x="x",
            y="y"
        ))
])
```

*You can use the **.unique()** method to grab all possible values in a column to populate your dropdown*

*This will let the user select from the possible values to filter the visual by*

**Anscombe's Quartet**

Select...




*Note that the dropdown and the visual are not connected yet!*

# INTERACTIVE PLOTLY VISUALS



**STEP 3** *Connect the interactive component to the visual using a callback function*

```
app.layout = html.Div([
    html.H3("Anscombe's Quartet"),
    dcc.Dropdown(id="dropdown", options = list(df["dataset"].unique())),
    dcc.Graph(id="visual")
])

@app.callback(Output("visual", "figure"), Input("dropdown", "value"))
def interactive_visual(selection):
    fig=px.scatter(
        df.query(f"dataset == '{selection}'"),
        x="x",
        y="y"
    )
    return fig
```

*This uses the value in the dropdown to filter the DataFrame to be plotted, then updates the "figure" property of the graph to show the updated visual*

*Note that the plot is only created in the callback function and doesn't need to be declared inside the dcc.Graph component*

# **ASSIGNMENT**: A MORE REALISTIC DASH APP

1 **NEW MESSAGE**

*March 2024 ,3*

From: **Ernie Educator** *(Teacher & Friend)*

Subject: **Embedding a Chart?**

Hey,

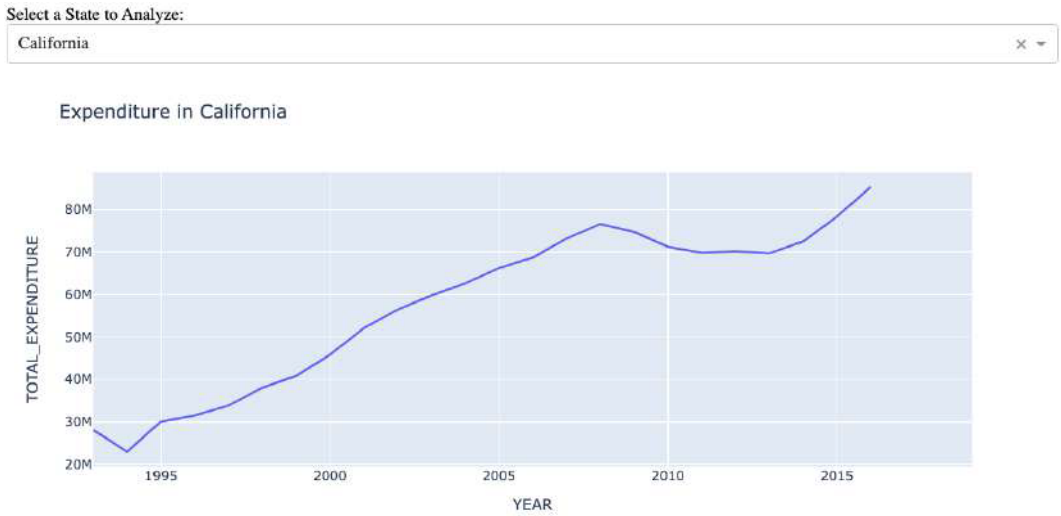
Cool application! I have a bit of Python knowledge, so I went ahead and prototyped a line chart that I'd like embedded in the application based on the state selected to analyze. Can you update the application? You can remove the html text output.

Thanks!

*section01_assignments.ipynb*

Reply Forward

## *Results Preview*



# KEY TAKEAWAYS

- Plotly & Dash are Python libraries for creating **interactive visuals**
  - *Plotly lets you build charts and Dash lets you interact with them by deploying them as web applications*
- Dash applications have two main components: a **front-end** & **back-end**
  - *The front-end is the HTML layout that displays any text, interactive elements, and visuals*
  - *The back-end ties the interactive elements and visuals together by using callback functions*
- **Callback functions** look for user "inputs" to update "outputs" accordingly
  - *Each input and output is associated with a component id and property*
- **Pandas** is key for manipulating the data you want to visualize
  - *Plotly charts are built using Pandas DataFrames, and their interactivity often relies on sorting & filtering them*

# PLOTLY CHARTS

# PLOTLY CHARTS

In this section we'll dive into the **Plotly** library and use it to build & customize several chart types, including line charts, bar charts, pie charts, scatterplots, histograms, and maps

## TOPICS WE'LL COVER:

- Plotly Figures
- Basic Charts
- Chart Formatting
- Map Visuals

## GOALS FOR THIS SECTION:

- Understand the difference between the Plotly Graph Objects and Plotly Express plotting methods
- Create basic charts using Plotly Express functions
- Customize Plotly Express charts using update methods
- Create map-based visuals using geographic data

# PLOTLY FIGURES

**Plotly figures** are created by drawing "traces" on top of a blank canvas

There are two methods for creating them:



## Plotly Graph Objects

*Charts are created by defining a figure object, and traces are customized using figure methods*

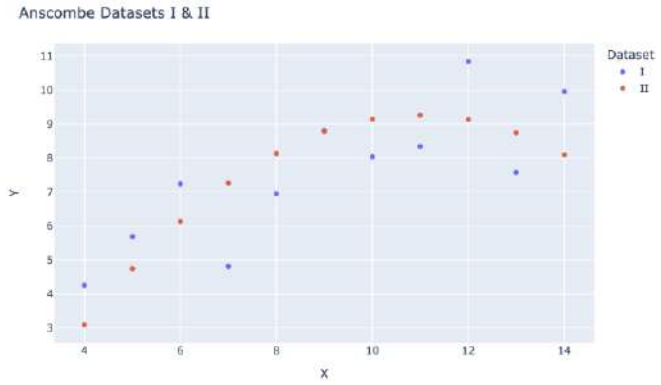
```
fig = go.Figure()

fig.add_scatter(
    x=anscombe.loc[anscombe["Dataset"]=="I", "x"],
    y=anscombe.loc[anscombe["Dataset"]=="I", "y"],
    mode="markers",
    name="I"
)

fig.add_scatter(
    x=anscombe.loc[anscombe["Dataset"]=="II", "x"],
    y=anscombe.loc[anscombe["Dataset"]=="II", "y"],
    mode="markers",
    name="II"
)

fig.layout.title = "Anscombe Datasets I & II"
fig.layout.legend.title = "Dataset"
fig.layout.xaxis.title = "X"
fig.layout.yaxis.title = "Y"

fig.show()
```

## Plotly Express

*Charts are created with Plotly Express functions, with most customization options built in*

```
px.scatter(
    anscombe.query("Dataset in ['I', 'II']"),
    x="x",
    y="y",
    color="Dataset",
    title="Ancsombe Datasets I & II"
)
```

*We'll focus on **Plotly Express**, as it's easier to work with and provides most of the options*



# BASIC CHARTS

Plotly Figures

Basic Charts

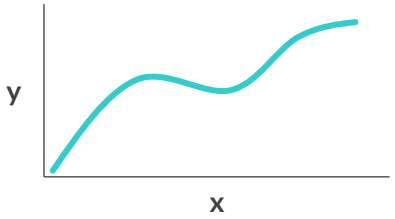
Chart Formatting

Map Visuals

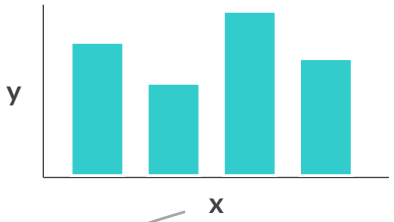
You can create these **basic charts** with Plotly Express by using these functions

- You simply need to select a DataFrame as the first argument and specify the DataFrame columns to plot for the rest of the arguments *(just be mindful of data types!)*

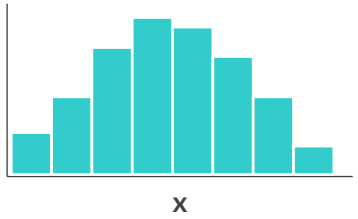
px.**line**(df, **x**="", **y**="")



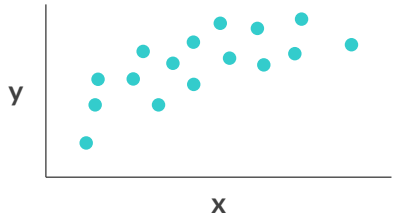

px.**bar**(df, **x**="", **y**="")



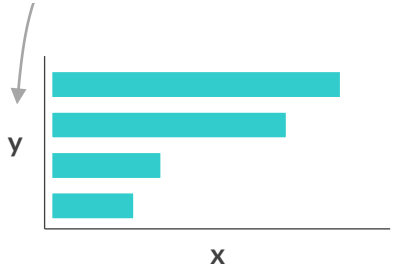

px.**histogram**(df, **x**="")



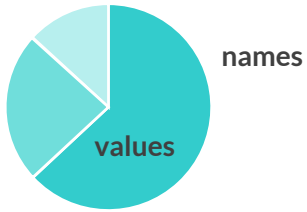

px.**scatter**(df, **x**="", **y**="")




*Just swap the columns!*




px.**pie**(df, **values**="", **names**="")



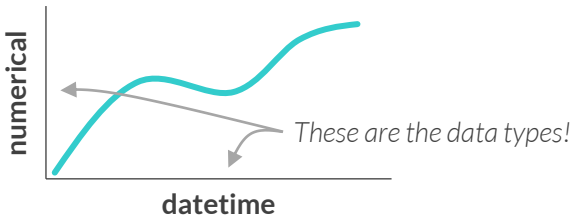
# BASIC CHARTS



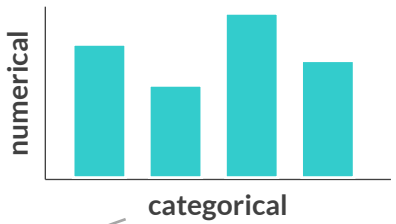
You can create these **basic charts** with Plotly Express by using these functions

- You simply need to select a DataFrame as the first argument and specify the DataFrame columns to plot for the rest of the arguments *(just be mindful of data types!)*

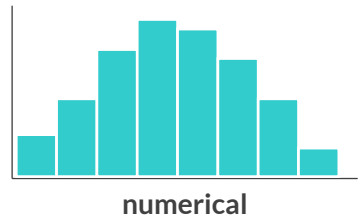
px.**line**(df, **x**="", **y**="")



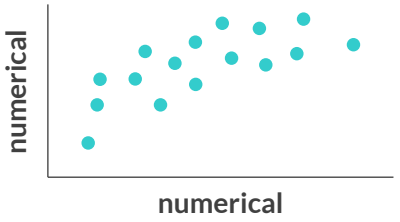

px.**bar**(df, **x**="", **y**="")

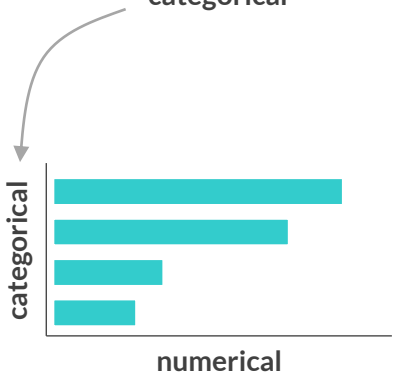



px.**histogram**(df, **x**="")

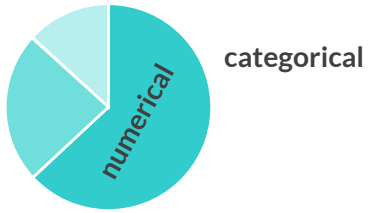



px.**scatter**(df, **x**="", **y**="")

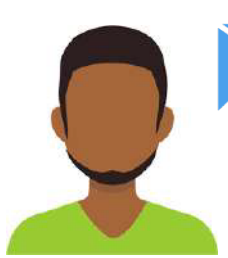



px.**pie**(df, **values**="", **names**="")



# **ASSIGNMENT:** LINE CHARTS



1 **NEW MESSAGE**

*March 2023 ,7*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **Info for Client**

Hey there,

I have a very "challenging" client who is asking me a million questions about European Ski Resorts.

Apparently, they went to Spain for a ski trip in the Pyrenees two decades ago and said it was way too crowded.

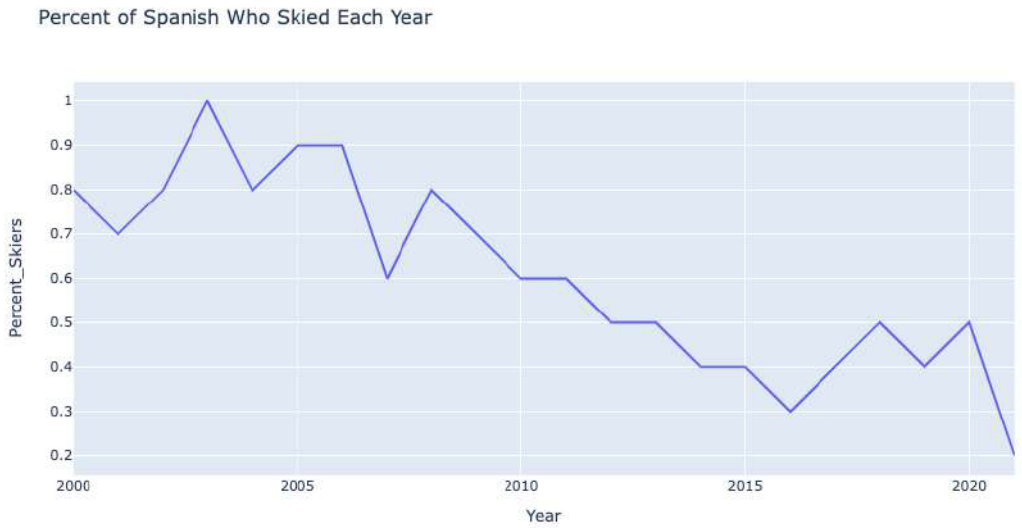
Can you look at the Spanish ski data and see if skiing is still as popular in Spain as it was back then?

Thanks!

*section02_assignments.ipynb*

Reply Forward

## *Results Preview*



# MULTIPLE SERIES

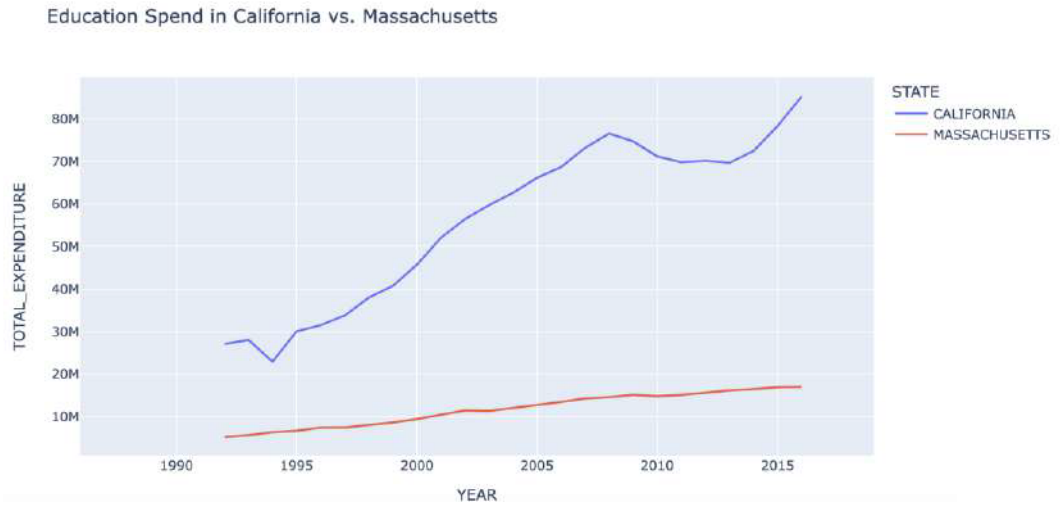
You can plot **multiple series** by using the "color" argument on most chart types



```
education.head()
```

| | STATE | YEAR | TOTAL_EXPENDITURE |
|---|---|---|---|
| 0 | ALABAMA | 1992 | 2653798.0 |
| 1 | ALASKA | 1992 | 972488.0 |
| 2 | ARIZONA | 1992 | 3401580.0 |
| 3 | ARKANSAS | 1992 | 1743022.0 |
| 4 | CALIFORNIA | 1992 | 27138832.0 |

```
px.line(
    education.query("STATE in ['CALIFORNIA', 'MASSACHUSETTS']"),
    x="YEAR",
    y="TOTAL_EXPENDITURE",
    color="STATE",
    title="Education Spend in California vs. Massachusetts"
)
```



# MULTIPLE SERIES

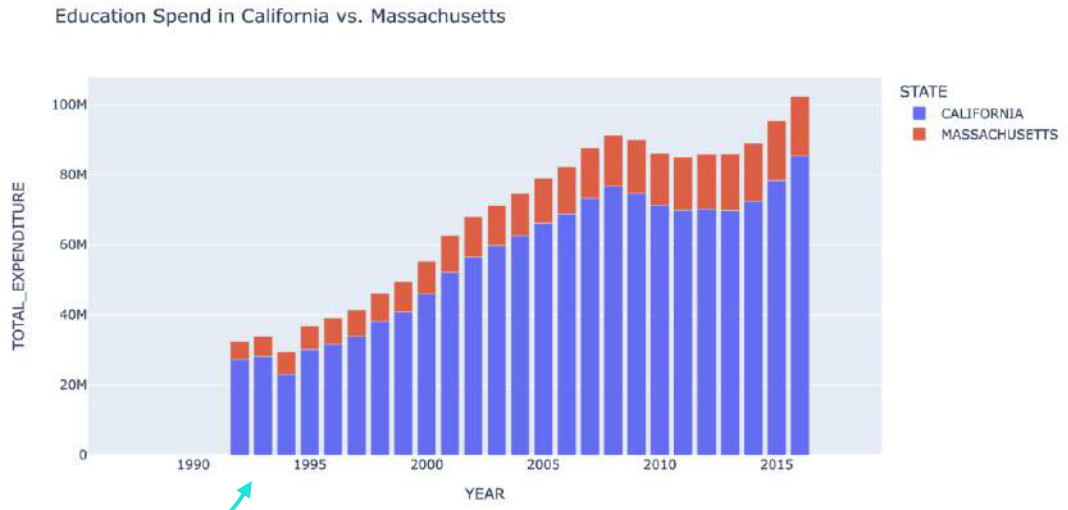
You can plot **multiple series** by using the "color" argument on most chart types

- Plotly Figures
- Basic Charts
- Chart Formatting
- Map Visuals

```
education.head()
```

| | STATE | YEAR | TOTAL_EXPENDITURE |
|---|---|---|---|
| 0 | ALABAMA | 1992 | 2653798.0 |
| 1 | ALASKA | 1992 | 972488.0 |
| 2 | ARIZONA | 1992 | 3401580.0 |
| 3 | ARKANSAS | 1992 | 1743022.0 |
| 4 | CALIFORNIA | 1992 | 27138832.0 |

```
px.bar(
    education.query("STATE in ['CALIFORNIA', 'MASSACHUSETTS']"),
    x="YEAR",
    y="TOTAL_EXPENDITURE",
    color="STATE",
    title="Education Spend in California vs. Massachusetts"
)
```




*Bars are stacked by default!*

# PRO TIP: GROUPED BAR CHARTS

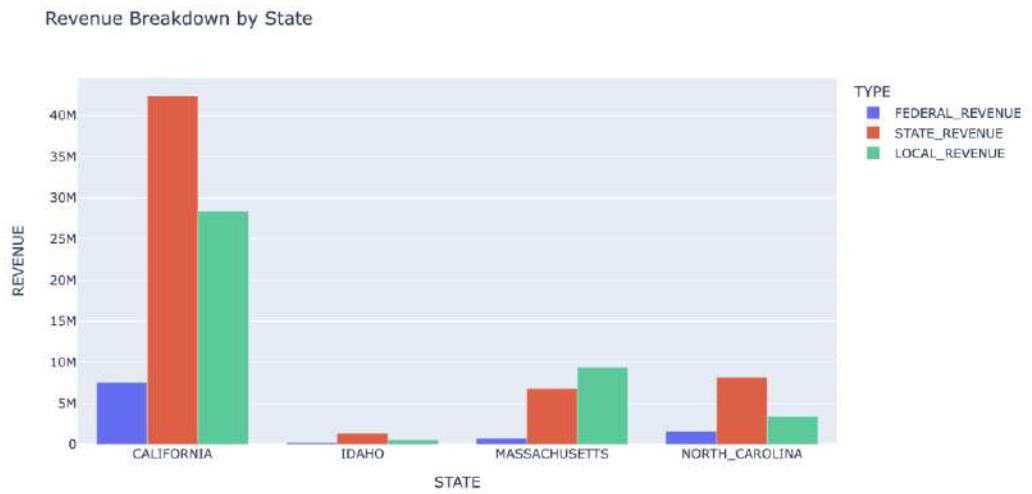
- Plotly Figures
- Basic Charts
- Chart Formatting
- Map Visuals

You can turn a stacked bar chart into a **grouped bar chart** with barmode="group"

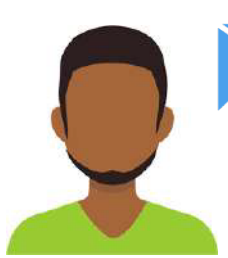
```
revenue
```

| | STATE | TYPE | REVENUE |
|---|---|---|---|
| 4 | CALIFORNIA | FEDERAL_REVENUE | 7556365.0 |
| 12 | IDAHO | FEDERAL_REVENUE | 232449.0 |
| 21 | MASSACHUSETTS | FEDERAL_REVENUE | 778939.0 |
| 33 | NORTH_CAROLINA | FEDERAL_REVENUE | 1589570.0 |
| 57 | CALIFORNIA | STATE_REVENUE | 42360470.0 |
| 65 | IDAHO | STATE_REVENUE | 1381205.0 |
| 74 | MASSACHUSETTS | STATE_REVENUE | 6808436.0 |
| 86 | NORTH_CAROLINA | STATE_REVENUE | 8172685.0 |
| 110 | CALIFORNIA | LOCAL_REVENUE | 28331207.0 |
| 118 | IDAHO | LOCAL_REVENUE | 554313.0 |
| 127 | MASSACHUSETTS | LOCAL_REVENUE | 9397810.0 |
| 139 | NORTH_CAROLINA | LOCAL_REVENUE | 3384679.0 |

```
px.bar(
    revenue,
    x="STATE",
    y="REVENUE",
    color="TYPE",
    barmode="group",
    title="Revenue Breakdown by State"
)
```



# **ASSIGNMENT**: BAR CHARTS



**NEW MESSAGE**

*March 2024 ,9*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **National Lift Characteristics**

Hey there,

We get questions from customers about which countries not only have the most ski lifts (some like to do multi-country tours) but also what types of lifts are available. Some customers really like gondola lifts for whatever reason.

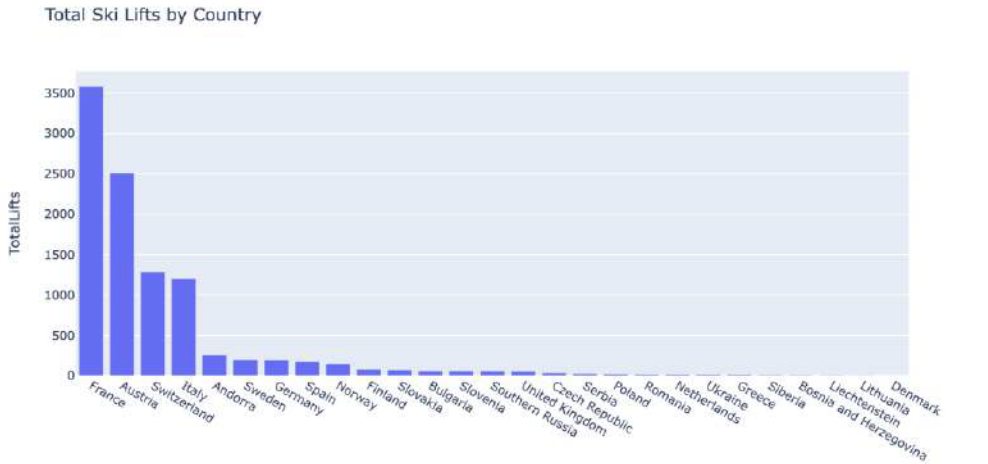
Could you produce a bar chart of lift count by country, and a second one that has these lifts broken out by lift category?

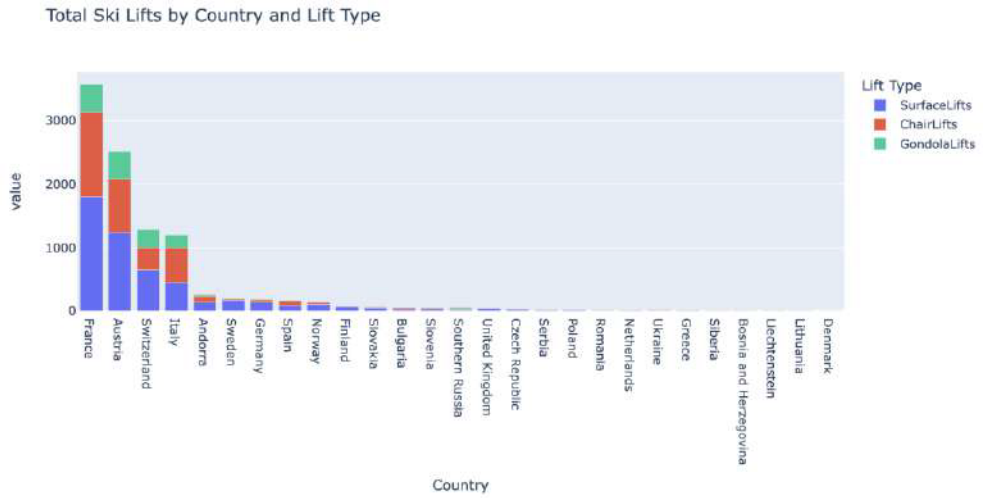
Thanks!

*section02_assignments.ipynb*

Reply Forward

## *Results Preview*



# PRO TIP: BUBBLE CHARTS

You can turn a scatterplot into a **bubble chart** by using the "size" argument



```
expenditure.head()
```

| | STATE | TOTAL_EXPENDITURE | AVG_MATH_8_SCORE | AVG_READING_8_SCORE |
|---|---|---|---|---|
| 1173 | ALABAMA | 7501799.0 | 267.0 | 259.0 |
| 1174 | ALASKA | 2968341.0 | 280.0 | 260.0 |
| 1175 | ARIZONA | 7902600.0 | 283.0 | 263.0 |
| 1176 | ARKANSAS | 5350543.0 | 275.0 | 259.0 |
| 1177 | CALIFORNIA | 78365958.0 | 275.0 | 259.0 |

```
px.scatter(
    expenditure,
    x="AVG_MATH_8_SCORE",
    y="AVG_READING_8_SCORE",
    size="TOTAL_EXPENDITURE",
    trendline="ols",
    title = "Math & Reading Scores by Spend"
)
```


**PRO TIP:** Use **trendline="ols"** to add the regression line

# PRO TIP: DONUT CHARTS



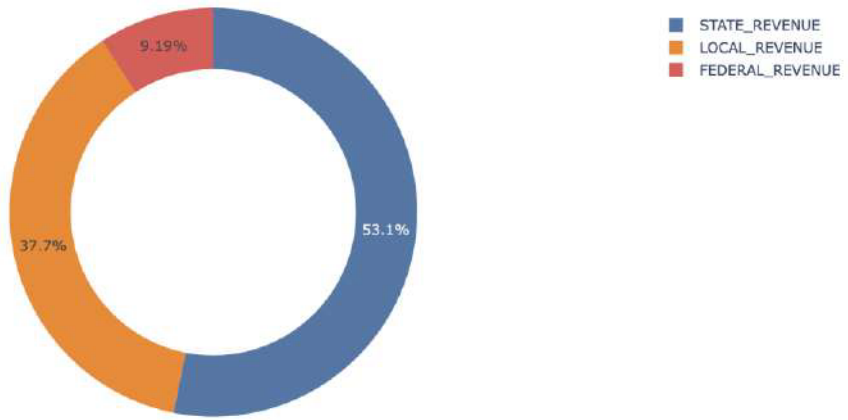
You can turn a pie into a **donut chart** by using the "hole" argument

- Other options include changing the sort order, and modifying the color sequence

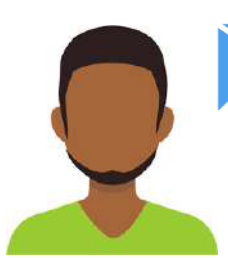
```
revenue
```

| | STATE | TYPE | REVENUE |
|---|---|---|---|
| 4 | CALIFORNIA | FEDERAL_REVENUE | 7556365.0 |
| 12 | IDAHO | FEDERAL_REVENUE | 232449.0 |
| 21 | MASSACHUSETTS | FEDERAL_REVENUE | 778939.0 |
| 33 | NORTH_CAROLINA | FEDERAL_REVENUE | 1589570.0 |
| 57 | CALIFORNIA | STATE_REVENUE | 42360470.0 |
| 65 | IDAHO | STATE_REVENUE | 1381205.0 |
| 74 | MASSACHUSETTS | STATE_REVENUE | 6808436.0 |
| 86 | NORTH_CAROLINA | STATE_REVENUE | 8172685.0 |
| 110 | CALIFORNIA | LOCAL_REVENUE | 28331207.0 |
| 118 | IDAHO | LOCAL_REVENUE | 554313.0 |
| 127 | MASSACHUSETTS | LOCAL_REVENUE | 9397810.0 |
| 139 | NORTH_CAROLINA | LOCAL_REVENUE | 3384679.0 |

```
px.pie(
    revenue,
    values="REVENUE",
    names="TYPE",
    hole=0.7,
    category_orders={"TYPE":["STATE_REVENUE","LOCAL_REVENUE","FEDERAL_REVENUE"]},
    color_discrete_sequence=px.colors.qualitative.T10,
    title="Revenue Breakdown"
)
```



# **ASSIGNMENT:** BUBBLE & DONUT CHARTS



1 **NEW MESSAGE**

*March 2024 ,9*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **RE: Info for Client**

Hi again,

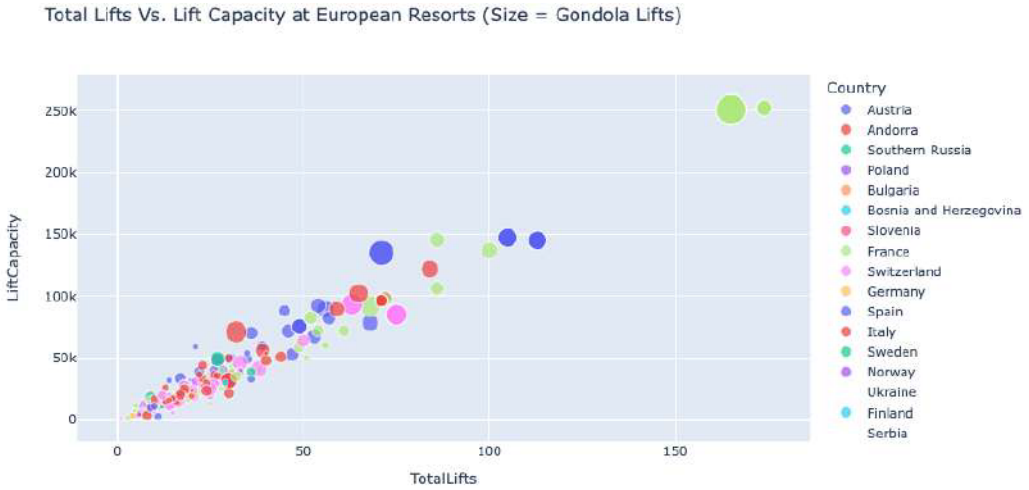
Thanks for getting me those charts, the client was really pleased! Somewhat related, this client hates waiting in line at lifts, and believes resorts with high lift capacity and lift numbers will have shorter waits. They also prefer Gondola lifts. Can you build a bubble chart that compares TotalLifts to LiftCapacity with the size of each marker as the number of Gondola Lifts?

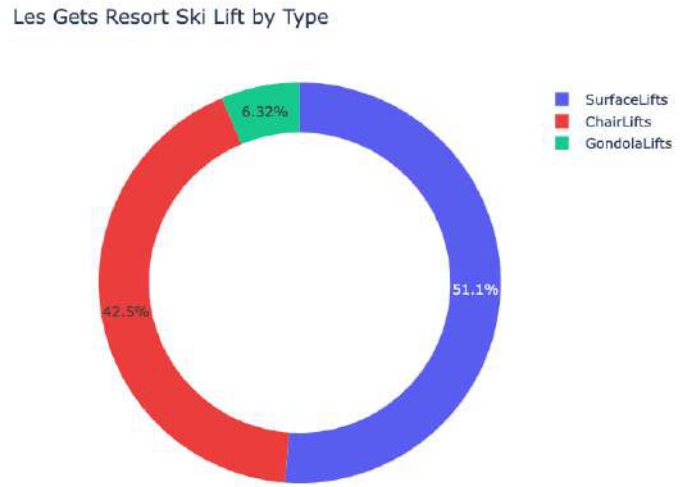
Then build a donut chart breaking down lift types for the resort with the highest lift capacity.

*section02_assignments.ipynb*

Reply Forward

## *Results Preview*



# HISTOGRAM OPTIONS

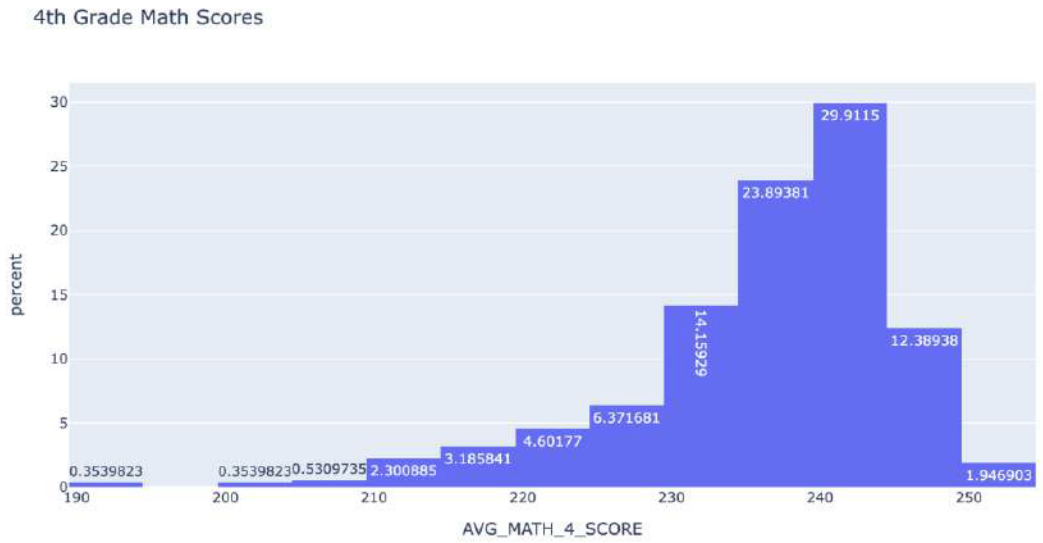
**Histogram options** include setting the number of bins, using relative frequencies, and adding data labels

- Plotly Figures
- Basic Charts
- Chart Formatting
- Map Visuals

```
grades.head()
```

| | STATE | YEAR | AVG_MATH_4_SCORE |
|---|---|---|---|
| 0 | ALABAMA | 1992 | 208.0 |
| 1 | ALASKA | 1992 | NaN |
| 2 | ARIZONA | 1992 | 215.0 |
| 3 | ARKANSAS | 1992 | 210.0 |
| 4 | CALIFORNIA | 1992 | 208.0 |

```
px.histogram(
    grades,
    x="AVG_MATH_4_SCORE",
    nbins=20,
    histnorm="percent",
    text_auto=True,
    title="4th Grade Math Scores"
)
```



# UPDATE METHODS

Plotly Figures

Basic Charts

Chart Formatting

Map Visuals

**Update methods** let you format Plotly Express charts using object-oriented commands, combining the best of both worlds!

- While Plotly Express does have formatting options, update methods are occasionally useful

| | | |
|---|---|---|
| **update_layout** | *Modifies chart elements like the title, legend, fonts, figure size, and more* | **px.line(…).update_layout(*legend_title*, …)** |
| **update_traces** | *Modifies plotted data like the line styles, colors, markers, and more* | **px.line(…).update_traces(*opacity*, …)** |
| **update_xaxes** | *Modifies the x-axis formatting, including the ticks, units, and text* | **px.line(…).update_xaxes(*nticks*, …)** |
| **update_yaxes** | *Modifies the y-axis formatting, including the ticks, units, and text* | **px.line(…).update_yaxes(*showgrid*, …)** |

*This can be any chart type*

*For a full list of arguments, visit:* ***https://plotly.com/python/reference/index/***

# UPDATE LAYOUT

Plotly Figures

Basic Charts

Chart Formatting

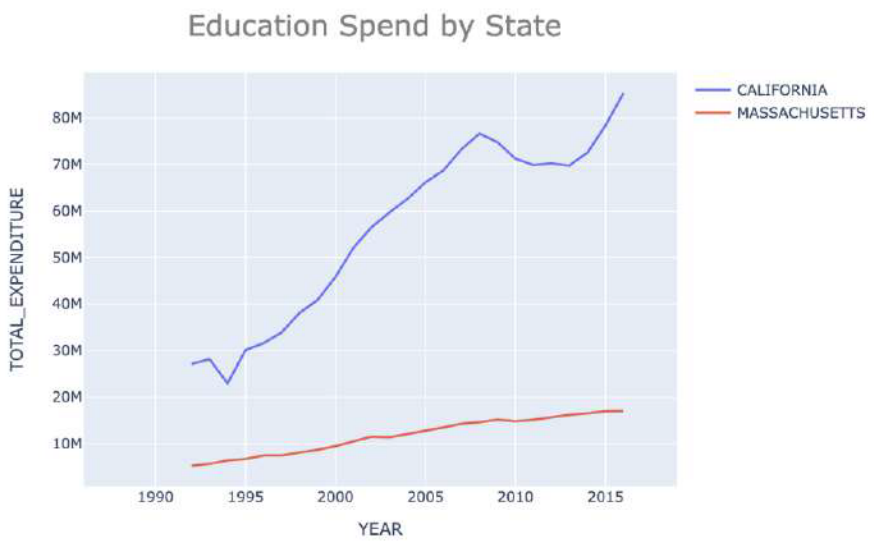
Map Visuals

The **.update_layout()** method lets you customize figure and plot elements

- This is commonly used to modify the title and legend formatting, as well as fonts and colors

```
px.line(
    education.query("STATE in ['CALIFORNIA', 'MASSACHUSETTS']"),
    x="YEAR",
    y="TOTAL_EXPENDITURE",
    color="STATE",
    title="Education Spend by State"
).update_layout(
    title_font = dict(
        color="grey",
        size=24
    ),
    title = {
        "x":.43,
        "y":.9,
        "xanchor":"center"
    },
    legend_title="",
    width=750
)
```

*This modifies the title's color, size, and position, removes the legend title, and changes the figure's width*



# UPDATE LAYOUT



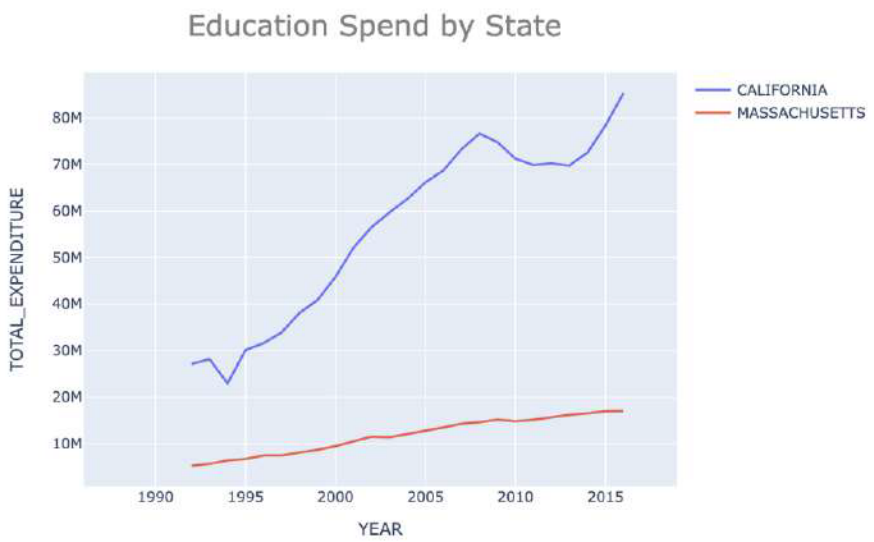
The **.update_layout()** method lets you customize figure and plot elements

- This is commonly used to modify the title and legend formatting, as well as fonts and colors

```
fig = px.line(
    education.query("STATE in ['CALIFORNIA', 'MASSACHUSETTS']"),
    x="YEAR",
    y="TOTAL_EXPENDITURE",
    color="STATE",
    title="Education Spend by State"
)
```

```
fig.update_layout(
    title_font = dict(
        color="grey",
        size=24
    ),
    title = {
        "x":.43,
        "y":.9,
        "xanchor":"center",
    },
    legend_title="",
    width=750
)
```

*By assigning the Plotly Express chart to a variable "fig", you can gradually apply chart formatting using update methods*



# UPDATE TRACES

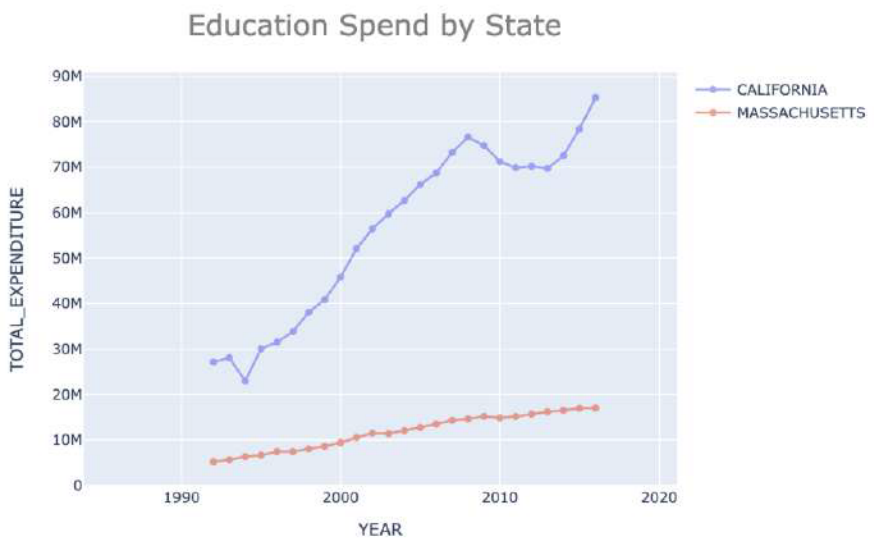


The **.update_traces()** method lets you modify the formatting of the plotted data

- This is commonly used to change the styling, colors, and opacity

```
fig.update_traces(
    mode="lines+markers",
    opacity=.6
)
```

*This adds markers to the line charts and changes their opacity*



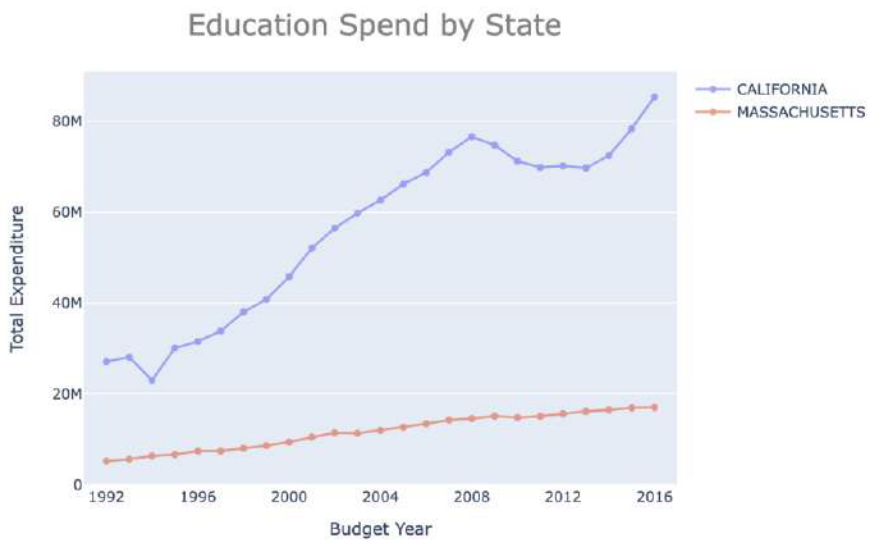
# UPDATE AXES

- Plotly Figures
- Basic Charts
- Chart Formatting
- Map Visuals

The **.update_xaxes()** & **.update_yaxes()** methods let you format each axis

- This is commonly used to customize the gridlines, ranges, and tick marks

```
fig.update_xaxes(
    title="Budget Year",
    showgrid=False,
    range=[1991,2017],
    dtick=4
)

fig.update_yaxes(
    title="Total Expenditure",
    nticks=5
)
```




*For the x-axis, this changes the title, removes the gridlines, modifies the range of years, and sets the ticks every 4 years*

*For the y-axis, tis changes the title, and sets 5 ticks in total*

# PRO TIP: TREATING DATES AS TEXT



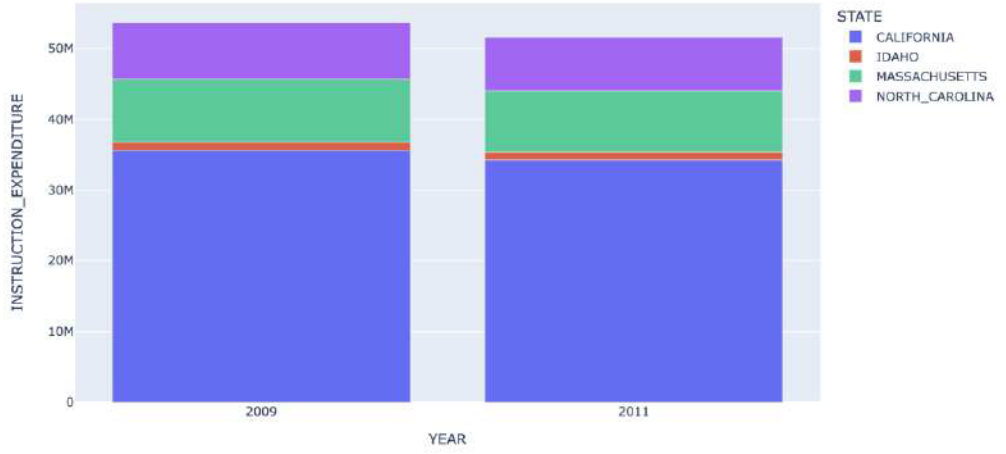
You can use **.update_xaxes(type="category")** to avoid gaps when using dates

```
education2years
```

| | YEAR | STATE | INSTRUCTION_EXPENDITURE |
|---|---|---|---|
| **871** | 2009 | CALIFORNIA | 35617964.0 |
| **879** | 2009 | IDAHO | 1157633.0 |
| **888** | 2009 | MASSACHUSETTS | 8885949.0 |
| **900** | 2009 | NORTH_CAROLINA | 7943541.0 |
| **973** | 2011 | CALIFORNIA | 34225248.0 |
| **981** | 2011 | IDAHO | 1104553.0 |
| **990** | 2011 | MASSACHUSETTS | 8685894.0 |
| **1002** | 2011 | NORTH_CAROLINA | 7566249.0 |

*Note that 2010 is missing!*

```
px.bar(
    education2years,
    x="YEAR",
    y="INSTRUCTION_EXPENDITURE",
    color="STATE"
).update_xaxes(type="category")
```

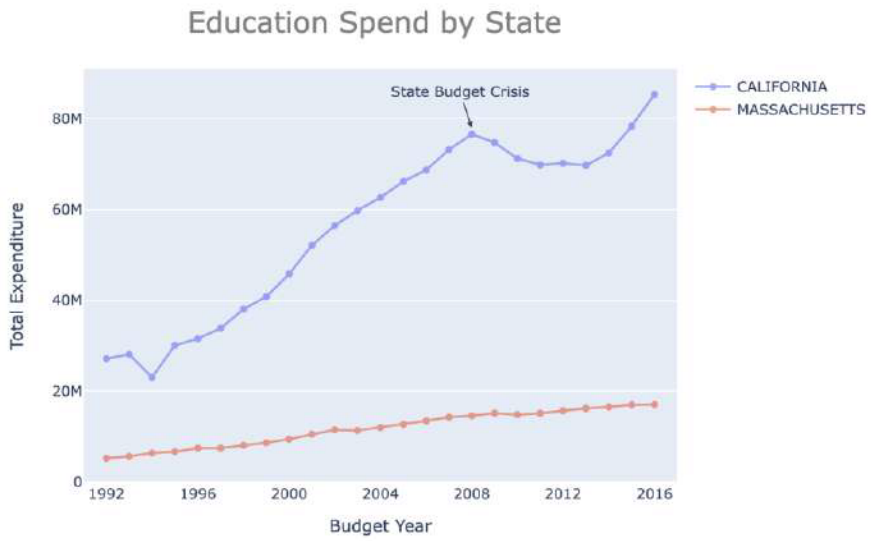


# ADDING ANNOTATIONS

- Plotly Figures
- Basic Charts
- Chart Formatting
- Map Visuals

The **.add_annotation()** method lets you call out key data points in charts

```
fig.add_annotation(
    text="State Budget Crisis",
    x=2008,
    y=78000000,
    arrowhead=2
)
```

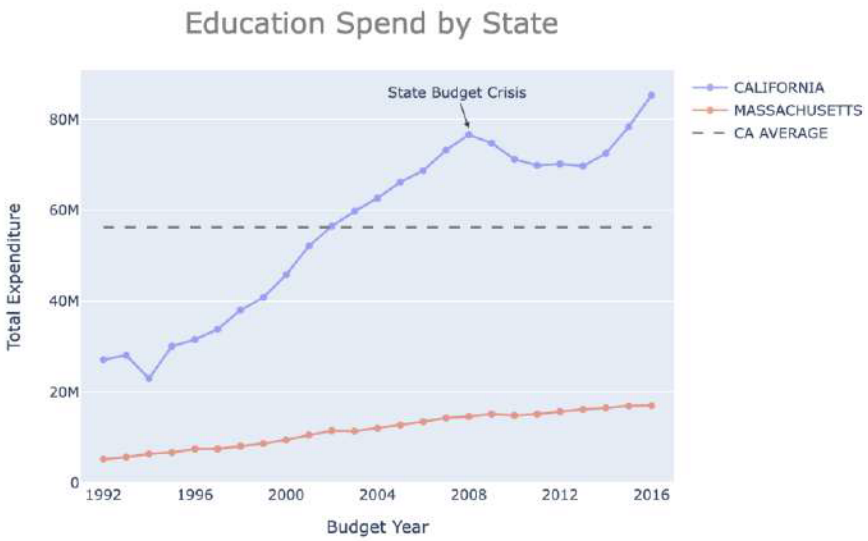


# ADDING TRACES

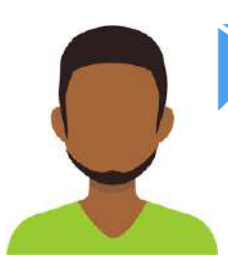
The **.add_trace()** method lets you add shapes *(like reference lines!)* to charts



```
import plotly.graph_objects as go

ca_mean = education.loc[
    education["STATE"]=="CALIFORNIA",
    "TOTAL_EXPENDITURE"
].mean()

reference_line = go.Scatter(
    x=(1992,2016),
    y=(ca_mean, ca_mean),
    mode="lines",
    line={"color":"gray", "dash":"dash"},
    name="CA AVERAGE"
)

fig.add_trace(
    reference_line
)
```



# **ASSIGNMENT**: CHART FORMATTING



1 **NEW MESSAGE**

*March 2024 ,12*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **Aesthetics**

Hi,

One of our clients has asked for some changes to the aesthetics of the bar chart we made earlier.

Additionally, they want to limit the view to the top 10 countries.

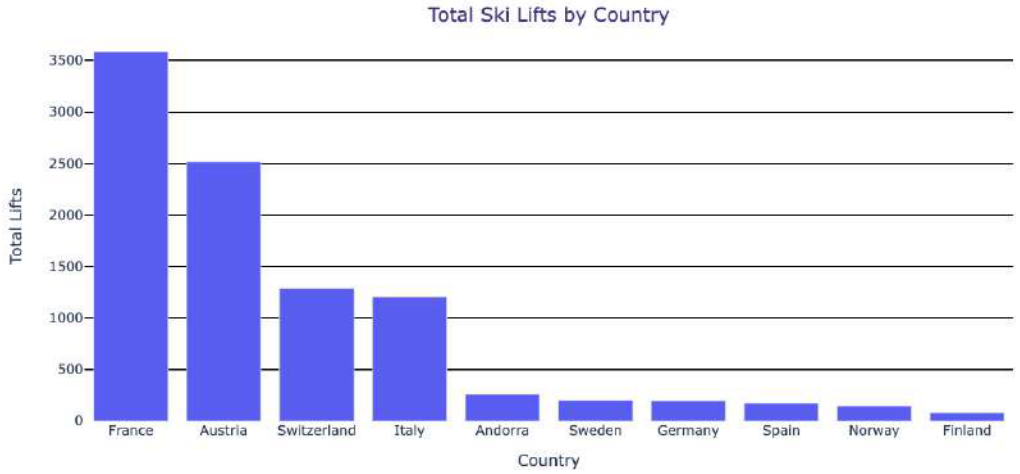
I've added details in the notebook.

Thanks!

*section02_assignments.ipynb*

Reply Forward

## *Results Preview*

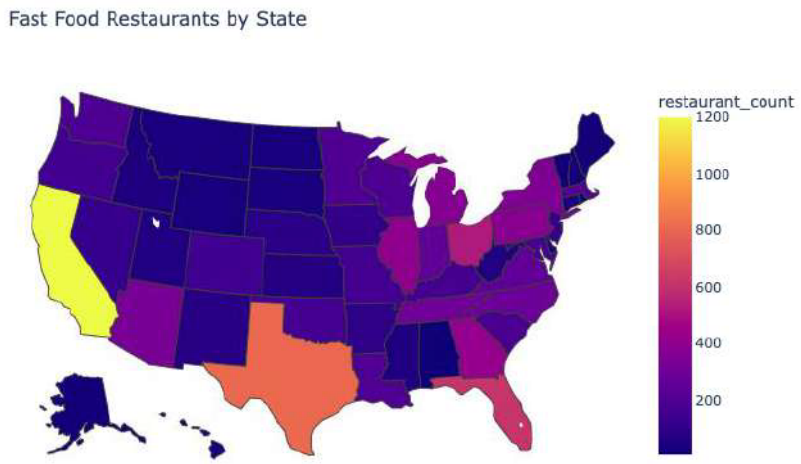


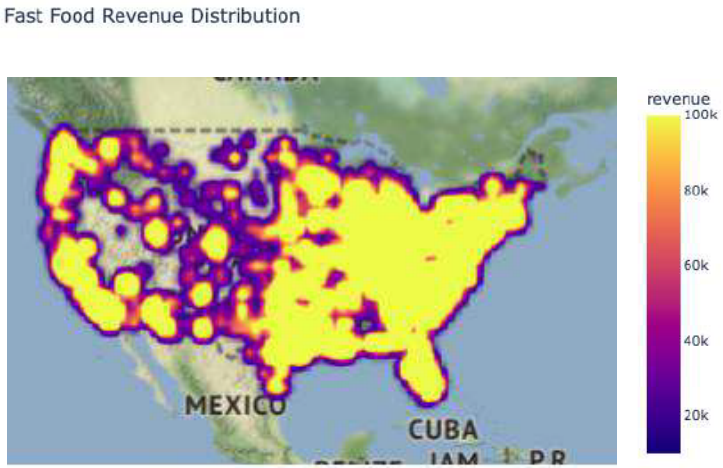
# MAP-BASED VISUALS

Plotly Figures
Basic Charts
Chart Formatting
Map Visuals

Plotly Express also has functions for **map-based visuals** using geographical data:

- px.**choropleth** uses geographic regions *(countries, states, etc.)*
- px.**scatter_mapbox** and px.**density_mapbox** use latitude & longitude pairs




**PRO TIP:** Plotly has built-in options for plotting countries and US states, but Googling examples for other regions is a great way to avoid creating a custom solution from scratch

# CHOROPLETH MAPS

Plotly Figures
Basic Charts
Chart Formatting
Map Visuals

**Choropleth maps** are created with the px.choropleth() function

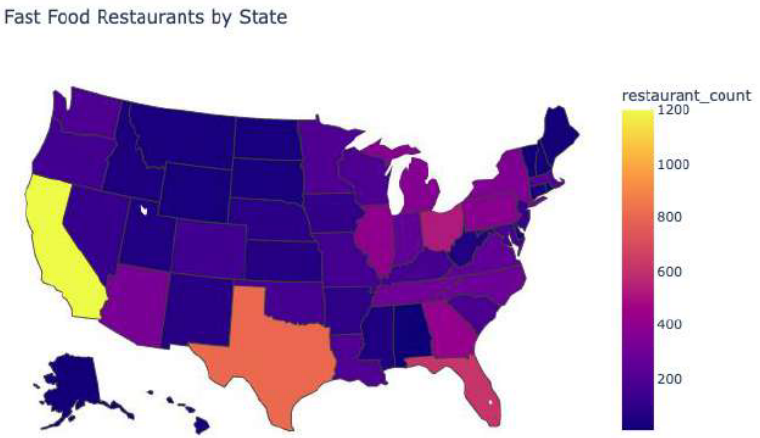
Like all Plotly Express charts, the first argument is the DataFrame, followed by:

- **locations**: the DataFrame column that contains the geographical data *(state, country, etc.)*
- **locationmode**: the type of geographical data used *("ISO-3", "USA-states", or "country names")*
- **color**: the numerical DataFrame column that will determine the shade of each region
- **scope**: the range of the map shown *(default is "world", others include "usa", "europe", "asia", etc.)*

```
fast_food_by_state.head()
```

| | province | restaurant_count |
|---|---|---|
| 0 | AK | 16 |
| 1 | AL | 6 |
| 2 | AR | 102 |
| 3 | AZ | 330 |
| 4 | CA | 1201 |

```
px.choropleth(
    fast_food_by_state,
    locations="province",
    color="restaurant_count",
    locationmode="USA-states",
    scope="usa",
    title="Fast Food Restaurants by State"
)
```




The **locations** & **location_mode** must match for the map to be created properly

# PRO TIP: MAPBOX MAPS

Plotly Figures

Basic Charts

Chart Formatting

Map Visuals

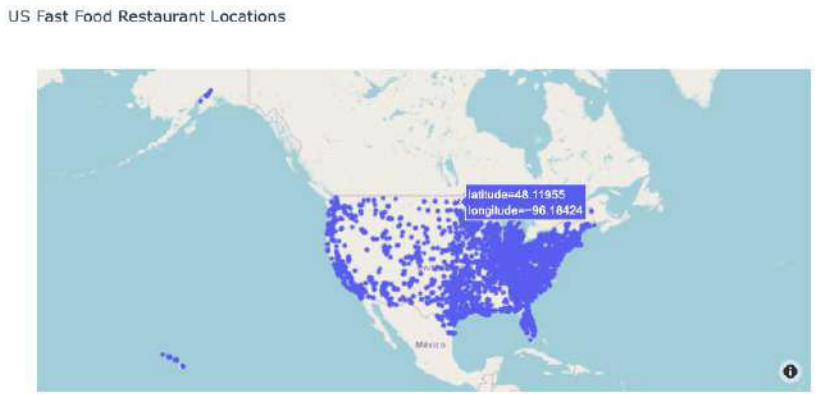
**Mapbox maps** use an open-source mapping API integrated into Plotly to create map-based visuals using latitude & longitude pairs

```
fast_food.head()
```

| | city | country | latitude | longitude | revenue |
|---|---|---|---|---|---|
| 0 | Thibodaux | US | 29.814697 | -90.814742 | 10974.865182 |
| 1 | Thibodaux | US | 29.814697 | -90.814742 | 82284.225538 |
| 2 | Pigeon Forge | US | 35.803788 | -83.580553 | 52336.439630 |
| 3 | Pigeon Forge | US | 35.782339 | -83.551408 | 20083.872548 |
| 4 | Morrow | US | 33.562738 | -84.321143 | 73965.018760 |

```
px.scatter_mapbox(
    fast_food,
    lat="latitude",
    lon="longitude",
    center={"lat": 44.5, "lon": -103.5},
    zoom=2,
    mapbox_style="open-street-map",
    title="US Fast Food Restaurant Locations"
)
```

*The **"center"** & **"zoom"** arguments let you set a default view for the map*



# **PRO TIP:** MAPBOX MAPS

- Plotly Figures
- Basic Charts
- Chart Formatting
- Map Visuals

**Mapbox maps** use an open-source mapping API integrated into Plotly to create map-based visuals using latitude & longitude pairs

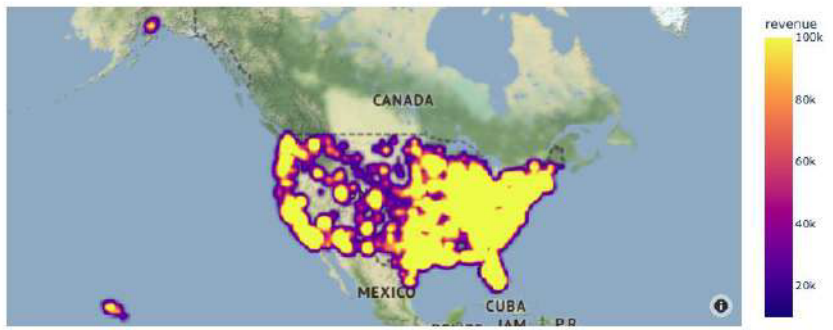
```
fast_food.head()
```

| | city | country | latitude | longitude | revenue |
|---|---|---|---|---|---|
| 0 | Thibodaux | US | 29.814697 | -90.814742 | 10974.865182 |
| 1 | Thibodaux | US | 29.814697 | -90.814742 | 82284.225538 |
| 2 | Pigeon Forge | US | 35.803788 | -83.580553 | 52336.439630 |
| 3 | Pigeon Forge | US | 35.782339 | -83.551408 | 20083.872548 |
| 4 | Morrow | US | 33.562738 | -84.321143 | 73965.018760 |

```
px.density_mapbox(
    fast_food,
    lat="latitude",
    lon="longitude",
    z="revenue",
    radius=fast_food["revenue"]/10000,
    center={"lat": 44.5, "lon": -103.5},
    zoom=2,
    mapbox_style="stamen-terrain",
    title="Fast Food Revenue Distribution"
)
```

*The **"z"** & **"radius"** arguments let you specify a numerical field for the heatmap*

**PRO TIP:** You can use the px.density_mapbox() function to create a heatmap style map



# **ASSIGNMENT:** MAPS

**NEW MESSAGE**

*March 2024 ,11*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **Lift Capacity**

Ok,

This is hopefully the last need for our client (and me, the lowly concierge).

Can we get a map-based view of lift capacity in each country?

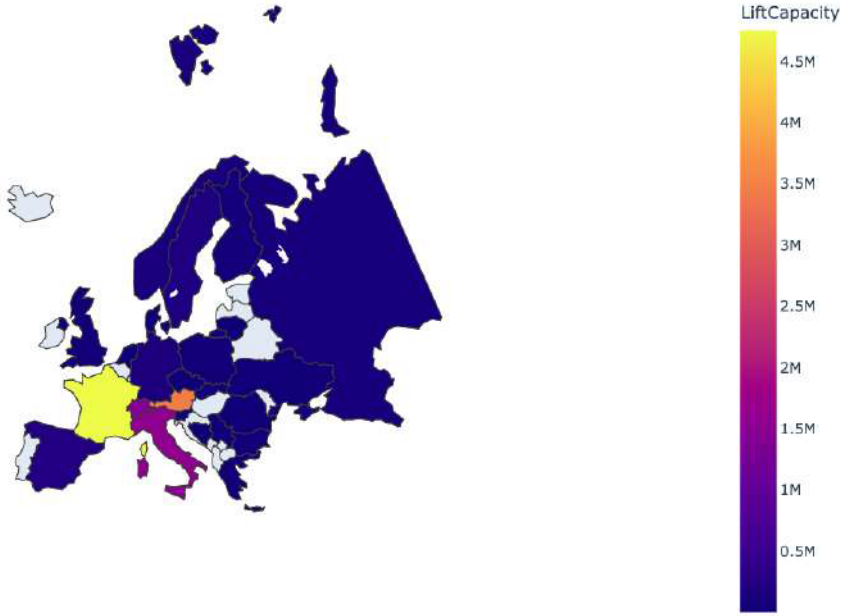
Should be easier to digest than a bar chart for this purpose.

Thanks!

*section02_assignments.ipynb*

Reply Forward

## *Results Preview*



# KEY TAKEAWAYS

Plotly has **two methods** for creating charts

- *Plotly Graph Objects is an object-oriented approach that offers extreme customization*
- *Plotly Express is an intuitive functional framework that lets you easily create & customize charts*

**Plotly Express** has functions for most basic chart types

- *By reading in a DataFrame you can create line charts, bar charts, pie charts, scatterplots, and histograms*

Use **update methods** to customize the layout, plot, and axes of your charts

- *These leverage the object-oriented approach to creating charts for combining the best of both worlds*

Geographical data can be plotted with **choropleth maps**

- *Plotly Express has built-in options for countries and US states, but solutions for other regions can be found online*

# INTERACTIVE ELEMENTS

# INTERACTIVE ELEMENTS

In this section we'll cover **interactive elements** from the Dash Core Components module, and use them to provide different data inputs for manipulating Plotly visuals in Dash apps

## TOPICS WE'LL COVER:

- Basic Interactivity
- Interactive Elements
- Multiple Input Callbacks
- Multiple Output Callbacks

## GOALS FOR THIS SECTION:

- Identify the various types of components used to interact with Plotly visuals in Dash and their use cases
- Learn to create & format interactive elements and process their inputs through callback functions
- Add multiple interactive elements to your Dash apps and write callback functions to handle multiple inputs

# BASIC INTERACTIVITY

Basic Interactivity

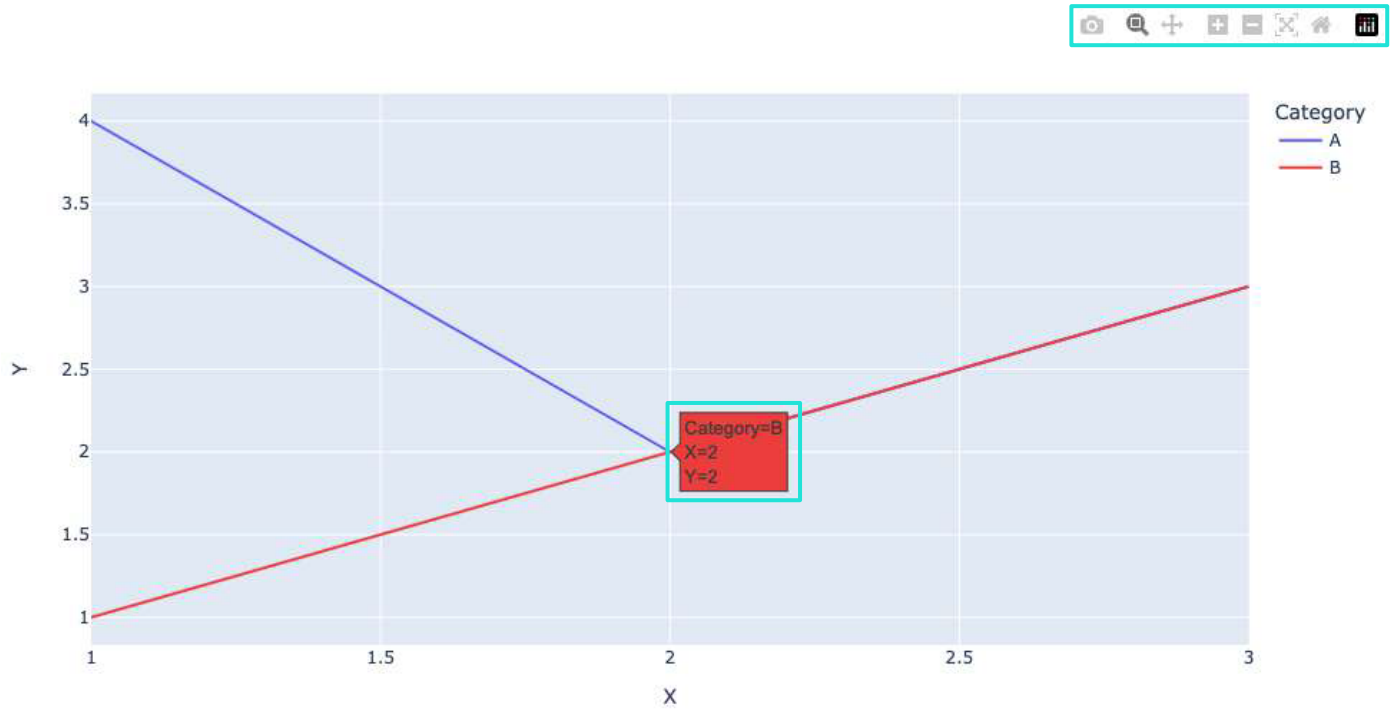
Interactive Elements

Multiple Input Callbacks

Multiple Output Callbacks

Plotly figures have some **basic interactivity** by default

- This includes tool tips with data labels, the ability to zoom in & out, and more!




**Hover Menu Options**:

- *Download Image*
- *Select Zoom Area*
- *Pan*
- *Zoom In*
- *Zoom Out*
- *Auto Scale*
- *Return to Default*

# INTERACTIVE ELEMENTS

- Basic Interactivity
- Interactive Elements
- Multiple Input Callbacks
- Multiple Output Callbacks

The Dash Core Components (dcc) module has several **interactive elements** that can be used in Dash apps to create dynamic dashboards with Plotly figures

| Component | Description | Logical Operators |
|---|---|---|
| dcc.Dropdown() | Dropdown list of options for the user to select (or multi-select) | ==, !=, **in**, **not in** |
| dcc.Checklist() | Checkboxes with options for the user to select or deselect | ==, !=, **in**, **not in** |
| dcc.RadioItems() | Radio buttons with options for the user to toggle between | ==, !=, **in**, **not in** |
| dcc.Slider() | Slider with a handle for the user to drag and select values with | ==, <, <=, >, >= |
| dcc.RangeSlider() | Slider with two handles for the user to drag and select ranges with | .between(value[0], value[1]) |
| dcc.DatePickerSingle() | Dropdown calendar for the user to select a date with | ==, <, <=, >, >= |
| dcc.DatePickerRange() | Dropdown calendar for the user to select a date range with | .between(start, end) |

*These are used in the callback functions to filter the visuals based on the user selections*

Other key considerations to discuss are **data types** and the **number of options available** to users

*For a full list of arguments, visit:* ***https://dash.plotly.com/dash-core-components***

# DROPDOWN MENUS

Basic Interactivity

Interactive Elements

Multiple Input Callbacks

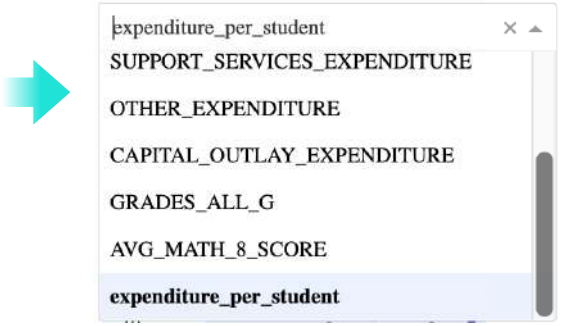
Multiple Output Callbacks

**Dropdown menus** provide a list of options for the user to select *(or multi-select)*

- dcc.Dropdown(id, options, value, multi)

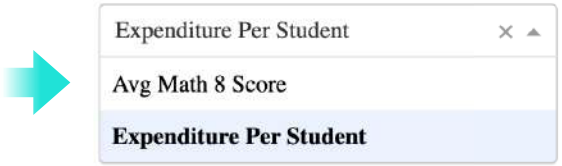
```
dcc.Dropdown(
        id="X Column Picker",
        options=list(education.select_dtypes(include='number').columns),
        value="expenditure_per_student",
)
```

*This sets the names for the columns with numerical data types as the options*




```
dcc.Dropdown(
        id="X Column Picker",
        options=[
            {'value': "AVG_MATH_8_SCORE", "label": "Avg Math 8 Score"},
            {"value": "expenditure_per_student", "label": "Expenditure Per Student"}
        ],
        value="expenditure_per_student",
)
```

*You can use **dictionaries** to show users labels in the dropdown that are different from the values passed through*



# DROPDOWN MENUS

- Basic Interactivity
- Interactive Elements
- Multiple Input Callbacks
- Multiple Output Callbacks

**EXAMPLE** | *Changing the dependent "x" variable in a scatterplot with a regression line*

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.Dropdown(
        id="X Column Picker",
        options=list(
            education.select_dtypes(include="number").columns
        ),
        value="expenditure_per_student"
    ),
    dcc.Graph(id="Graph")
])

@app.callback(
    Output("Graph", "figure"),
    Input("X Column Picker", "value")
)
def plot_rev_scatter(x):
    if not x:
        raise PreventUpdate
    fig = px.scatter(
        education,
        x=x,
        y="AVG_MATH_8_SCORE",
        trendline="ols",
        title=f"Relationship between {x} & AVG_MATH_8_SCORE"
    )
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```

*This lets the user select a numerical column in the dropdown, and sets "expenditure_per_student" as the default*

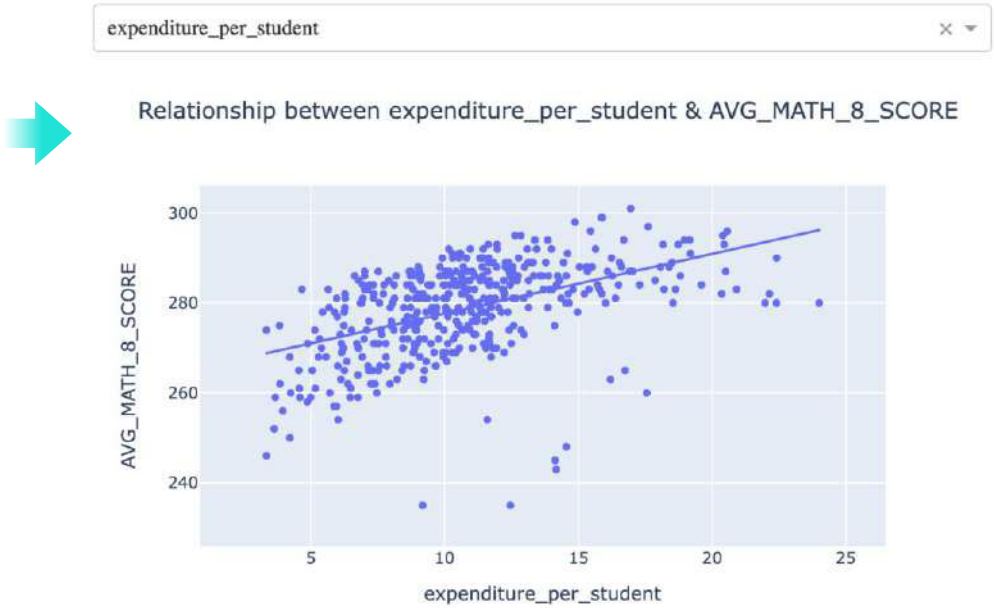
*The selection gets passed into the callback function and used as the "x" variable in the scatterplot, returning the updated chart*

# DROPDOWN MENUS

- Basic Interactivity
- Interactive Elements
- Multiple Input Callbacks
- Multiple Output Callbacks

**EXAMPLE** *Changing the dependent "x" variable in a scatterplot with a regression line*

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.Dropdown(
        id="X Column Picker",
        options=list(
            education.select_dtypes(include="number").columns
        ),
        value="expenditure_per_student"
    ),
    dcc.Graph(id="Graph")
])

@app.callback(
    Output("Graph", "figure"),
    Input("X Column Picker", "value")
)
def plot_rev_scatter(x):
    if not x:
        raise PreventUpdate
    fig = px.scatter(
        education,
        x=x,
        y="AVG_MATH_8_SCORE",
        trendline="ols",
        title=f"Relationship between {x} & AVG_MATH_8_SCORE"
    )
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```



# MULTI-SELECT DROPDOWN MENUS



You can create **multi-select dropdown menus** with the "multi" argument

- This passes a list to the callback function instead of a string

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.Dropdown(
        id="filter",
        options=education["STATE"].unique(),
        value=["CALIFORNIA", "OREGON"],
        multi=True
    ),
    dcc.Graph(id="graph")
])

@app.callback(
    Output("graph", "figure"),
    Input("filter", "value")
)
def plot_rev_scatter(state):
    fig = px.line(
        education.query("STATE in @state"),
        x="YEAR",
        y="TOTAL_EXPENDITURE",
        color="STATE",
        title="Expenditure Over Time"
    )
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```

*This sets the unique "STATE" values as the options, with "CALIFORNIA" and "OREGON" as the default, and enables multi-select*

*The "in" logical operator lets you process the list, and the "color" argument plots a series for each state*

# MULTI-SELECT DROPDOWN MENUS

Basic Interactivity

Interactive Elements

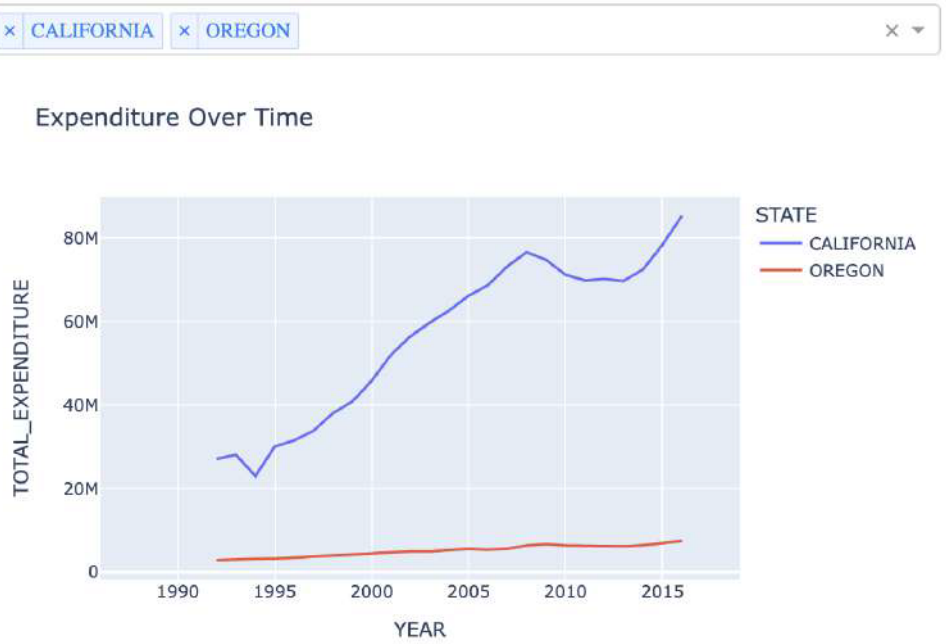
Multiple Input Callbacks

Multiple Output Callbacks

You can create **multi-select dropdown menus** with the "multi" argument

- This passes a list to the callback function instead of a string

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.Dropdown(
        id="filter",
        options=education["STATE"].unique(),
        value=["CALIFORNIA", "OREGON"],
        multi=True
    ),
    dcc.Graph(id="graph")
])

@app.callback(
    Output("graph", "figure"),
    Input("filter", "value")
)
def plot_rev_scatter(state):
    fig = px.line(
        education.query("STATE in @state"),
        x="YEAR",
        y="TOTAL_EXPENDITURE",
        color="STATE",
        title="Expenditure Over Time"
    )
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```



# CHECKLISTS

**Checklists** provide a list of options for the user to select *(or multi-select)*

- dcc.Checklist(id, options, value)


Basic Interactivity

Interactive Elements

Multiple Input Callbacks

Multiple Output Callbacks


```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.Checklist(
        id="scores",
        options=[
            "AVG_MATH_4_SCORE",
            "AVG_MATH_8_SCORE",
            "AVG_READING_4_SCORE",
            "AVG_READING_8_SCORE"
        ],
        value=["AVG_MATH_4_SCORE"]
    ),
    dcc.Graph(id="graph")
])

@app.callback(
    Output("graph", "figure"),
    Input("scores", "value")
)
def plot_rev_hist(scores):
    fig = px.histogram(
        education,
        x=scores,
        barmode="overlay",
        title="Score Distribution"
    ).update_traces(opacity=.6)
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```

*This sets the list of options (column names from the DataFrame) and the default selection*

*This passes the list of selected columns into the "x" argument of the histogram, plotting each one*

# CHECKLISTS

Basic Interactivity

Interactive Elements

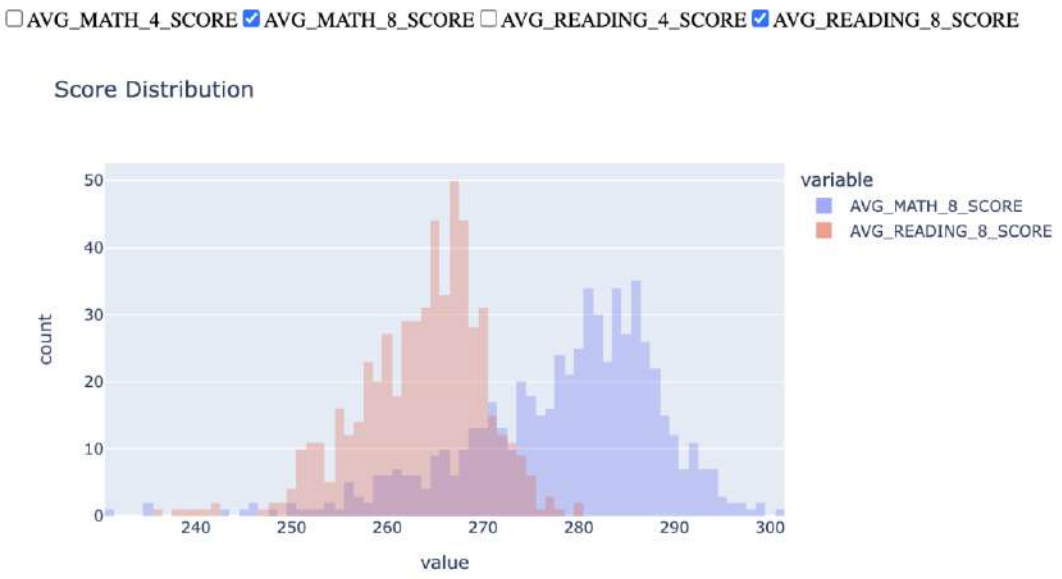
Multiple Input Callbacks

Multiple Output Callbacks

**Checklists** provide a list of options for the user to select *(or multi-select)*

- dcc.Checklist(id, options, value)

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.Checklist(
        id="scores",
        options=[
            "AVG_MATH_4_SCORE",
            "AVG_MATH_8_SCORE",
            "AVG_READING_4_SCORE",
            "AVG_READING_8_SCORE"
        ],
        value=["AVG_MATH_4_SCORE"]
    ),
    dcc.Graph(id="graph")
])

@app.callback(
    Output("graph", "figure"),
    Input("scores", "value")
)
def plot_rev_hist(scores):
    fig = px.histogram(
        education,
        x=scores,
        barmode="overlay",
        title="Score Distribution"
    ).update_traces(opacity=.6)
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```




**PRO TIP:** Checklists are great for 2-8 options at a time, but beyond that can add significant clutter – consider a multi-select dropdown if you need more!

# **ASSIGNMENT:** CHECKLISTS

1 **NEW MESSAGE**

*March 2024 ,20*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **Park Features Map**

Thanks for the help with the Map earlier!

Can you add a checklist that allows me to filter down to ski resorts that have Snow Parks, Night Skiing or both?

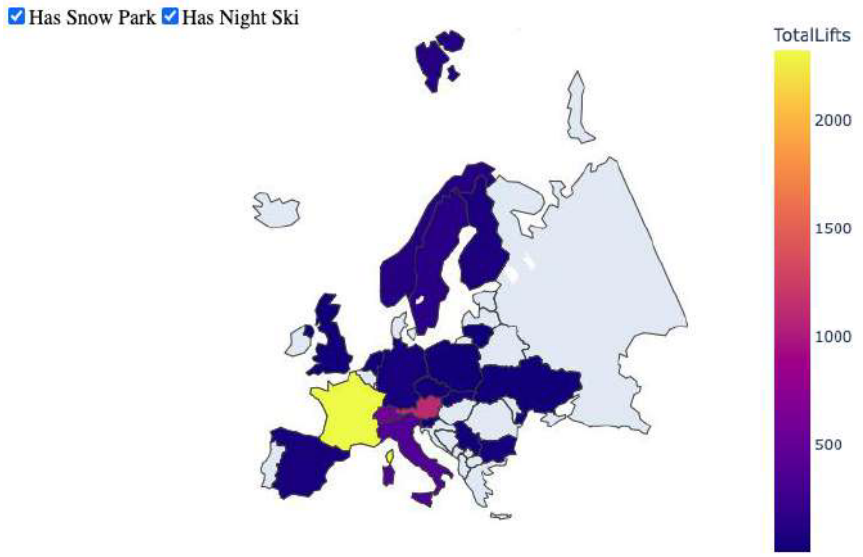
Will be really helpful to pinpoint which countries make the best destinations for clients looking for specific experiences.

Thanks!

*section03_assignments.ipynb*

Reply Forward

## *Results Preview*



# RADIO BUTTONS

**Radio buttons** provide a list of options for the user to toggle between

- dcc.RadioItems(id, options, value)

Basic Interactivity

Interactive Elements

Multiple Input Callbacks

Multiple Output Callbacks

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.RadioItems(
        id="OLS Toggle",
        options=["Trendline On", "Trendline Off"],
        value="Trendline Off"
),
    dcc.Graph(id="graph"),
])

@app.callback(
    Output('graph', 'figure'),
    Input("OLS Toggle", "value"),
)

def plot_rev_scatter(ols):
    fig = px.scatter(
        education,
        x="expenditure_per_student",
        y="AVG_MATH_8_SCORE",
        trendline=None if ols == "Trendline Off" else "ols",
        title = "The Relationship Between Spending and Test Scores",
        )
    return fig

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

*This sets the two options for the user to choose between, and sets the default*

*This adds an "ols" trendline if the selection is not "Trendline Off"*

# RADIO BUTTONS

**Radio buttons** provide a list of options for the user to toggle between

- dcc.RadioItems(id, options, value)

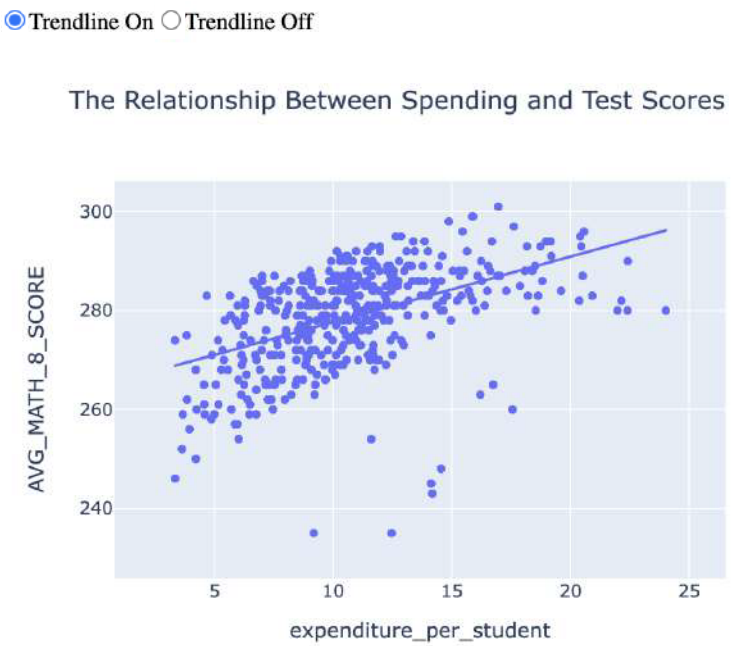
Basic Interactivity

Interactive Elements

Multiple Input Callbacks

Multiple Output Callbacks

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.RadioItems(
        id="OLS Toggle",
        options=["Trendline On", "Trendline Off"],
        value="Trendline Off"
),
    dcc.Graph(id="graph"),
])

@app.callback(
    Output('graph', 'figure'),
    Input("OLS Toggle", "value"),
)

def plot_rev_scatter(ols):
    fig = px.scatter(
        education,
        x="expenditure_per_student",
        y="AVG_MATH_8_SCORE",
        trendline=None if ols == "Trendline Off" else "ols",
        title = "The Relationship Between Spending and Test Scores",
        )
    return fig

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



# SLIDERS

Basic Interactivity

Interactive Elements

Multiple Input Callbacks

Multiple Output Callbacks

**Sliders** let users drag a handle to select a value inside a defined range

- dcc.Slider(min, max, step, value)

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.Slider(id="spend", min=0, max=30, step=5, value=15),
    dcc.Graph(id="graph"),
])

@app.callback(Output('graph', 'figure'), Input("spend", "value"))

def bar_chart(spending):
    fig = px.bar(
        (education
         .query("expenditure_per_student > @spending")
         .groupby("YEAR", as_index=False)
         .count()),
        x="YEAR",
        y="expenditure_per_student",
        title=f"States that spent over ${spending}K per student"
    )
    fig.update_yaxes(title="COUNT", range=[0, 50])
    fig.update_xaxes(range=[1991, 2017])
    return fig

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

*This starts the slider at 15 and lets the user select values between 0-30 in 5-step intervals*

*This filters the DataFrame for "expenditures per student" greater than the selected value, groups the results by year, and plots the number of rows (States) in a bar chart*

**PRO TIP:** These often align with Boolean operators, where users select options equal to, less than, or greater than some value

# SLIDERS

**Sliders** let users drag a handle to select a value inside a defined range

- dcc.Slider(min, max, step, value)

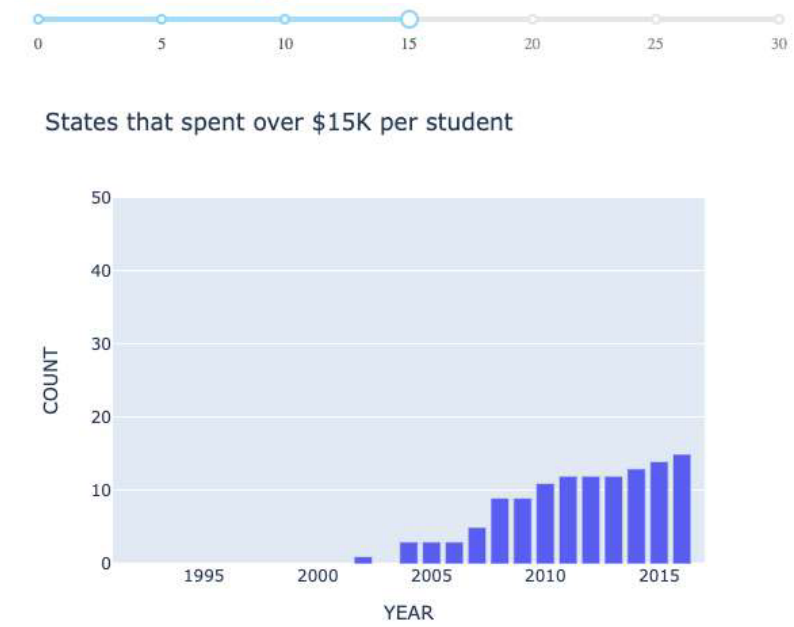
Basic Interactivity

Interactive Elements

Multiple Input Callbacks

Multiple Output Callbacks

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.Slider(id="spend", min=0, max=30, step=5, value=15),
    dcc.Graph(id="graph"),
])

@app.callback(Output('graph', 'figure'), Input("spend", "value"))

def bar_chart(spending):
    fig = px.bar(
        (education
         .query("expenditure_per_student > @spending")
         .groupby("YEAR", as_index=False)
         .count()),
        x="YEAR",
        y="expenditure_per_student",
        title=f"States that spent over ${spending}K per student"
    )
    fig.update_yaxes(title="COUNT", range=[0, 50])
    fig.update_xaxes(range=[1991, 2017])
    return fig

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



# RANGE SLIDERS

Basic Interactivity

Interactive Elements

Multiple Input Callbacks

Multiple Output Callbacks

**Range sliders** let users drag two handles to select a range of values

- dcc.RangeSlider(min, max, step, value)

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.RangeSlider(id="spend", min=0, max=30, step=5, value=[10,15]),
    dcc.Graph(id="graph"),
])

@app.callback(Output('graph', 'figure'), Input("spend", "value"))

def bar_chart(spending):
    fig = px.bar(
        (education
         .query("@spending[0] <= expenditure_per_student <= @spending[1]")
         .groupby("YEAR", as_index=False)
         .count()),
        x="YEAR",
        y="expenditure_per_student",
        title=f"States that spent between ${spending[0]}-{spending[1]}K per student"
    )
    fig.update_yaxes(title="COUNT", range=[0, 50])
    fig.update_xaxes(range=[1991, 2017])
    return fig

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

*This sets a list (or tuple) as the value*

*This filters the DataFrame using the **.between()** method instead of a Boolean operator*

*(Note that the values are accessed with their index)*

**PRO TIP:** Range Sliders let you perform "between" selections rather than single value or single directions selections (like with traditional sliders)

# RANGE SLIDERS

Basic Interactivity

Interactive Elements

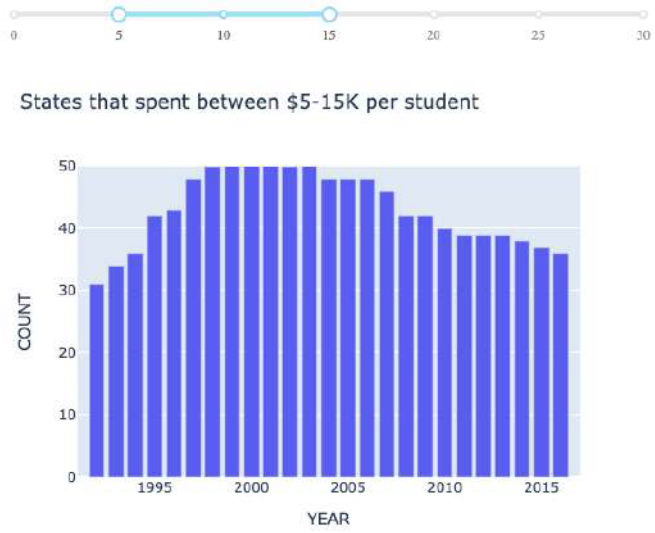
Multiple Input Callbacks

Multiple Output Callbacks

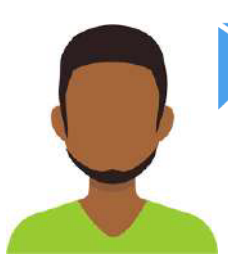
**Range sliders** let users drag two handles to select a range of values

- dcc.RangeSlider(min, max, step, value)

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.RangeSlider(id="spend", min=0, max=30, step=5, value=[10,15]),
    dcc.Graph(id="graph"),
])

@app.callback(Output('graph', 'figure'), Input("spend", "value"))

def bar_chart(spending):
    fig = px.bar(
        (education
         .query("@spending[0] <= expenditure_per_student <= @spending[1]")
         .groupby("YEAR", as_index=False)
         .count()),
        x="YEAR",
        y="expenditure_per_student",
        title=f"States that spent between ${spending[0]}-{spending[1]}K per student"
    )
    fig.update_yaxes(title="COUNT", range=[0, 50])
    fig.update_xaxes(range=[1991, 2017])
    return fig

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



# ASSIGNMENT: SLIDERS



1 **NEW MESSAGE**

*March 2024 ,23*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **Elevation Selector**

Ok,

I'd like to update another visual you created before with interactivity.

Some of our customers love to get HIGH (in terms of elevation, of course).

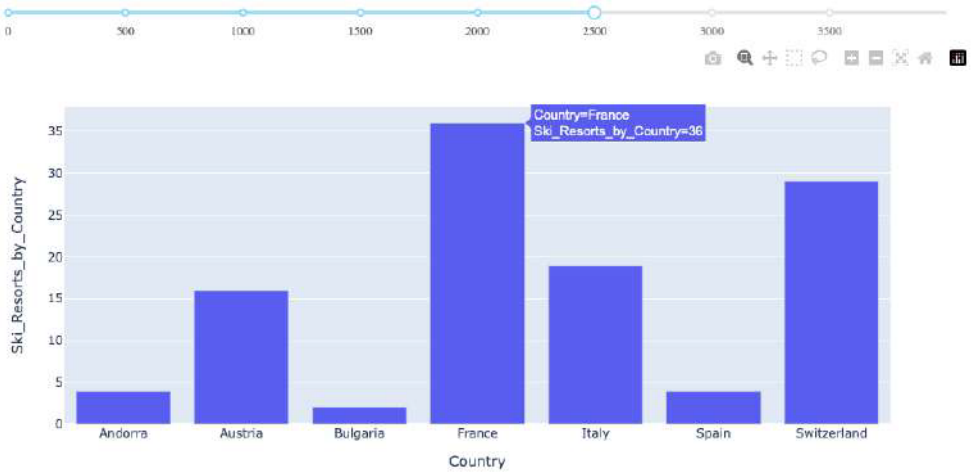
Can you build a bar chart that shows number of ski resorts by country based on the elevation users select?

Thanks!

*section03_assignments.ipynb*

Reply Forward

## *Results Preview*



# DATE PICKERS

Basic Interactivity

Interactive Elements

Multiple Input Callbacks

Multiple Output Callbacks

**Date pickers** let users select a date from a calendar drop down

- dcc.DatePickerSingle(id, min_date_allowed, max_date_allowed, initial_visible_month, date, display_format)

```
collisions.head()
```

| | DATE | BOROUGH | COLLISIONS |
|---|---|---|---|
| 0 | 2012-07-01 | BRONX | 39 |
| 1 | 2012-07-01 | BROOKLYN | 135 |
| 2 | 2012-07-01 | MANHATTAN | 119 |
| 3 | 2012-07-01 | QUEENS | 101 |
| 4 | 2012-07-01 | STATEN ISLAND | 26 |

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.DatePickerSingle(
        id="date picker",
        min_date_allowed=collisions["DATE"].min(),
        max_date_allowed=collisions["DATE"].max(),
        initial_visible_month=collisions["DATE"].max(),
        date=collisions["DATE"].max(),
        display_format="YYYY-MM-DD"
    ),
    dcc.Graph(id="graph")
])

@app.callback(Output("graph", "figure"), Input("date picker", "date"))
def plot_collisions_bar(date):
    fig = px.bar(
        collisions.loc[collisions["DATE"].eq(date)],
        x="COLLISIONS",
        y="BOROUGH",
        title=f"Traffic Accidents in NYC on {date}"
    )
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```

*This sets the min/max dates in the calendar from the data, and sets the max date as the starting value*

*Note that the input property is **"date"**, not "value"*

*This plots the collisions for the selected date on a bar chart by borough*

# DATE PICKERS

Basic Interactivity

Interactive Elements

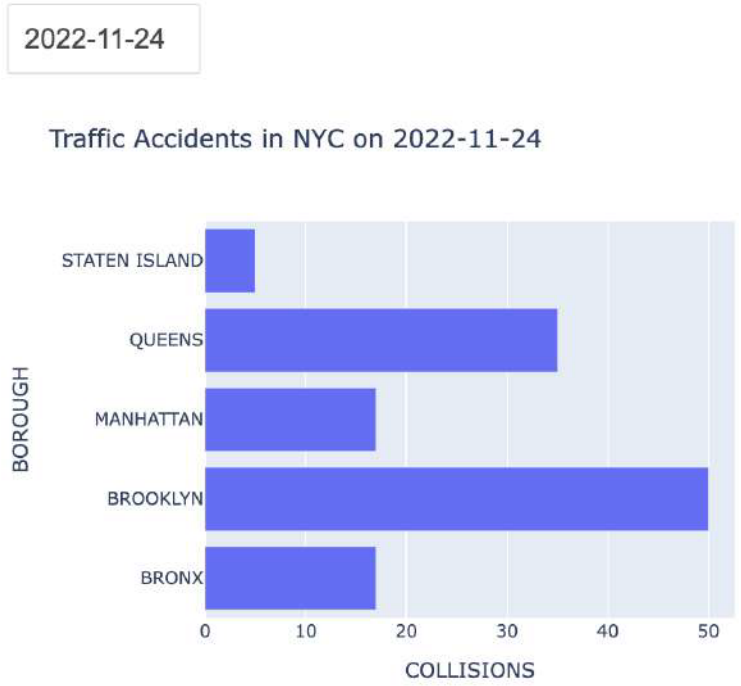
Multiple Input Callbacks

Multiple Output Callbacks

**Date pickers** let users select a date from a calendar drop down

- dcc.DatePickerSingle(id, min_date_allowed, max_date_allowed, initial_visible_month, date, display_format)

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.DatePickerSingle(
        id="date picker",
        min_date_allowed=collisions["DATE"].min(),
        max_date_allowed=collisions["DATE"].max(),
        initial_visible_month=collisions["DATE"].max(),
        date=collisions["DATE"].max(),
        display_format="YYYY-MM-DD"
    ),
    dcc.Graph(id="graph")
])

@app.callback(Output("graph", "figure"), Input("date picker", "date"))
def plot_collisions_bar(date):
    fig = px.bar(
        collisions.loc[collisions["DATE"].eq(date)],
        x="COLLISIONS",
        y="BOROUGH",
        title=f"Traffic Accidents in NYC on {date}"
    )
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```



# DATE RANGE PICKERS

Basic Interactivity

Interactive Elements

Multiple Input Callbacks

Multiple Output Callbacks

**Date range pickers** let users select a range of dates from calendar drop downs

- dcc.DatePickerRange(id, start_date, end_date, display_format)

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.DatePickerRange(
        id="dates",
        start_date=collisions["DATE"].min(),
        end_date=collisions["DATE"].max(),
        display_format="YYYY-MM-DD"
    ),
    dcc.Graph(id="graph")
])

@app.callback(
    Output("graph", "figure"),
    [Input("dates", "start_date"),Input("dates", "end_date")]
)
def plot_rev_line(start, end):
    fig = px.line(
        (collisions
         .loc[collisions["DATE"].between(start, end)]
         .groupby("DATE", as_index=False)
         .sum()),
        x="DATE",
        y="COLLISIONS",
        title=f"Traffic Accidents in NYC"
    )
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```

*This sets the min/max dates in the data as the start/end dates for the calendar dropdowns*

*Note that **two inputs** are needed (they don't need to be in a list, but it keeps them organized)*

*This filters the "DATE" using the range selected, groups the DataFrame by "DATE", and sums the collisions, plotting them in a line chart*

# DATE RANGE PICKERS

Basic Interactivity

Interactive Elements

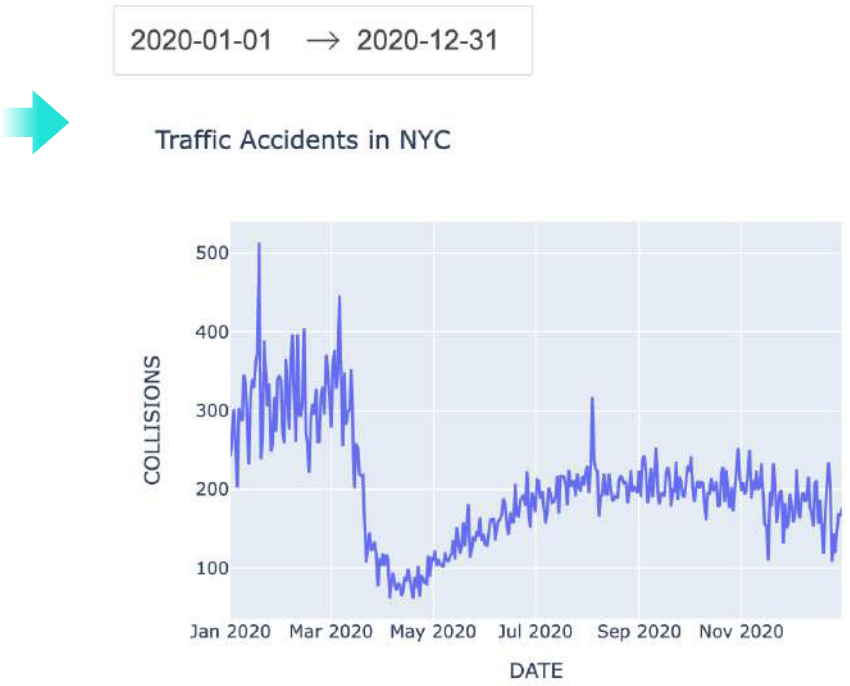
Multiple Input Callbacks

Multiple Output Callbacks

**Date range pickers** let users select a range of dates from calendar drop downs

- dcc.DatePickerRange(id, start_date, end_date, display_format)

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.DatePickerRange(
        id="dates",
        start_date=collisions["DATE"].min(),
        end_date=collisions["DATE"].max(),
        display_format="YYYY-MM-DD"
    ),
    dcc.Graph(id="graph")
])

@app.callback(
    Output("graph", "figure"),
    [Input("dates", "start_date"),Input("dates", "end_date")]
)
def plot_rev_line(start, end):
    fig = px.line(
        (collisions
         .loc[collisions["DATE"].between(start, end)]
         .groupby("DATE", as_index=False)
         .sum()),
        x="DATE",
        y="COLLISIONS",
        title=f"Traffic Accidents in NYC"
    )
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```



# MULTIPLE INPUT CALLBACKS

- Basic Interactivity
- Interactive Elements
- Multiple Input Callbacks
- Multiple Output Callbacks

## A single callback function can have **multiple inputs**

- This lets you add multiple interactive elements to the same application!

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.Dropdown(
        id="x column",
        options=list(education.select_dtypes(include="number").columns),
        value="expenditure_per_student"
    ),
    dcc.Dropdown(
        id="y column",
        options=list(education.select_dtypes(include="number").columns),
        value="AVG_MATH_8_SCORE"
    ),
    dcc.Graph(id="graph")
])

@app.callback(
    Output("graph", "figure"),
    Input("x column", "value"),
    Input("y column", "value")
)
def plot_rev_scatter(x, y):
    fig = px.scatter(
        education,
        x=x,
        y=y,
        trendline="ols"
    )
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```

*This creates **two dropdowns** for the user to select DataFrame columns with*

*Note that **two inputs** are needed, and they are passed into the function in the same order*

*This sets the "x" and "y" variables for the scatterplot using the selections*

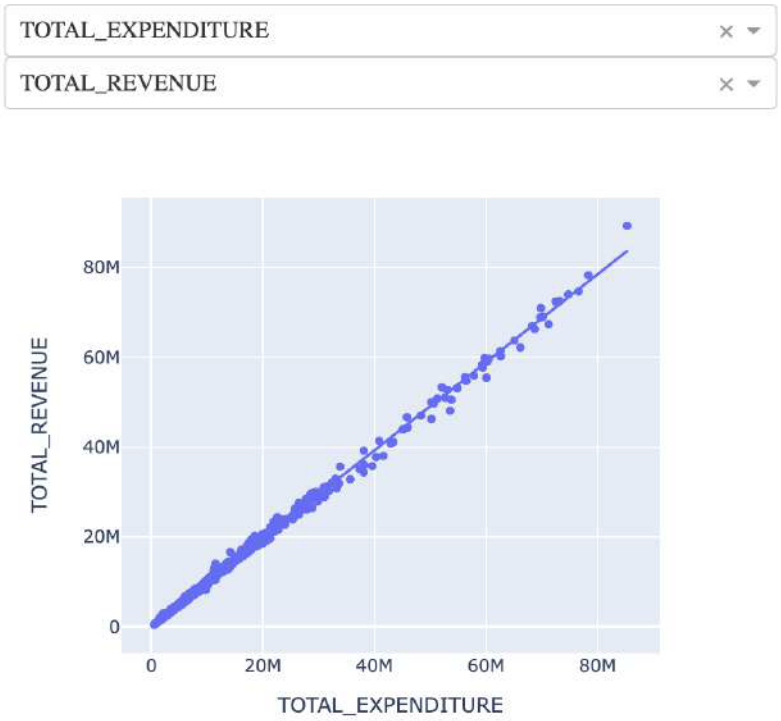
# MULTIPLE INPUT CALLBACKS

- Basic Interactivity
- Interactive Elements
- Multiple Input Callbacks
- Multiple Output Callbacks

A single callback function can have **multiple inputs**

- This lets you add multiple interactive elements to the same application!

```
app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.Dropdown(
        id="x column",
        options=list(education.select_dtypes(include="number").columns),
        value="expenditure_per_student"
    ),
    dcc.Dropdown(
        id="y column",
        options=list(education.select_dtypes(include="number").columns),
        value="AVG_MATH_8_SCORE"
    ),
    dcc.Graph(id="graph")
])

@app.callback(
    Output("graph", "figure"),
    Input("x column", "value"),
    Input("y column", "value")
)
def plot_rev_scatter(x, y):
    fig = px.scatter(
        education,
        x=x,
        y=y,
        trendline="ols"
    )
    return fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```



# MULTIPLE OUTPUT CALLBACKS

Basic Interactivity

Interactive Elements

Multiple Input Callbacks

Multiple Output Callbacks

## A single callback function can also return **multiple outputs**

- This can help modify text or multiple charts with a single interactive element *(or several!)*

```
app = JupyterDash(__name__)

app.layout = html.Div([
    html.H2(id="header"),
    dcc.Dropdown(
        id="x column",
        options=list(education.select_dtypes(include="number").columns),
        value="expenditure_per_student"
    ),
    dcc.Dropdown(
        id="y column",
        options=list(education.select_dtypes(include="number").columns),
        value="AVG_MATH_8_SCORE"
    ),
    dcc.Graph(id="graph")
])

@app.callback(
    Output("header", "children"),
    Output("graph", "figure"),
    Input("x column", "value"),
    Input("y column", "value")
)
def plot_rev_scatter(x, y):
    fig = px.scatter(
        education,
        x=x,
        y=y,
        trendline="ols"
    )
    header = f"{x.title().replace('_', ' ')} vs {y.title().replace('_', ' ')}"
    return header, fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```

*This creates an empty header to be updated by the callback function (at the same time as the graph)*

*Note that **two outputs** are used, one for the "graph" and the other for the "header"*

*The order must follow the order of the outputs!*

# MULTIPLE OUTPUT CALLBACKS

A single callback function can also return **multiple outputs**

- This can help modify text or multiple charts with a single interactive element *(or several!)*

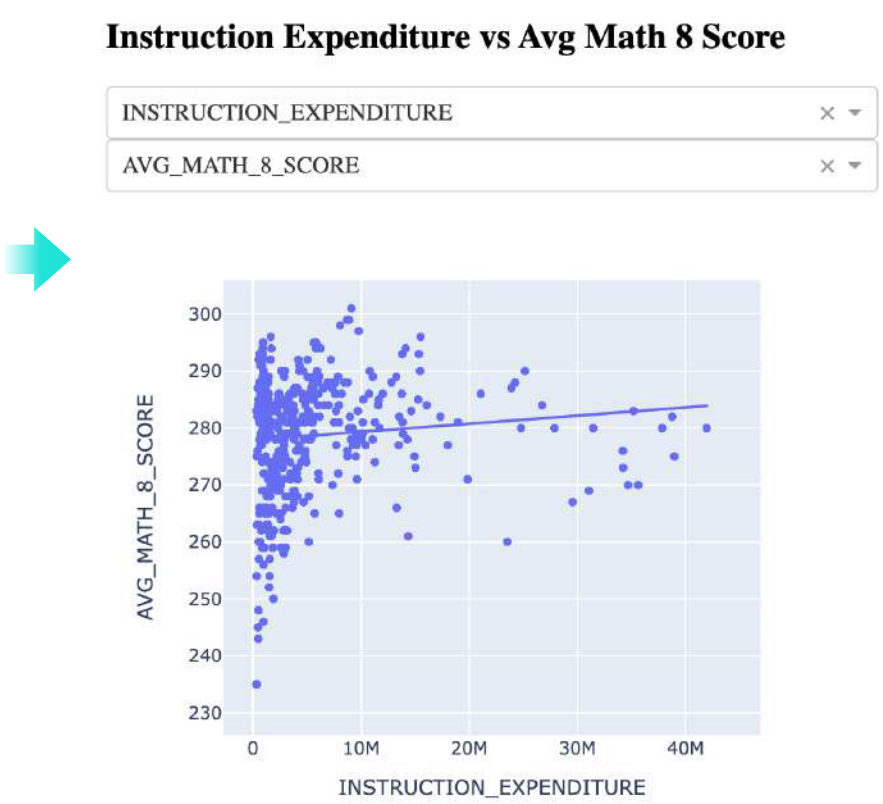
Basic Interactivity

Interactive Elements

Multiple Input Callbacks

Multiple Output Callbacks

```
app = JupyterDash(__name__)

app.layout = html.Div([
    html.H2(id="header"),
    dcc.Dropdown(
        id="x column",
        options=list(education.select_dtypes(include="number").columns),
        value="expenditure_per_student"
    ),
    dcc.Dropdown(
        id="y column",
        options=list(education.select_dtypes(include="number").columns),
        value="AVG_MATH_8_SCORE"
    ),
    dcc.Graph(id="graph")
])

@app.callback(
    Output("header", "children"),
    Output("graph", "figure"),
    Input("x column", "value"),
    Input("y column", "value")
)
def plot_rev_scatter(x, y):
    fig = px.scatter(
        education,
        x=x,
        y=y,
        trendline="ols"
    )
    header = f"{x.title().replace('_', ' ')} vs {y.title().replace('_', ' ')}"
    return header, fig

if __name__ == "__main__":
    app.run_server(mode="inline")
```



# **ASSIGNMENT**: MULTIPLE INTERACTIVE ELEMENTS

1 **NEW MESSAGE**

*March 2024 ,25*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **Single Map Feature View**

Hey there,

Can we apply the elevation slider to the map-based view?

It's much more helpful given we're working with geographic areas (please keep the Snow Park & Night Ski checklist in the same map too!)

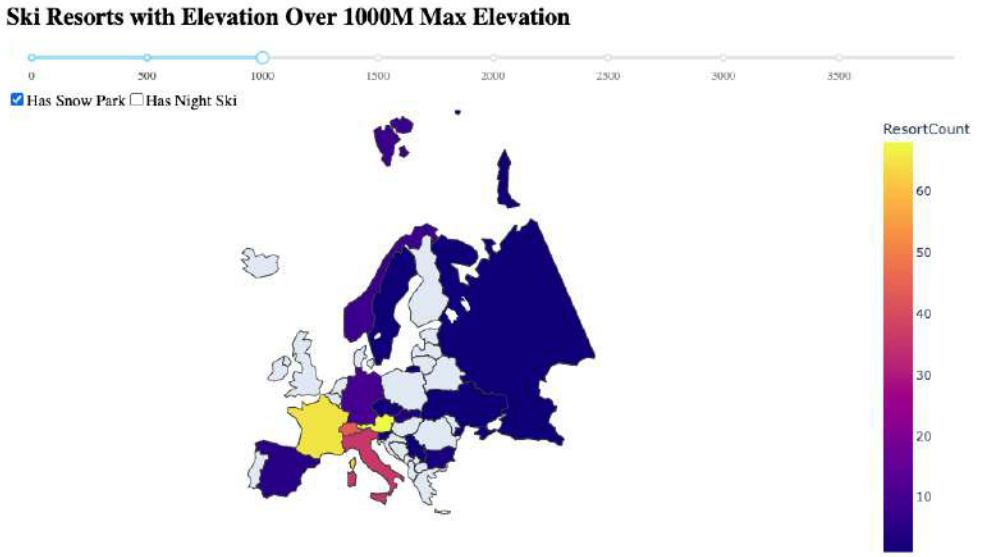
If you're daring, try to embed the elevation selected in the header of the chart as well!

Thanks!

*Section03_assignments.ipynb*

Reply Forward

## *Results Preview*



# KEY TAKEAWAYS

## Dash has a wide range of **interactive elements** you can use in apps

- *Choose the right one by considering the data types, number of options, and filtering criteria (logical comparisons)*

## Use dropdowns, checklists, and radio buttons for **categories**

- *Checklists and radio buttons are great for a limited number of options, while dropdown menus can incorporate dozens while keeping your applications compact*

## Use sliders and date pickers for **numbers** & **dates**

- *Singe value sliders and date pickers work well with inequalities, while range options are best for "between" logic*

## Callback functions can have **multiple inputs** & **outputs**

- *Multiple outputs let you modify several visuals or text using a single interactive element*
- *Multiple inputs let you use several interactive elements to modify text and visuals in your app*

# MID-COURSE PROJECT

# PROJECT DATA: US & CANADA RESORTS

```
resorts.head()
```

| | ID | Resort | Latitude | Longitude | Country | Continent | Price | Season | Highest point | Lowest point | ... | Snow cannons | Surface lifts | Chair lifts | Gondola lifts | Total lifts | Lift capacity |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 4 | Red Mountain Resort-Rossland | 49.105520 | -117.846280 | Canada | North America | 60 | December - April | 2075 | 1185 | ... | 0 | 2 | 5 | 1 | 8 | 9200 |
| **1** | 11 | Fernie | 49.504175 | -115.062867 | Canada | North America | 67 | December - April | 2134 | 1052 | ... | 11 | 3 | 7 | 0 | 10 | 14514 |
| **2** | 12 | Sun Peaks | 50.884468 | -119.882329 | Canada | North America | 62 | November - April | 2082 | 1198 | ... | 0 | 6 | 6 | 0 | 12 | 13895 |
| **3** | 13 | Panorama | 50.736999 | -119.120561 | Canada | North America | 62 | December - April | 2365 | 1140 | ... | 0 | 3 | 6 | 4 | 13 | 11890 |
| **4** | 22 | Steamboat | 35.754022 | -109.853751 | United States | North America | 120 | November - April | 3221 | 2103 | ... | 0 | 1 | 14 | 2 | 17 | 32720 |

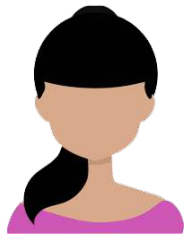
5 rows × 25 columns

```
resorts.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 98 entries, 0 to 97
Data columns (total 25 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   ID                    98 non-null     int64
 1   Resort                98 non-null     object
 2   Latitude              98 non-null     float64
 3   Longitude             98 non-null     float64
 4   Country               98 non-null     object
 5   Continent             98 non-null     object
 6   Price                 98 non-null     int64
 7   Season                98 non-null     object
 8   Highest point         98 non-null     int64
 9   Lowest point          98 non-null     int64
 10  Beginner slopes       98 non-null     int64
 11  Intermediate slopes   98 non-null     int64
 12  Difficult slopes      98 non-null     int64
 13  Total slopes          98 non-null     int64
 14  Longest run           98 non-null     int64
 15  Snow cannons          98 non-null     int64
 16  Surface lifts         98 non-null     int64
 17  Chair lifts           98 non-null     int64
 18  Gondola lifts         98 non-null     int64
 19  Total lifts           98 non-null     int64
 20  Lift capacity         98 non-null     int64
 21  Child friendly        98 non-null     object
 22  Snowparks             98 non-null     object
 23  Nightskiing           98 non-null     object
 24  Summer skiing         98 non-null     object
dtypes: float64(2), int64(15), object(8)
memory usage: 19.3+ KB
```

MAVELUXE

# **ASSIGNMENT:** MIDCOURSE PROJECT



**NEW MESSAGE**

*March 28, 2024*

From: **Deepthi Downhill** *(VP of Analytics)*

Subject: **More Ambitious Ski Resort App**

Hello,

The work you've been doing with Leonard is very exciting. This type of application can save our agents hundreds of hours annually! I want to applaud you both on this amazing initiative.

That said, it's time to think a bit bigger. While Europe is a solid market, it's behind the US and Canada for us given our customers are almost exclusively from North America. Can you create two apps that will help us with these markets?

Thanks!

*section04_midcourse_project.ipynb*

Reply Forward

## *Key Objectives*

1. Build two working Dash Applications
2. Add multiple chart types and interactive elements
3. Connect them with callback functions capable of taking multiple inputs and returning multiple outputs

MAVELUXE

# DASHBOARD LAYOUTS

# DASHBOARD LAYOUTS

In this section we'll introduce dashboard design principles and build **dashboard layouts** in Dash, including some more advanced HTML and the Dash Bootstrap Components library

## TOPICS WE'LL COVER:

- Dashboard Basics
- HTML Layouts
- App Styling
- Dash Bootstrap Components

## GOALS FOR THIS SECTION:

- Identify the types of dashboards, their key elements, and design principles for effective dashboard layouts
- Create dashboard layouts using HTML, markdown, and Dash Bootstrap Components
- Add custom formatting to layouts using themes or by styling each component individually

# DASHBOARDS 101

- Dashboard Basics
- HTML Layouts
- App Styling
- Dash Bootstrap Components

**Dashboards** are groups of visuals that help understand data and make decisions

- They can be used for both **exploratory** and **explanatory** analysis

## EXPLORATORY

- Goal is to **explore and profile** the data to see what insights emerge
- Helps you understand the data and identify interesting patterns & trends

## EXPLANATORY

- Goal is to **tell a specific story** or explain what happened and why
- Identifies key business drivers and delivers insights & recommendations

We'll mostly focus on dashboards for **exploratory analysis** in this course, but you can check out our Data Visualization with Matplotlib & Seaborn for good explanatory examples

# DASHBOARD ELEMENTS



There are two main types of **dashboard elements**:

## 1 Metrics & visuals

- What are the main metrics (KPIs) the dashboard needs to track?
- What other metrics add context to complement these?
- What chart type is appropriate to visualize each metric?

## 2 Filters & interactivity

- Will users need to see specific, filtered views?
- Will they need to drill up or down to different levels of granularity?

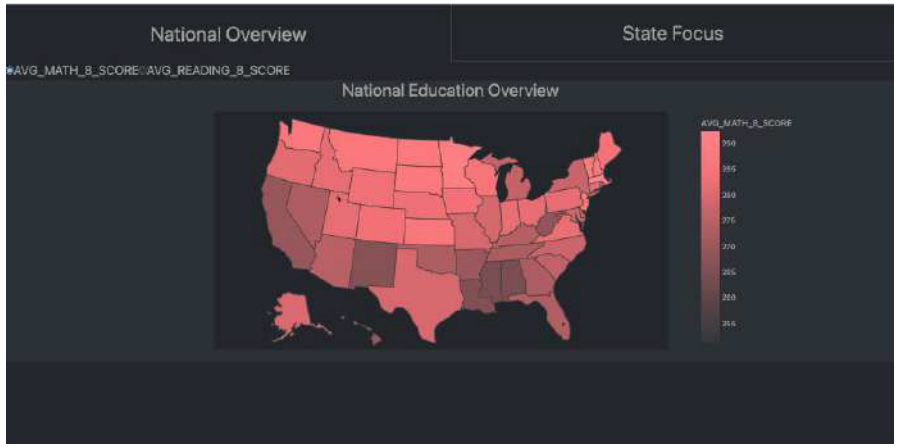
**PRO TIP:** Think like a **business owner** before you think like an analyst; before you begin building your dashboard with code, take time to understand the outcomes you're trying to impact, the key stakeholders and their motivations, and the specific purpose your dashboard will serve

# DASHBOARD LAYOUTS

- Dashboard Basics
- HTML Layouts
- App Styling
- Dash Bootstrap Components

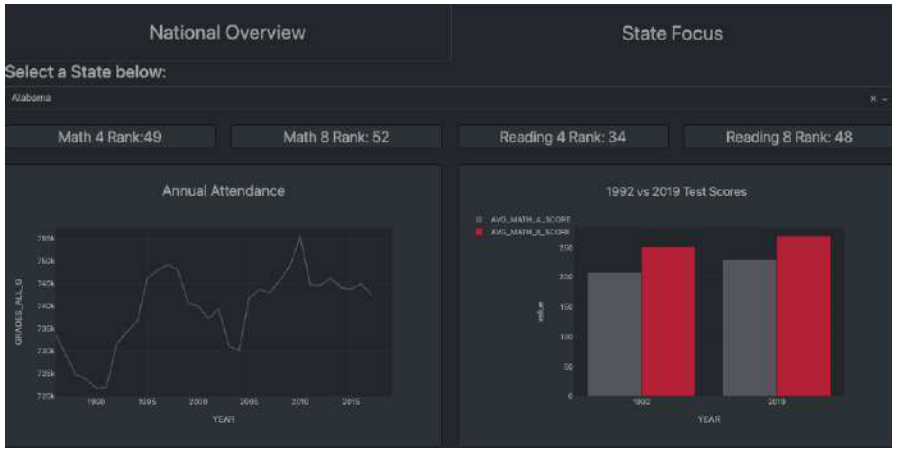
A strong **dashboard layout** adds cohesion to its visuals & interactivity, drawing attention to key metrics and guiding the viewer through a logical progression

***First tab***




*Nationwide view of test performance and other KPIs*

***Second tab***




*State-level deep dive with context on relative ranks*

**PRO TIP:** Design your dashboard layout like an **inverse pyramid**; the most important metrics and visuals should come first, followed by any supporting data or more granular views

# RECAP: HTML LAYOUTS

- Dashboard Basics
- HTML Layouts
- App Styling
- Dash Bootstrap Components

Dash uses **HTML layouts** for designing the front-end of the application

- Use the **html** module to specify the visual components and assign it to **app.layout**

```
app.layout = html.Div([
    html.H2(id="header"),
    html.P("Select a State Below"),
    dcc.Dropdown(options=["Alabama"], id="State Dropdown"),
    dcc.Graph(id="Enrollment Line"),
    html.P("Important footnote about this data")
])
```

*The **html.Div** works as a container for its "children", which can be html or dcc components*

Styles specified for html Divs will **cascade to standard html elements** within them, but dcc components override these styles by default (more on this later!)

# RECAP: HTML LAYOUTS

- Dashboard Basics
- HTML Layouts
- App Styling
- Dash Bootstrap Components

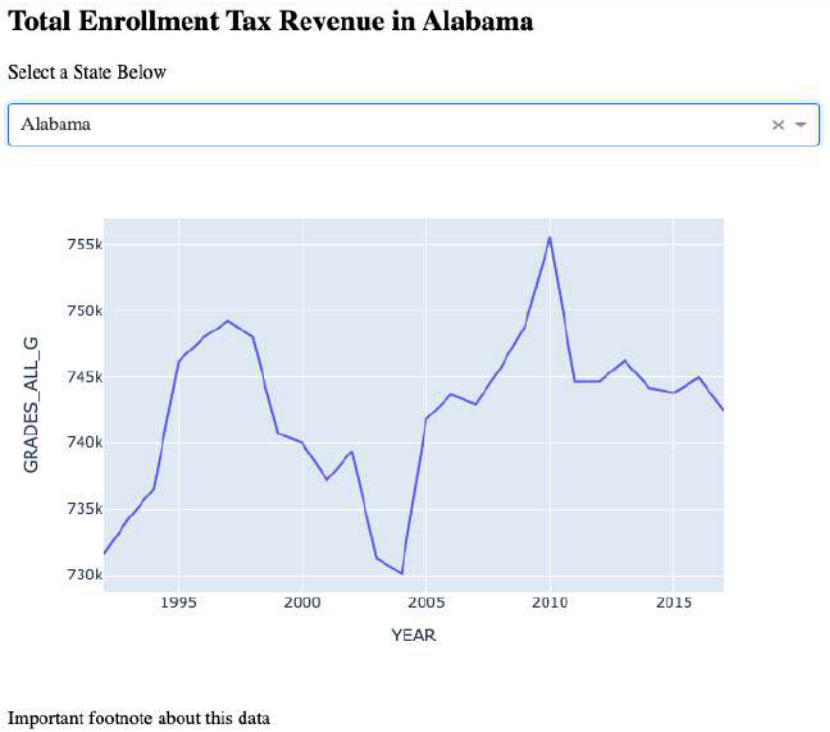
Dash uses **HTML layouts** for designing the front-end of the application

- Use the **html** module to specify the visual components and assign it to **app.layout**

```
app.layout = html.Div([
    html.H2(id="header"),
    html.P("Select a State Below"),
    dcc.Dropdown(options=["Alabama"], id="State Dropdown"),
    dcc.Graph(id="Enrollment Line"),
    html.P("Important footnote about this data")
])
```

*The html module writes html code for us!*

```
▼<div id="_dash-app-content">
  ▼<div>
    <h2 id="header">Total Enrollment Tax Revenue in Alabama</h2>
    <p>Select a State Below</p>
   ▶<div id="State Dropdown" class="dash-dropdown">…</div>
   ▶<div id="Enrollment Line" class="dash-graph">…</div>
    <p>Important footnote about this data</p>
```



# HTML COMPONENT CHEATSHEET



These are the basic **HTML components** you need to get started:

| Component | Description |
|---|---|
| html.Div() | A web page section (you can use multiple Divs to create sections with different styles) |
| html.H1(), H2(), ..., H6() | Different sized headers used to denote hierarchy or importance (more so than size itself) |
| html.P() | A paragraph, or generic body text, often smaller than and placed immediately below a header |
| html.Span() | Inline containers used to apply different colors or styles to text within headers or paragraphs |

```
app.layout = html.Div([
    html.H1("This is a Header"),
    html.H2("This is a Header"),
    html.H3("This is a Header"),
    html.H4("This is a Header"),
    html.H5("This is a Header"),
    html.H6("This is a Header"),
    html.P([
        "This is a ",
        html.Span("Paragraph", style={"color": "red"}),
        html.Span(", or body text.")
    ])

])
```


*For a full list of components, visit:* ***https://dash.plotly.com/dash-html-components***

# PRO TIP: MARKDOWN

You can also use **dcc.Markdown()** to embed text into the front-end of your app



```
app.layout = html.Div([
    dcc.Markdown('''
        # One Hash is the biggest header.
        ## Two Hashes is smaller
        ### Three is even smaller
        #### Four... and so on
        ##### Five... and so on
        ###### Until six, which is the smallest

        Type without hashes for body text.

        * Asterisks
        * For
        * Bullets

        1, Number and Period
        2. For Numbered Lists

        *italic* **bold** ***bold and italic***
        '''
    )
])
```

**One Hash is the biggest header.**

**Two Hashes is smaller**

**Three is even smaller**

**Four... and so on**

**Five... and so on**

**Until six, which is the smallest**

Type without hashes for body text.

- Asterisks
- For
- Bullets

1, Number and Period 2. For Numbered Lists

*italic* **bold** ***bold and italic***

**PRO TIP:** Markdown is easier to write and more convenient for things like modifying font weight and building lists than HTML – it's a bit harder to style, but generally more than sufficient for most apps!

# **ASSIGNMENT**: HTML & MARKDOWN

1 **NEW MESSAGE**

*April 2024 ,1*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **HTML?**

Hey there,

Are you familiar with HTML?

I was talking to our designer who is slammed, but I don't want to delay – can you send me a basic HTML layout to show what you can do?

If it's decent, we can move forward with styling our app.

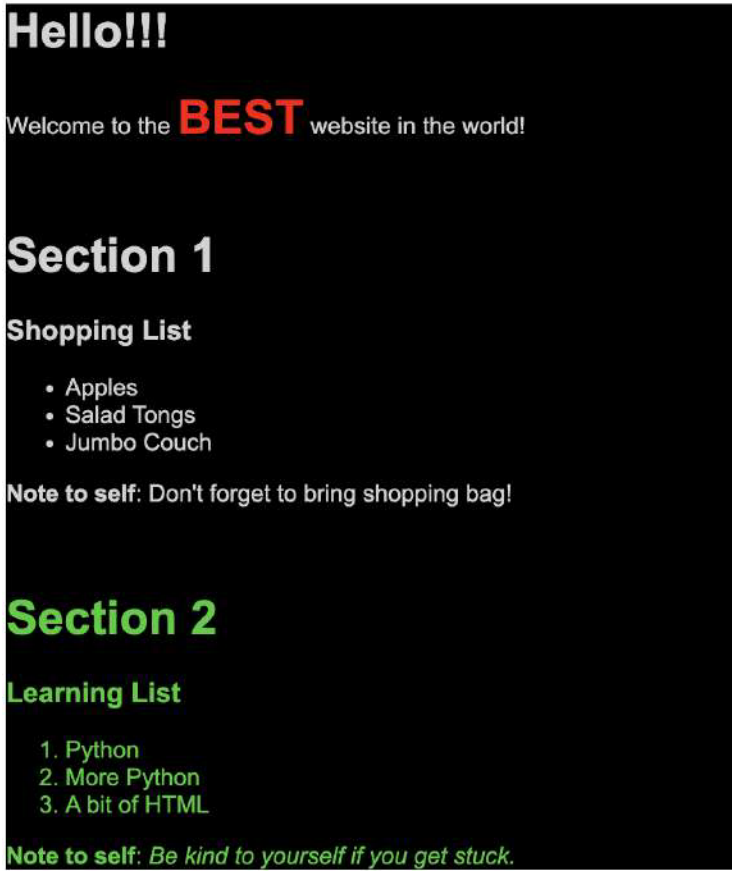
More details in the notebook!

Thanks!

*Section05_assignments.ipynb*

Reply Forward

## *Results Preview*



# STYLING HTML



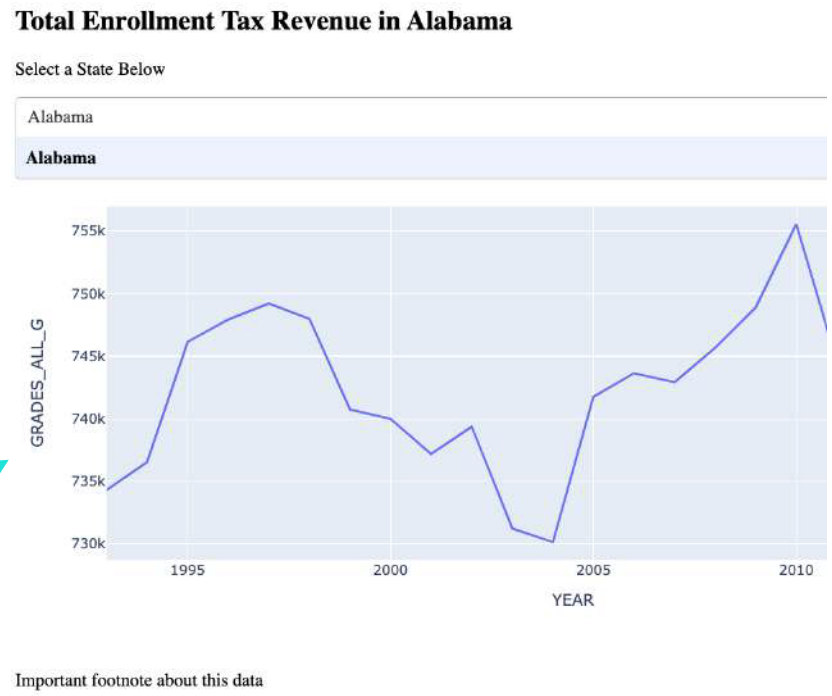
You can **style HTML components** with the "style" argument

- This lets you modify things like font types, sizes, and colors, as well as the background color

```
app.layout = html.Div([
    html.H2(id="header"),
    html.P("Select a State Below"),
    dcc.Dropdown(options=["Alabama"], id="State Dropdown"),
    dcc.Graph(id="Enrollment Line"),
    html.P("Important footnote about this data")
])
```

*No style has been specified yet*

*The default style for Dash apps and their components is black text with white backgrounds*



# STYLING HTML



You can **style HTML components** with the "style" argument

- This lets you modify things like font types, sizes, and colors, as well as the background color

```
my_dash_app_style={
    "color":"lightgrey",
    "backgroundColor": "black",
    "fontFamily": "Arial"
}

app.layout = html.Div(
    style=my_dash_app_style,
    children = [
        html.H2(id="header"),
        html.P("Select a State Below"),
        dcc.Dropdown(options=["Alabama"], id="State Dropdown"),
        dcc.Graph(id="Enrollment Line"),
        html.P(
            "Important footnote about this data",
            style={"backgroundColor": "grey", "fontSize": 8}
        )
])
```

*You can create a style dictionary in advance and assign it to a variable that can be reused in the app*

*Styles applied to an outer Div are passed through to (most) of its children – more on this later!*

*You can always apply a style to an individual html component directly to override any parent styles*

*This sets a light grey Arial font on a black background for the app components, except for the final paragraph (this has a grey background color with an absurdly small font size)*

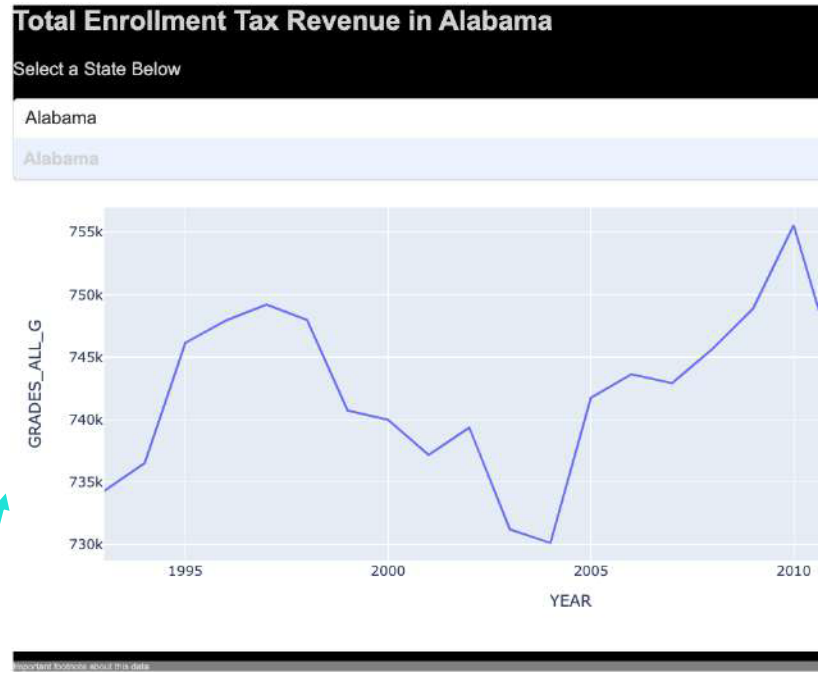
# STYLING HTML



You can **style HTML components** with the "style" argument

- This lets you modify things like font types, sizes, and colors, as well as the background color

```
my_dash_app_style={
    "color":"lightgrey",
    "backgroundColor": "black",
    "fontFamily": "Arial"
}

app.layout = html.Div(
    style=my_dash_app_style,
    children = [
        html.H2(id="header"),
        html.P("Select a State Below"),
        dcc.Dropdown(options=["Alabama"], id="State Dropdown"),
        dcc.Graph(id="Enrollment Line"),
        html.P(
            "Important footnote about this data",
            style={"backgroundColor": "grey", "fontSize": 8}
        )
])
```




*Note that the dropdown and chart didn't inherit the parent style!*

# HTML STYLE CHEATSHEET

Dashboard Basics

HTML Layouts

App Styling

Dash Bootstrap Components

These are the basic arguments for modifying **HTML styles** you need to get started:

| Argument | Examples |
|---|---|
| color | “red”, “#FF0000” |
| backgroundColor | “red”, “#FF0000” |
| fontSize | 12, 14, 20 |
| fontFamily | “Arial”, “Calibri” |
| text-align | “center”, “left”, “right” |

**PRO TIP:** Consider defining style dictionaries in advance to improve readability within the front-end code

```
disclaimer_style = {"color": "red", "fontSize": 12}
bluewhite_style = {
            "color": "white",
            "backgroundColor": "darkblue",
            "fontSize": 20,
            "fontFamily":"Microsoft Sans Serif",
            "text-align": "center"
            }

app.layout = html.Div([
     html.P([
        "Defaults or Parent style applied if Style not specified. ",
        html.Span(
            "Spans apply different syles within an HTML element.",
            style=disclaimer_style
        ),
    html.P(
        "Here are all the basic properties specified.",
        style=bluewhite_style
        )
    ])
])
```

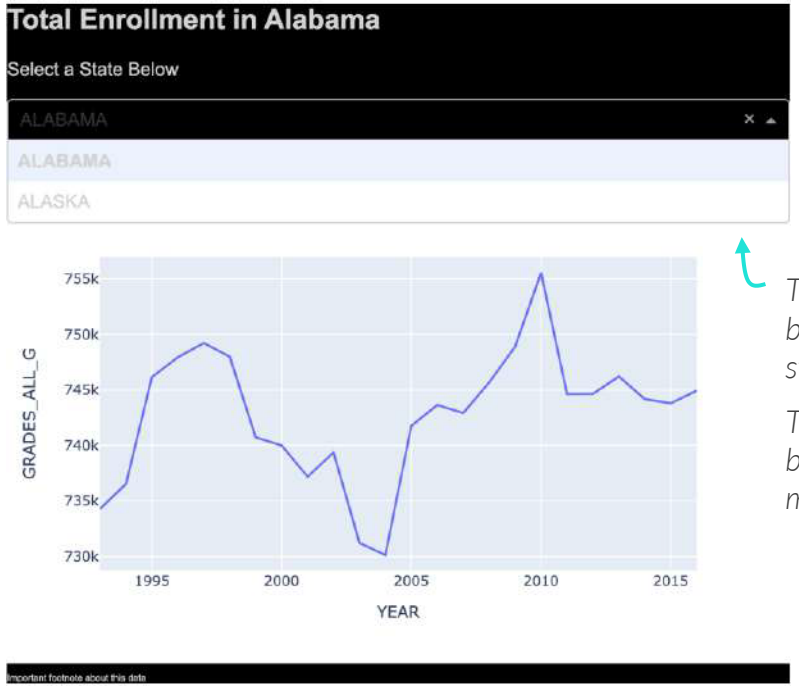
# STYLING DCC COMPONENTS

- Dashboard Basics
- HTML Layouts
- App Styling
- Dash Bootstrap Components

**DCC components** don't inherit parent Div styles, but some styling can be applied

- NOTE: Only some portions of DCC components can be styled without modifying the CSS code

```
my_dash_app_style={
    "color":"lightgrey",
    "backgroundColor": "black",
    "fontFamily": "Arial"
}

app.layout = html.Div(
    style=my_dash_app_style,
    children = [
        html.H2(id="header"),
        html.P("Select a State Below"),
        dcc.Dropdown(
            id="State Dropdown",
            options=["ALABAMA", "ALASKA"],
            value="ALABAMA",
            style={
                "color": "lightgrey",
                "backgroundColor": "black"
            }
        ),
        dcc.Graph(id="Enrollment Line"),
        html.P(
            "Important footnote about this data",
            style={"fontSize": 8}
        )
])
```




*The background is now black, but the expanded options are still white,*

*The font color is light grey, but the placeholder text is much darker*

# STYLING DCC COMPONENTS

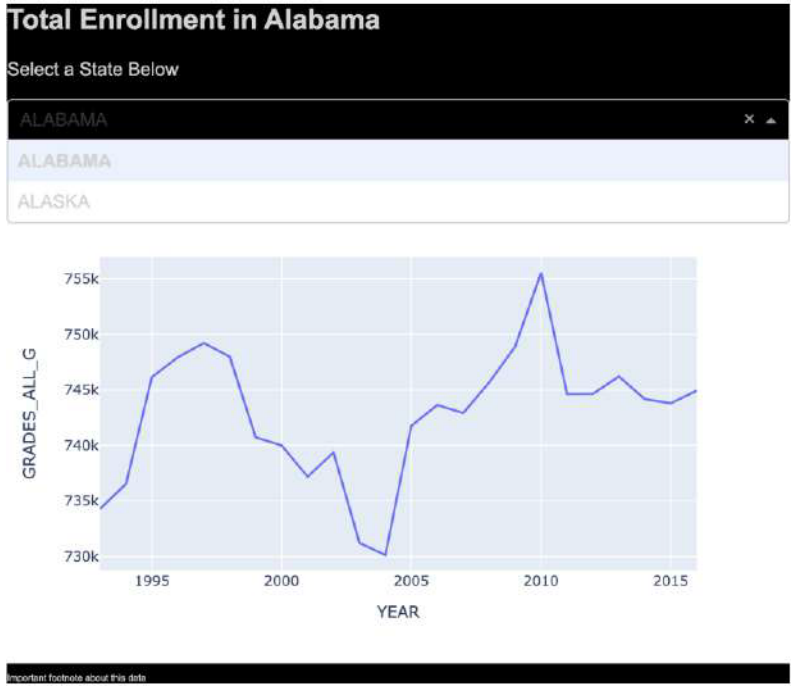


**DCC components** don't inherit parent Div styles, but some styling can be applied

- NOTE: Only some portions of DCC components can be styled without modifying the CSS code

```
my_dash_app_style={
    "color":"lightgrey",
    "backgroundColor": "black",
    "fontFamily": "Arial"
}

app.layout = html.Div(
    style=my_dash_app_style,
    children = [
        html.H2(id="header"),
        html.P("Select a State Below"),
        dcc.Dropdown(
            id="State Dropdown",
            options=["ALABAMA", "ALASKA"],
            value="ALABAMA",
            style=my_dash_app_style
        ),
        dcc.Graph(id="Enrollment Line"),
        html.P(
            "Important footnote about this data",
            style={"fontSize": 8}
        )
])
```

*You can simply call the style you have set*



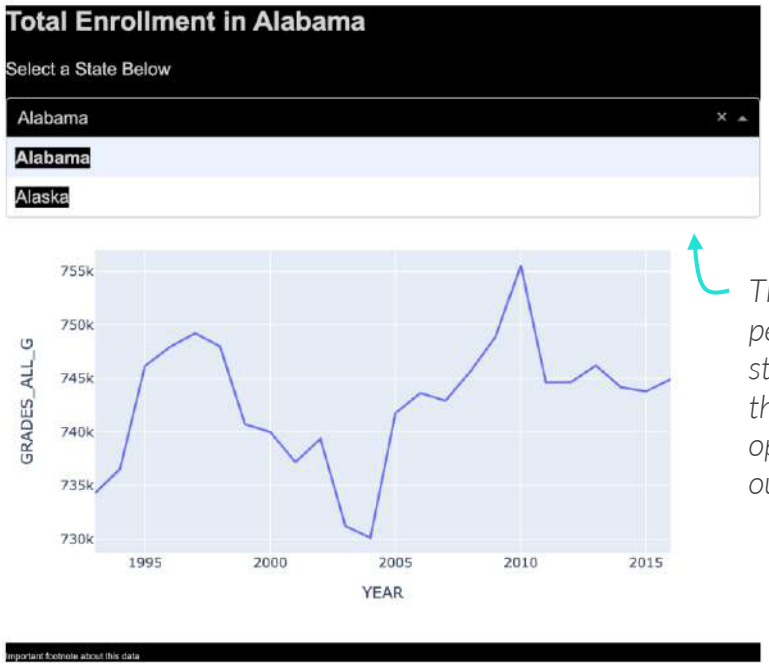
# STYLING DCC COMPONENTS

- Dashboard Basics
- HTML Layouts
- App Styling
- Dash Bootstrap Components

**DCC components** don't inherit parent Div styles, but some styling can be applied

- NOTE: Only some portions of DCC components can be styled without modifying the CSS code

```
my_dash_app_style={
    "color":"lightgrey",
    "backgroundColor": "black",
    "fontFamily": "Arial"
}

app.layout = html.Div(
    style=my_dash_app_style,
    children = [
        html.H2(id="header"),
        html.P("Select a State Below"),
        dcc.Dropdown(
            id="State Dropdown",
            options=[
                {"label": html.Span("Alabama", style=my_dash_app_style),
                 "value": "ALABAMA"},
                {"label": html.Span("Alaska", style=my_dash_app_style),
                 "value": "ALASKA"}
            ],
            value="ALABAMA",
            style={
                "color": "lightgrey",
                "backgroundColor": "black"
            }
        ),
        dcc.Graph(id="Enrollment Line"),
        html.P(
            "Important footnote about this data",
            style={"fontSize": 8}
        )
])
```




*The text colors are perfect after applying styles to each label, but the dropdown menu options are still white outside of the text!*

DCC components are **deceptively difficult to style** because they are defined by Dash's CSS, but there are ways to get around this without needing to know CSS

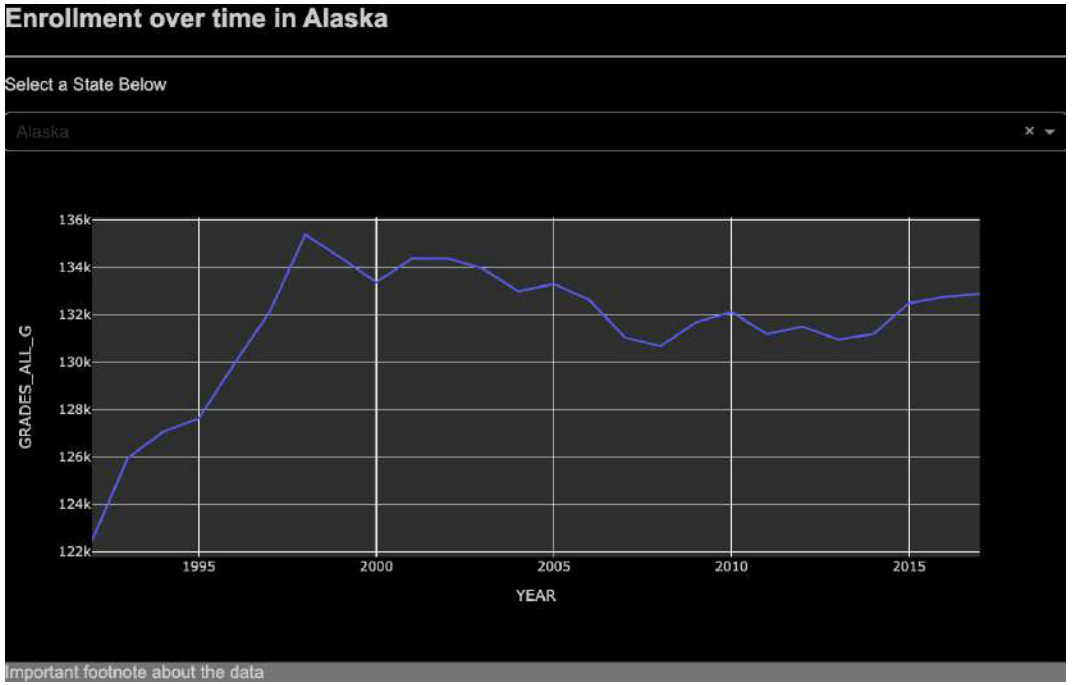
# FIGURE STYLING

- Dashboard Basics
- HTML Layouts
- App Styling
- Dash Bootstrap Components

You need to **style Plotly figures** independently from HTML components

- You can use the **.update_layout()** method to modify the style to match the rest of the app

```
fig=px.line(
    df,
    x="YEAR",
    y="GRADES_ALL_G",
    title="Enrollment over Time"
).update_layout(
    paper_bgcolor="black",
    plot_bgcolor="#343634",
    font_color="lightgrey"
)
```



# **ASSIGNMENT:** APP STYLING

1 **NEW MESSAGE**

*April 2023 ,3*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **Ski Resort Map Styling**

Hey there,

Your HTML looked ok!

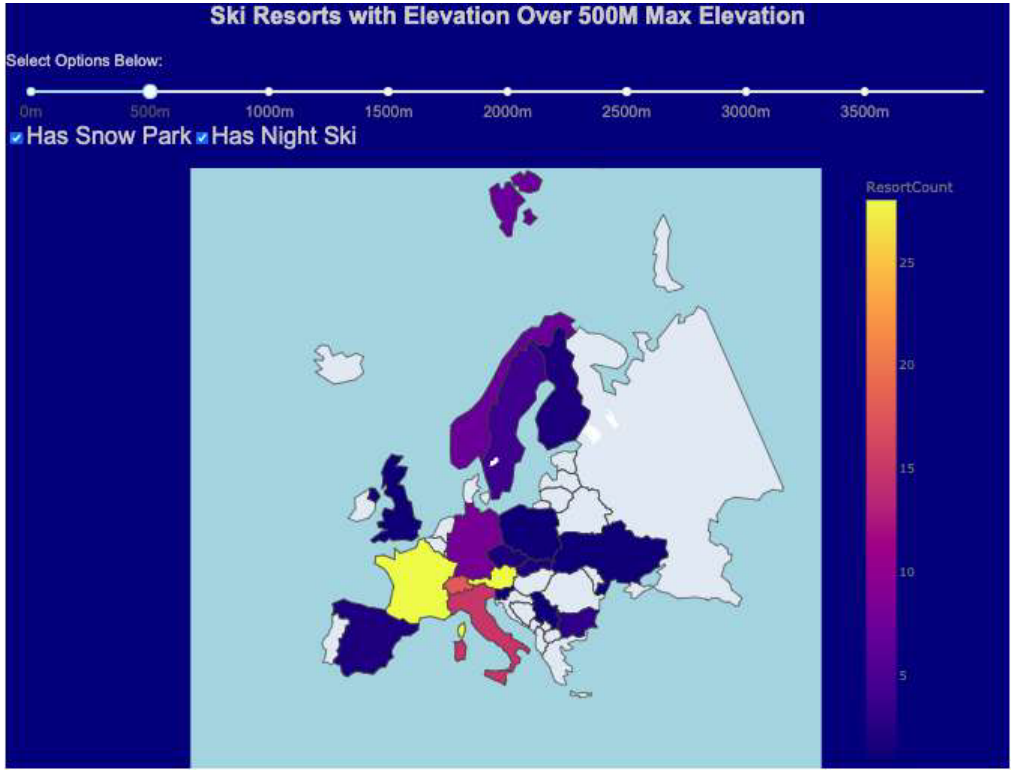
Let's modify the color and text on the Ski Resorts by Country map that we worked on.

Thanks again!

*section05_assignments.ipynb*

Reply Forward

## *Results Preview*



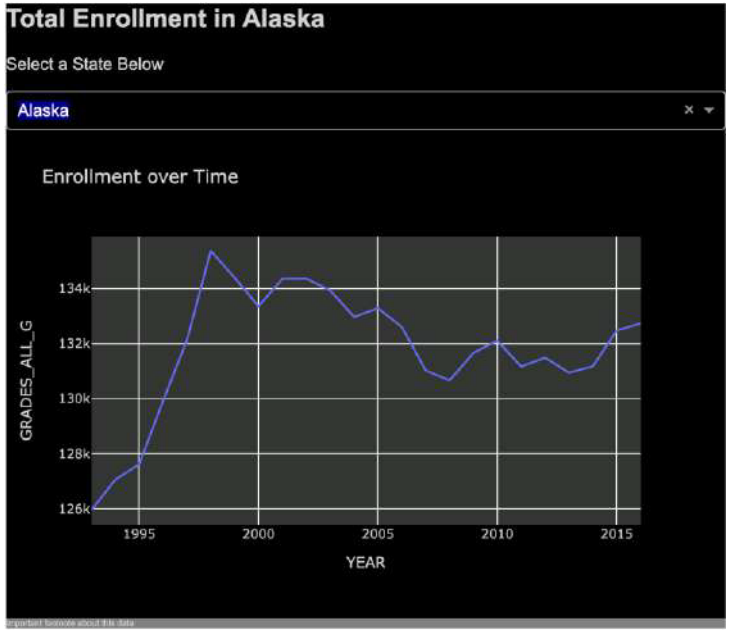
# DASH BOOTSTRAP COMPONENTS



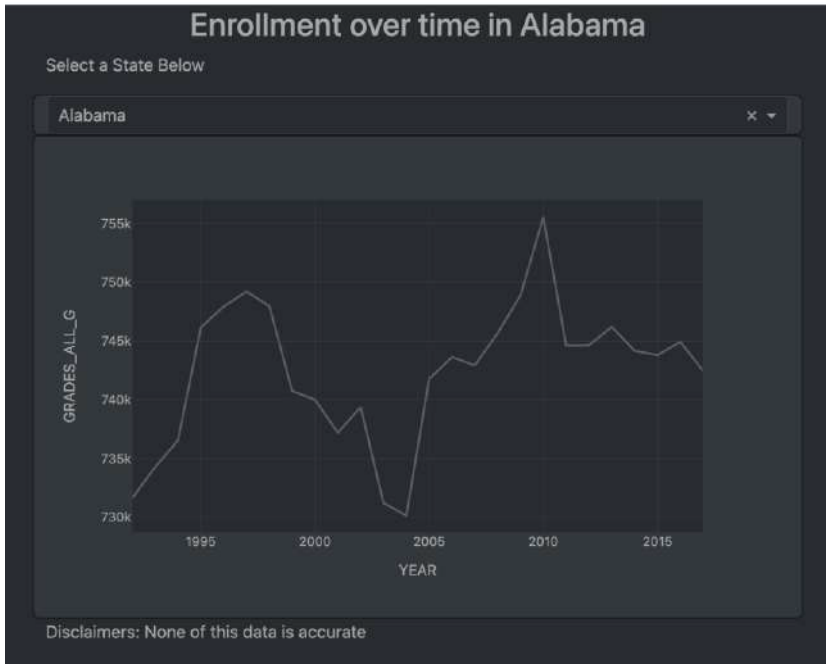
The **Dash Bootstrap Components** (DBC) library offers incredible options for designing polished applications with fewer lines of code

- This includes cohesive styles, built-in padding around components, and a grid-based framework

***Individual component styling***




***Dash Bootstrap Components theme***



# DBC CHEATSHEET

Dashboard Basics

HTML Layouts

App Styling

Dash Bootstrap Components

These are the basic **Dash Bootstrap Components** you need to get started:

| Component | Description |
|---|---|
| dbc.themes | Pre-built CSS style sheets that apply cohesive formatting to your application |
| dbc.Container() | The DBC equivalent of a Div that acts as a style wrapper for sections of the app layout |
| dbc.Card() | A specific type of container for components that adds padding & polish around them |
| dbc.Row() | Represents a horizontal row inside a dbc.Container (or html.Div) |
| dbc.Col() | Represents a vertical column inside a dbc.Row |
| dcc.Tabs() | Creates different tabs for users to navigate between |

**PRO TIP:** Use Dash Bootstrap Components to quickly apply a theme to your dashboard, then tweak individual elements as needed with the methods we've already covered

*For a full list of components, visit:* ***https://dash-bootstrap-components.opensource.faculty.ai/***

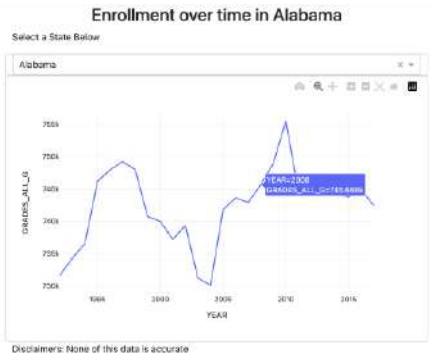
# DASH BOOTSTRAP THEMES



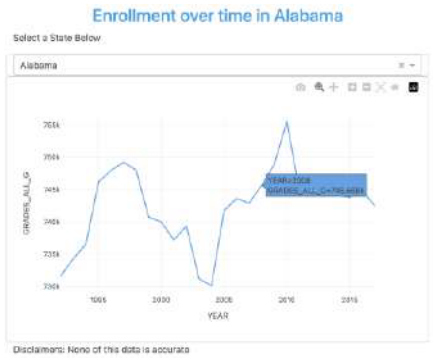
There are 26 **themes** available in the Dash Bootstrap Components library

- Dash(__name__, external_stylesheets=[dbc.themes.THEME_NAME])

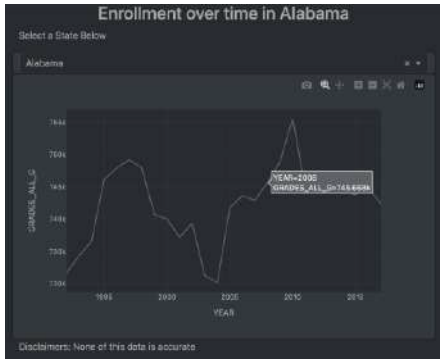
***Bootstrap***



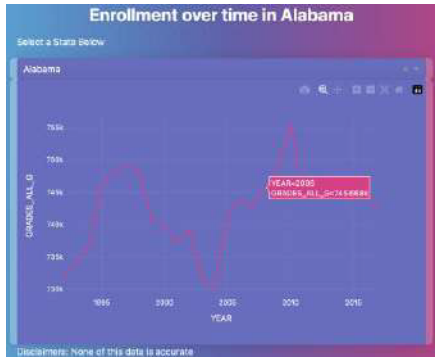

***Cerulean***




***Slate***




***Quartz***




The **DBC Theme Explorer** (linked below) has an app for previewing the different themes available – odds are you will find one that matches your desired aesthetic!

*For an interactive theme viewer, visit:* ***https://dash-bootstrap-components.opensource.faculty.ai/docs/themes/explorer/***

# DASH BOOTSTRAP THEMES

Dashboard Basics
HTML Layouts
App Styling
Dash Bootstrap Components

There are 26 **themes** available in the Dash Bootstrap Components library

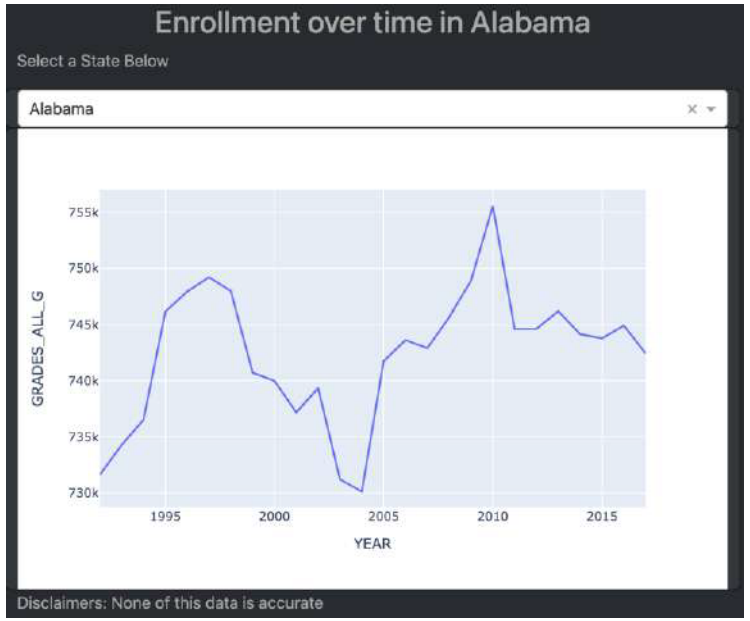
- Dash(__name__, external_stylesheets=[dbc.themes.THEME_NAME])

**EXAMPLE** | *Applying the "slate" theme*

```
app = JupyterDash(__name__, external_stylesheets=[dbc.themes.SLATE])
```

*Simply specify the theme name here!*

*Note that the DCC components & Plotly figures don't adopt the style (for now!)*



# APPLYING THEMES TO FIGURES



You can **apply themes to figures** by using the dash_bootstrap_templates library

- load_figure_template("THEME_NAME")

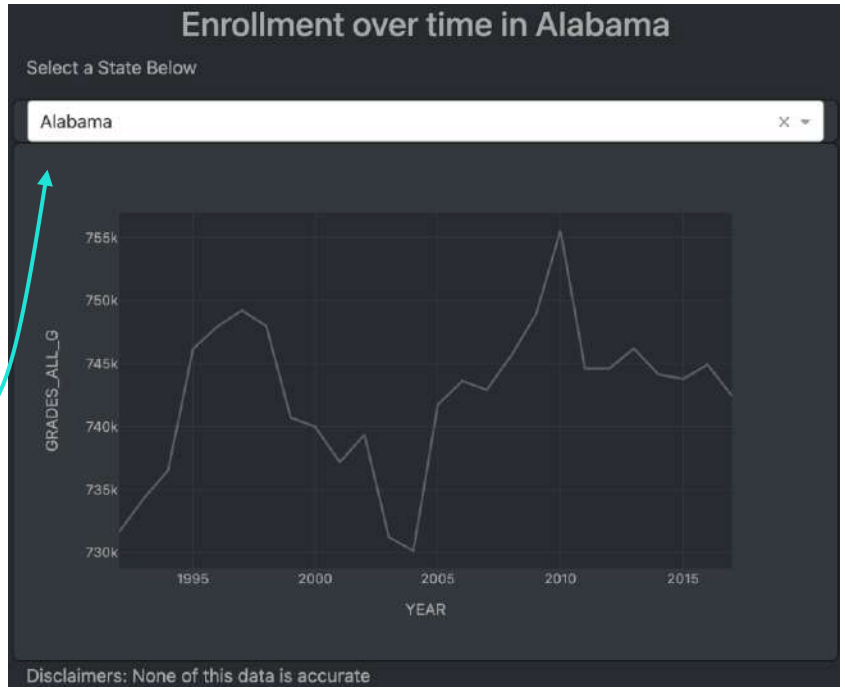
*1) Install the library*

```
from os import sys

!pip install --prefix {sys.prefix} dash_bootstrap_templates
```

*2) Specify the desired theme*

```
from dash_bootstrap_templates import load_figure_template

app = JupyterDash(__name__, external_stylesheets=[dbc.themes.SLATE])

load_figure_template("SLATE")
```

*Now we just need to deal with this pesky dropdown menu!*



# APPLYING THEMES TO DCC COMPONENTS



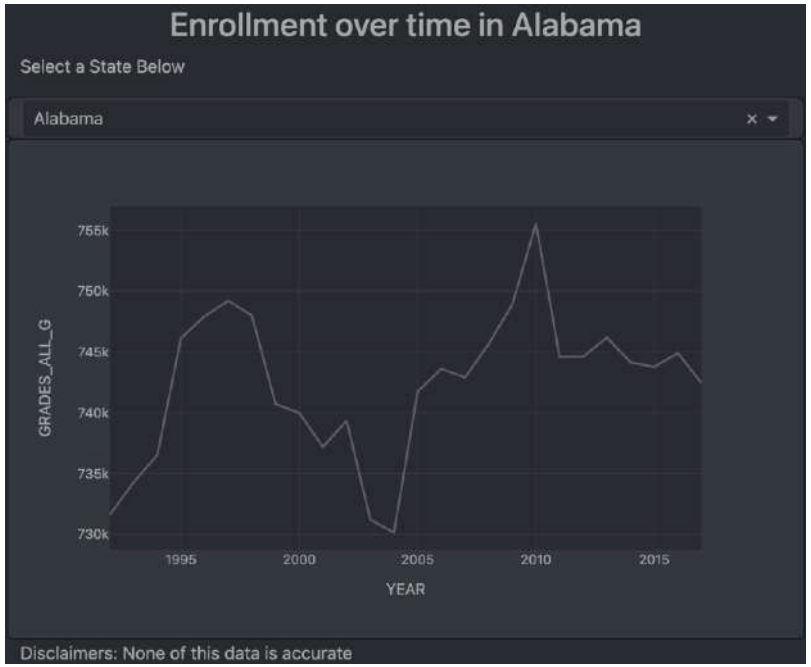
You can **apply themes to DCC components** by using a special link *(see code below)*

- You also need to specify className="dbc" in your dcc component

```
dbc_css = "https://cdn.jsdelivr.net/gh/AnnMarieW/dash-bootstrap-templates/dbc.min.css"

app = JupyterDash(__name__, external_stylesheets=[dbc.themes.SLATE, dbc_css])

load_figure_template("SLATE")

app.layout = dbc.Container(
    children=[
        dbc.Row(

        html.H2(
            id="Header Text",
            style={
                "textAlign": "center",
            })),
        html.P("Select a State Below", id='instructions'),
        dbc.Row(
            dbc.Card(
            dcc.Dropdown(
                options=["Alabama", "Alaska", "Arkansas"],
                value="Alabama",
                id="State Dropdown",
                className="dbc"
            )),
        ),
        dbc.Row(
            dbc.Card(dcc.Graph(id="Revenue Line")),
        ),
        html.P("Disclaimers: None of this data is accurate")
])
```

*Imports external CSS that applies to specified dcc components*

*Applies theme to the component*




You can **apply your own CSS code** with the same steps; Learning CSS is generally well beyond the scope of analyst roles, but it's worth being aware of if you have the web developer skills (or want to learn them!)

# GRID BASED LAYOUTS



The DBC components for rows & columns let you create **grid-based layouts**

- The height of each row is determined by the height of its content
- You can specify the width of each column or let them distribute evenly by default

```
app.layout = dbc.Container([
    dbc.Row(
        dcc.Markdown(
        '''
        ## A Row without columns spans the whole app!
        '''
        ),
    ),
    dbc.Row(
        [
            dbc.Col("Width equal if not specified"),
            dbc.Col("Because total width is 12"),
            dbc.Col("3 Cols each have width 4"),
    ]),
    dbc.Row(
        [
            dbc.Col("Width is 6", width=6),
            dbc.Col("Width is 3", width=3),
            dbc.Col("Width is 3", width=3),
        ]
    )
])
```

# GRID BASED LAYOUTS



The DBC components for rows & columns let you create **grid-based layouts**

- The height of each row is determined by the height of its content
- You can specify the width of each column or let them distribute evenly by default

```
app.layout = dbc.Container([
    dbc.Row(
        dcc.Markdown(
        '''
        ## A Row without columns spans the whole app!
        '''
        ),
    ),
    dbc.Row(
        [
            dbc.Col(dbc.Card("Width equal if not specified")),
            dbc.Col(dbc.Card("Because total width is 12")),
            dbc.Col(dbc.Card("3 Cols each have width 4")),
    ]),
    dbc.Row(
        [
            dbc.Col(dbc.Card("Width is 6"), width=6),
            dbc.Col(dbc.Card("Width is 3"), width=3),
            dbc.Col(dbc.Card("Width is 3"), width=3),
        ]
    )
])
```


**PRO TIP:** Even though it's not necessary, placing objects inside cards helps gives a better visual indication of the "grid" layout

# GRID BASED LAYOUTS

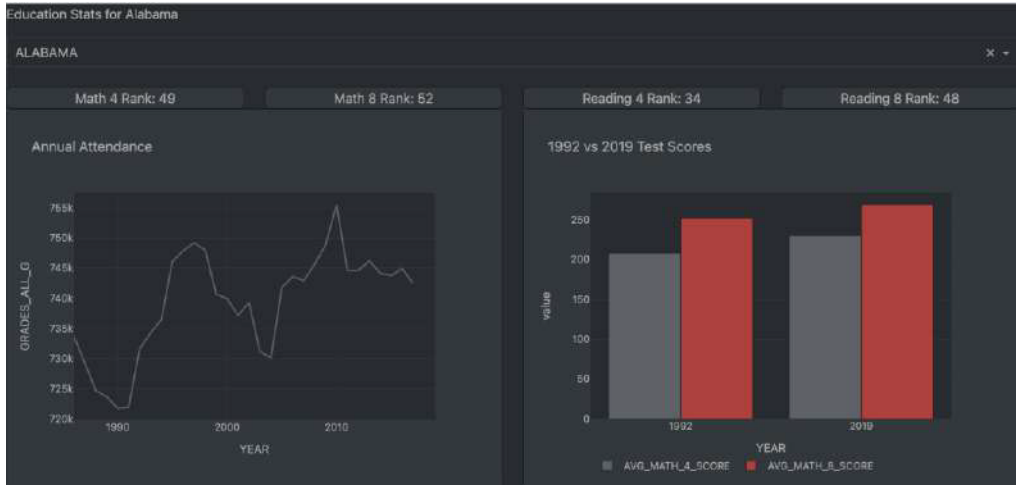


The DBC components for rows & columns let you create **grid-based layouts**

- The height of each row is determined by the height of its content
- You can specify the width of each column or let them distribute evenly by default

**EXAMPLE** *Adding KPI cards and another chart to our dashboard*

```
app.layout = html.Div([
    dcc.Markdown(id="Markdown Title"),
    dbc.Row([
        dcc.Dropdown(
            options=["ALABAMA", "ALASKA", "ARIZONA"],
            value="ALABAMA",
            className="dbc",
            id="State Dropdown"
         ),
        ]),
    html.Br(),
    dbc.Row([
        dbc.Col(dbc.Card(id="KPI 1"), style={"textAlign": "center"}),
        dbc.Col(dbc.Card(id="KPI 2"), style={"textAlign": "center"}),
        dbc.Col(dbc.Card(id="KPI 3"), style={"textAlign": "center"}),
        dbc.Col(dbc.Card(id="KPI 4"), style={"textAlign": "center"}),
        ]),
    dbc.Row([
        dbc.Col(dbc.Card(dcc.Graph(id="Performance Over Time"))),
        dbc.Col(dbc.Card(dcc.Graph(id="Funding Over Time")))
        ])
])
```



# MULTIPLE TABS

- Dashboard Basics
- HTML Layouts
- App Styling
- Dash Bootstrap Components

You can create dashboards with **multiple tabs** with dcc.Tabs() and dcc.Tab()

- Simply specify any number of dcc.Tab() components underneath parent dcc.Tabs()

```
app.layout = html.Div(
    [
        dcc.Tabs(
            id="tabs",
            children=[
                dcc.Tab(
                    label="TAB 1",
                    value="tab1",
                    children = [html.H1("WELCOME TO TAB 1")]
                ),
                dcc.Tab(
                    label="TAB 2",
                    value="tab2",
                    children=[html.H1("WELCOME TO TAB 2")]
                )
            ]
        )
    ]
)
```

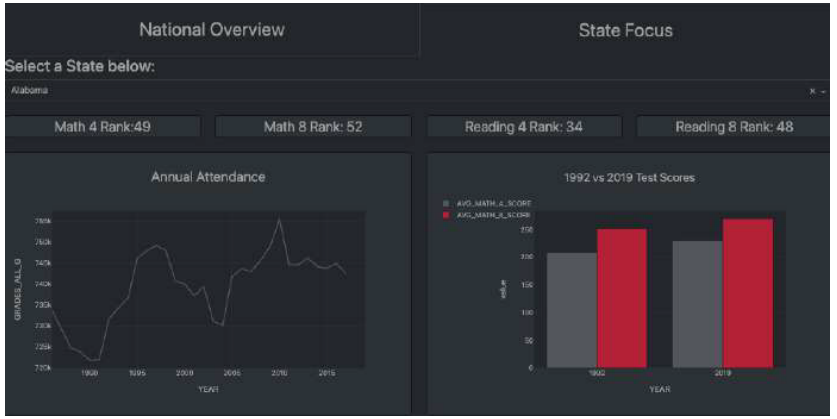

*Note that the components in each tab are specified as a list in the "children" argument*

# MULTIPLE TABS

- Dashboard Basics
- HTML Layouts
- App Styling
- Dash Bootstrap Components

**EXAMPLE** | *Adding a new tab our dashboard*

```
app.layout = html.Div(
    [
        dcc.Tabs(
            id="tabs",
            value="tab-1",
            className="dbc",
            children=[
                dcc.Tab(
                    label="National Overview",
                    value="tab-1",
                    className="dbc",
                    children=[
                        dcc.RadioItems(),
                        dbc.Card(
                            dcc.Graph(id="state_stat_map"),
                        ),

                    ]
                ),
                dcc.Tab(
                    label="State Focus",
                    value="tab-2",
                    className="dbc",
                    children=[
                        dcc.Markdown(id="Markdown Title"),
                        dbc.Row(dcc.Dropdown(id="State Dropdown")),
                        html.Br(),
                        dbc.Row([
                                dbc.Col(dbc.Card(id="KPI 1"), style={"textAlign": "center"}),
                                dbc.Col(dbc.Card(id="KPI 2"), style={"textAlign": "center"}),
                                dbc.Col(dbc.Card(id="KPI 3"), style={"textAlign": "center"}),
                                dbc.Col(dbc.Card(id="KPI 4"), style={"textAlign": "center"}),
                        ]),
                        dbc.Row([
                            dbc.Col(dbc.Card(dcc.Graph(id="Performance Over Time"))),
                            dbc.Col(dbc.Card(dcc.Graph(id="Funding Over Time"))),
                        ]),
                    ],
                ),
            ],
        )
    ]
)
```

# **ASSIGNMENT:** DASH BOOTSTRAP COMPONENTS

1 **NEW MESSAGE**

*April 2023 ,05*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **Updated Layout**

Hey there,

Can we modify the layout of the map application? I'd like for our interactive elements to be in a bar on the left side of our screen, with the map to the right.

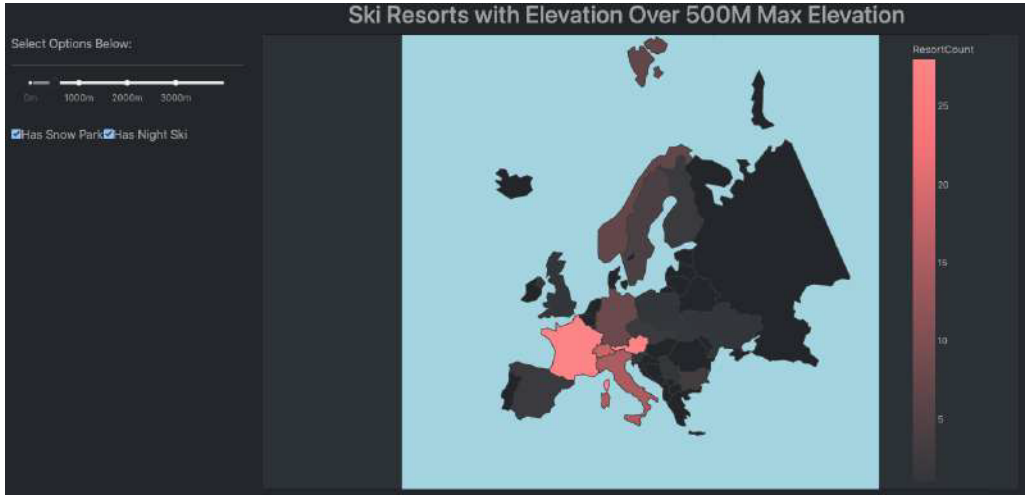
Additionally, apply a dbc theme of your choice to the application... it's likely better than what you created earlier ;)

Thanks again!

*Section05_assignments.ipynb*

Reply Forward

## *Results Preview*



# KEY TAKEAWAYS

**Dashboards** are groups of visuals that help understand data at a glance

- *Adding filters & interactivity lets users explore the data themselves to see what insights emerge*

You can create **dashboard layouts** with HTML, markdown, and DBC

- *Dash Bootstrap Components (DBC) let you easily create grid-based layouts with multiple tabs*
- *The focus of the layout should be to add cohesion to its visuals and interactive elements*

**Themes** let you easily apply pre-defined styles to your dashboards

- *You can still apply custom styles to individual components if needed*

# ADVANCED TOPICS

# ADVANCED TOPICS

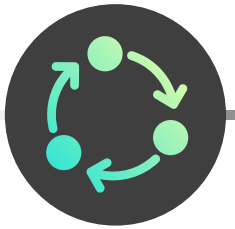
In this section we'll cover **advanced topics** like chained & conditional callback functions, cross-filtering, debug mode, data table outputs, and app deployment options

## TOPICS WE'LL COVER:

- Data Tables
- Advanced Callbacks
- App Deployment

## GOALS FOR THIS SECTION:

- Embed data tables that users can sort, filter, and export into Dash applications
- Write advanced callback function structures that take user interactivity to the next level
- Learn to deploy an application to an online server that stakeholders can access



# DATA TABLES

Data Tables

Advanced Callbacks

App Deployment

You can embed **data tables** into your dashboards with the dash_table module

- dash_table.DataTable(columns, data)

```
education.head()
```

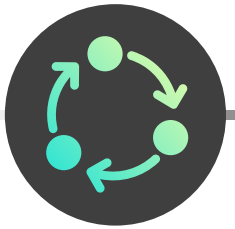
| | STATE | YEAR | ENROLL | TOTAL_REVENUE | FEDERAL_REVENUE | STATE_REVENUE |
|---|---|---|---|---|---|---|
| 0 | ALABAMA | 1992 | NaN | 2678885.0 | 304177.0 | 1659028.0 |
| 1 | ALASKA | 1992 | NaN | 1049591.0 | 106780.0 | 720711.0 |
| 2 | ARIZONA | 1992 | NaN | 3258079.0 | 297888.0 | 1369815.0 |
| 3 | ARKANSAS | 1992 | NaN | 1711959.0 | 178571.0 | 958785.0 |
| 4 | CALIFORNIA | 1992 | NaN | 26260025.0 | 2072470.0 | 16546514.0 |

```
from dash import dash_table

app.layout = html.Div(
    dash_table.DataTable(
        columns=[{"name": i, "id": i} for i in education.columns],
        data=education.to_dict("records"),
    )
```

*This creates a table with the data from the "education" DataFrame with the column names as headers*

| STATE | YEAR | ENROLL | TOTAL_REVENUE | FEDERAL_REVENUE | STATE_REVENUE |
|---|---|---|---|---|---|
| ALABAMA | 1992 | | 2678885 | 304177 | 1659028 |
| ALASKA | 1992 | | 1049591 | 106780 | 720711 |
| ARIZONA | 1992 | | 3258079 | 297888 | 1369815 |
| ARKANSAS | 1992 | | 1711959 | 178571 | 958785 |
| CALIFORNIA | 1992 | | 26260025 | 2072470 | 16546514 |

# DATA TABLES



Data Tables

Advanced Callbacks

App Deployment

You can use additional arguments to let users **sort**, **filter**, and **export** the table

- dash_table.DataTable(columns, data, filter_action, sort_action, export_format)

*The "native" options for sorting and filtering work quite well*

*You can also export to "xlsx"*

```
from dash import dash_table

app.layout = html.Div(
   dash_table.DataTable(
        columns=[{"name": i, "id": i} for i in education.columns],
        data=education.to_dict("records"),
        filter_action="native",
        sort_action="native",
        export_format="csv"
   )
```

*This was sorted by TOTAL_REVENUE with the STATE filtered to "CALIFORNIA"*

Export

| STATE | YEAR | ENROLL | TOTAL_REVENUE | FEDERAL_REVENUE | STATE_REVENUE |
|---|---|---|---|---|---|
| CALIFORN | | | | | |
| CALIFORNIA | 2016 | 6217031 | 89217262 | 7709079 | 50904567 |
| CALIFORNIA | 2015 | 6226523 | 78248042 | 7556365 | 42360470 |
| CALIFORNIA | 2008 | 6258421 | 74626928 | 7227456 | 43187637 |
| CALIFORNIA | 2009 | 6234155 | 73958896 | 9745250 | 40084244 |
| CALIFORNIA | 2007 | 6288686 | 72516936 | 7200298 | 42333637 |

The **default style for tables isn't the best**, but they can be styled like any other HTML object (DBC has a table implementation with better aesthetics, but less functionality)

# **ASSIGNMENT:** DATA TABLES

**NEW MESSAGE**

*April 2024 ,11*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **Table Output**

Hey there,

Can you create a simple app that allows users to select country from a dropdown, uses our elevation slider, and returns a table of the resorts in that country with a highest point greater than the elevation specified?

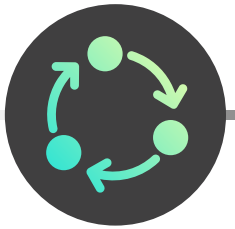
Make sure the table can be sorted, filtered, and exported – some of our clients want to browse and analyze our data themselves. If you're up for it, try styling it!

Thanks!

*Section06_assignments.ipynb*

Reply Forward

## *Results Preview*



# ADVANCED CALLBACK FUNCTIONS

Data Tables

Advanced Callbacks

App Deployment

These are some **advanced callback function** concepts:

| | |
|---|---|
| **Conditional Callbacks** | *Return different components to the front end based on some conditional logic* |
| **Chained Callbacks** | *Modify the options of an interactive element based on the option selected in another* |
| **Cross-Filtering Callbacks** | *Filter figures in the app based on the data selected (or hovered over) in another figure* |
| **Manual Callbacks** | *Prevent app updates until the user initiates the callback process, or "applies changes"* |
| **Periodic Callbacks** | *Updates the app by initiating the callback process at fixed time intervals* |

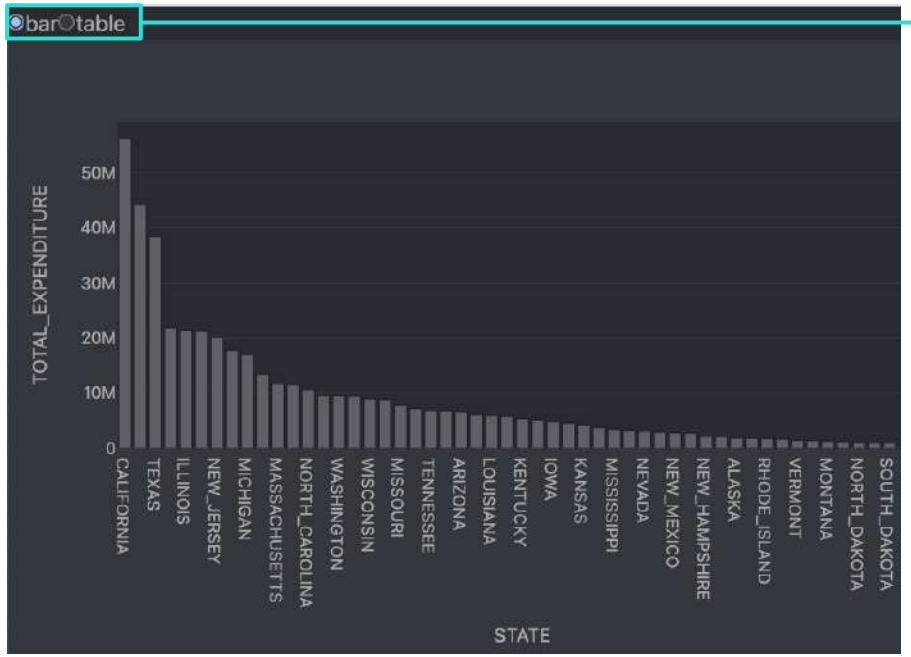
**Dash is extremely powerful and flexible**. These concepts are worth being aware of, but good dashboards are often quite simple, so you may never use most of these in practice

# CONDITIONAL CALLBACKS

**Conditional callbacks** use logical tests to return different types of components to the front end of your application

- Data Tables
- Advanced Callbacks
- App Deployment

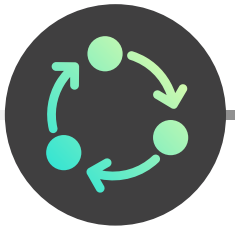
***dcc.Graph*** *(default)*




***dash_table.DataTable***

bar table

| STATE | TOTAL_EXPENDITURE |
|---|---|
| CALIFORNIA | 56247851.32 |
| NEW_YORK | 44201162.8 |
| TEXAS | 38347089.88 |
| PENNSYLVANIA | 21813791.52 |
| ILLINOIS | 21444064.84 |
| FLORIDA | 21287605.24 |
| NEW_JERSEY | 20051592.24 |
| OHIO | 17775446.24 |
| MICHIGAN | 17026245.84 |
| GEORGIA | 13332004.36 |
| MASSACHUSETTS | 11740076.32 |
| VIRGINIA | 11456358.48 |
| NORTH_CAROLINA | 10512102.72 |
| MARYLAND | 9574154.72 |
| WASHINGTON | 9481863.84 |
| INDIANA | 9406665.08 |



# CONDITIONAL CALLBACKS

**Conditional callbacks** use logical tests to return different types of components to the front end of your application

- Data Tables
- Advanced Callbacks
- App Deployment

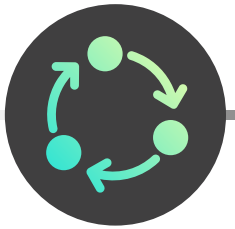
```
app.layout = html.Div([
    dcc.RadioItems(
        id="OutputPicker",
        options=["bar", "table"],
        value="bar"
    ),
    html.Div(id="Output Div")
])

@app.callback(Output("Output Div", "children"), Input("OutputPicker", "value"))
def output_generator(output_style):
    if output_style == "bar":
        output = dcc.Graph(figure=px.bar(df, x="STATE", y="TOTAL_EXPENDITURE"))
    else:
        output = dash_table.DataTable(
            columns=[{"name": i, "id": i} for i in df.columns],
            data=df.to_dict("records"))

    return output
```

*This gives the user two options: "bar" or table"*

*This empty Div will house a component output by the "conditional" callback function*

*If the user selects "bar", the callback returns a dcc.Graph() component with a bar chart*

*Otherwise, if the user selects "table", the callback returns a DataTable object*



# CHAINED CALLBACKS

Data Tables

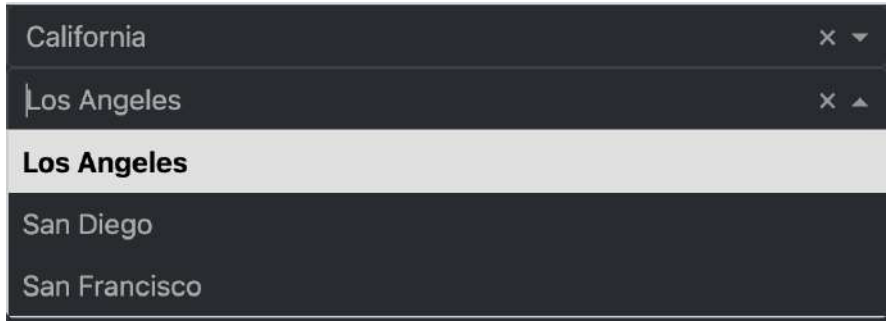
Advanced Callbacks

App Deployment

**Chained callbacks** use the output of a callback function as the input of another

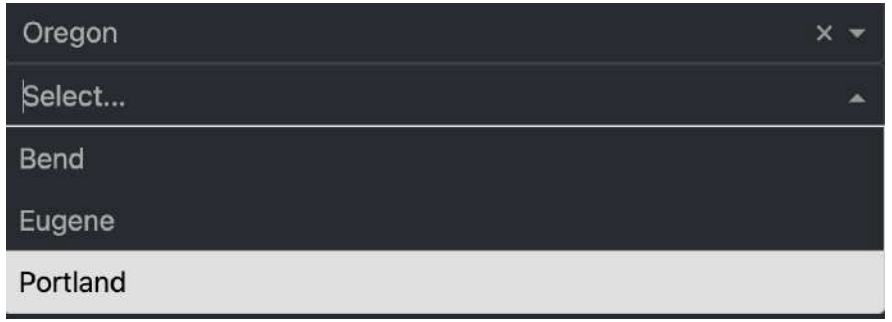
- This is typically used to modify the options of an interactive element based on the option selected in another *(like dependent dropdown lists)*

***"California" selected*** *(default)*

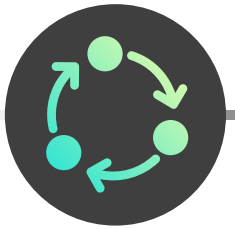



*The second dropdown has cities in California as options*

***"Oregon" selected***




*The second dropdown has cities in Oregon as options*



# CHAINED CALLBACKS

Data Tables

Advanced Callbacks

App Deployment

**Chained callbacks** use the output of a callback function as the input of another

- This is typically used to modify the options of an interactive element based on the option selected in another *(like dependent dropdown lists)*

```
states_cities = {
    "California": ["Los Angeles", "San Diego", "San Francisco"],
    "Oregon": ["Bend", "Eugene", "Portland"],
    "Washington": ["Olympia", "Spokane", "Seattle"],
}

app.layout= html.Div([
    dcc.Dropdown(
        id="state-dropdown",
        options=["California", "Oregon", "Washington"],
        value="California",
        className="dbc"
    ),
    dcc.Dropdown(id="city-dropdown", value="Los Angeles", className="dbc"),
    dbc.Card(id="text-output")
])

@app.callback(
    Output("city-dropdown", "options"),
    Input("state-dropdown", "value"))
def set_metrics_options(selected_state):
    return states_cities[selected_state]

@app.callback(
    Output("text-output", "children"),
    Input("city-dropdown", "value"))
def plot_bar(city):
    if not city:
        raise PreventUpdate
    return f"You have selected {city}!"
```

*This dictionary sets the dependencies, with the keys being the options for the first dropdown, and the values are their corresponding options for the second dropdown*

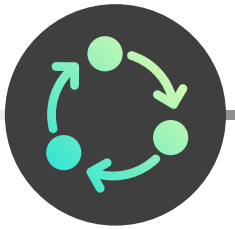
*This is the first dropdown with the dictionary keys*

*This is the second dropdown with no options specified (yet!)*

*The first callback takes the value from the first dropdown as the input, and uses it as the key to return its values from the "states_cities" dictionary as options for the second dropdown*

*The second callback takes the value from the second dropdown and returns it to the dbc.Card component*

*(you can use it to filter charts instead!)*



# PRO TIP: DEBUG MODE

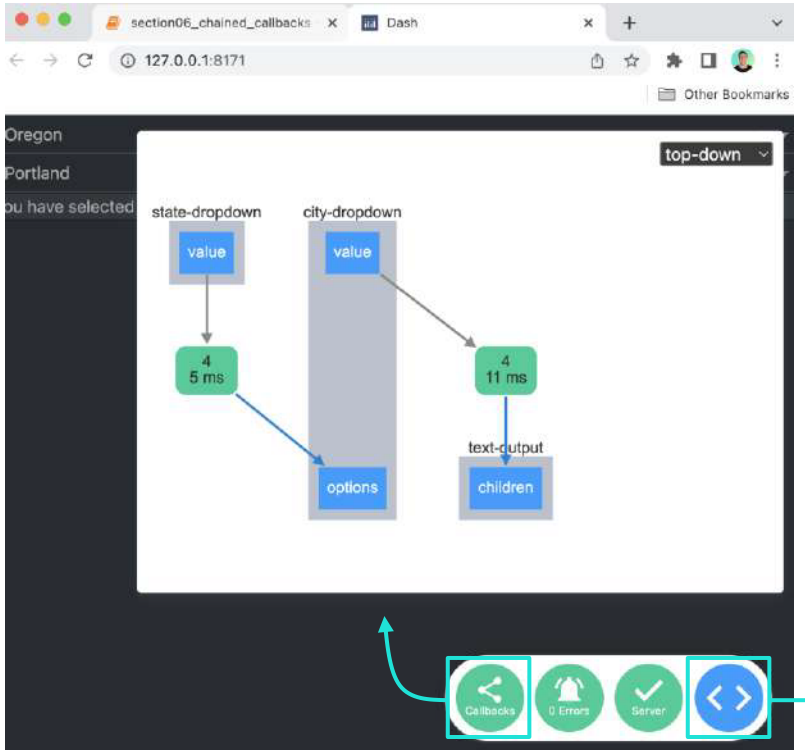
Data Tables

Advanced Callbacks

App Deployment

You can turn on **debug mode** when running an app to look at the structure of your callbacks, diagnose errors, and identify components causing poor performance

- **app.run_server(debug_mode=True)**




*This **callback diagram** shows your app's components, their properties, and their callback relationships*

*(and how long the callbacks take to run!)*

*This expands the **debug menu** shown here*

# PRO TIP: DEBUG MODE

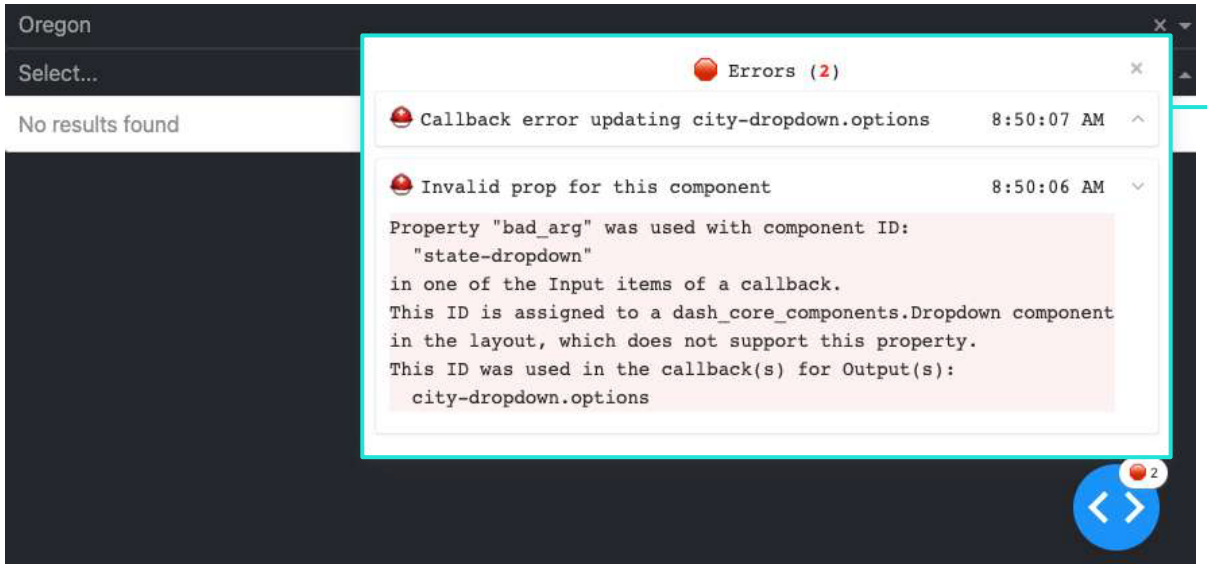
Data Tables

Advanced Callbacks

App Deployment

You can turn on **debug mode** when running an app to look at the structure of your callbacks, diagnose errors, and identify components causing poor performance

- **app.run_server(debug_mode=True)**




*The error messages in debug mode are generated when interacting with the app, letting you pinpoint the cause of errors*

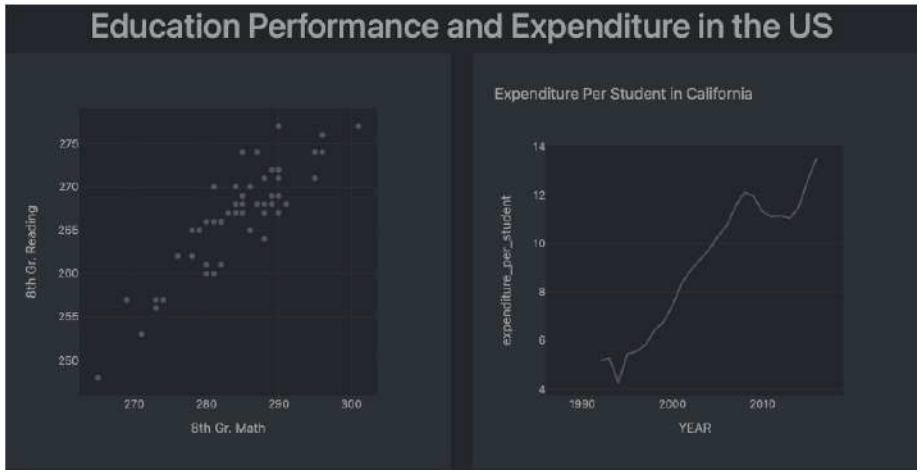
**PRO TIP:** Use debug mode when building the app, and turn it off before deployment

# CROSS-FILTERING

**Cross-filtering** uses chart selections as inputs to callbacks that filter other charts

- Data Tables
- Advanced Callbacks
- App Deployment

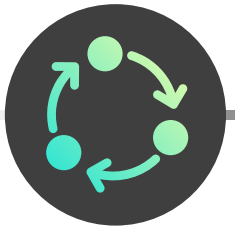
***Load state*** *(default)*




*The app loads by plotting "California" in the second chart*

***Hover state***


*As you hover over states in the first chart, the second chart updates*



# CROSS-FILTERING

Data Tables

Advanced Callbacks

App Deployment

**Cross-filtering** uses chart selections as inputs to callbacks that filter other charts

```
app.layout = html.Div([
    dbc.Row(
        html.H1("Education Performance and Expenditure in the US",
                style={"text-align": "center"})
    ),
    dbc.Row([
        dbc.Col(
            dcc.Graph(
                id="cross-filter-scatter",
                figure=px.scatter(
                    education.query("YEAR == 2013"),
                    x="8th Gr. Math",
                    y="8th Gr. Reading",
                    hover_name="STATE",
                    custom_data=["STATE"]
                ),
                hoverData={'points': [{'customdata': ['CALIFORNIA']}]}
            )
        ),
        dbc.Col(dcc.Graph(id="x-line"))
    ])
])
@app.callback(
    Output("x-line", "figure"),
    Input("cross-filter-scatter", "hoverData"))
def update_line(hoverData):
    state_name = hoverData["points"][0]["customdata"][0]
    df=education.query("STATE == @state_name")

    fig = px.line(
        df,
        x="YEAR",
        y="expenditure_per_student",
        title=f"Expenditure Per Student in {state_name.title()}"
    ).update_xaxes(showgrid=False)

    return fig
```

*The **custom_data** argument lets you specify which column(s) to grab values from when the user hovers over a datapoint*

*The **hoverData** argument in dcc.Graph() lets you specify a starting value to pass through before the user hovers over a datapoint*

*The callback function input is the hoverData from the scatterplot*

*This grabs the "STATE" value from the hoverData dictionary passed through the callback and uses it to filter the DataFrame*

*This plots a line chart with the "cross-filtered" DataFrame!*

# PRO TIP: MANUAL CALLBACKS

Data Tables

Advanced Callbacks

App Deployment

You can add a button component to your app that **run callbacks manually**, allowing users to make multiple selections before applying any updates

- This helps if your application has long processing times *(common in ML models)*

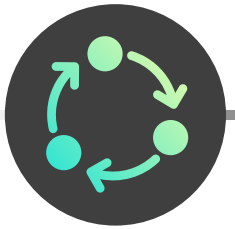
***Load state*** *(no errors!)*


*The first chart relies on inputs from the radio buttons on the left, but the callback function won't fire until the user clicks "Submit"*

*(the second chart is displaying default values, and is unrelated to the radio buttons on the left)*

***After manual callback***


*The first chart now displays values based on the user selections, and is also filtering the second chart by the selected data point*



# PRO TIP: MANUAL CALLBACKS

Data Tables

Advanced Callbacks

App Deployment

You can add a button component to your app that **run callbacks manually**, allowing users to make multiple selections before applying any updates

- This helps if your application has long processing times *(common in ML models)*

```
app.layout = html.Div([
    dbc.Row(html.H1("Education Performance and Expenditure in the US", style={"text-align": "center"})),
    dbc.Row([
       dbc.Col([
           dbc.Card([
               dcc.Markdown("Select an X Column"),
               dcc.RadioItems(
                   id="score-radio",
                   options=["4th Gr. Math", "8th Gr. Math", "4th Gr. Reading", "8th Gr. Reading"],
                   value="8th Gr. Math"
               ),
               html.Hr(),
               dcc.Markdown("Select a Y Column"),
               dcc.RadioItems(
                    id="score-radio2",
                    options=["4th Gr. Math", "8th Gr. Math", "4th Gr. Reading", "8th Gr. Reading"],
                    value="8th Gr. Reading"
               ),
               html.Button("Submit", id="submit-button", n_clicks=0)])
       ], width=2),
        dbc.Col(dcc.Graph(id="cross-filter-scatter",
                          hoverData={'points': [{'customdata': ['CALIFORNIA']}]}), width=5),
        dbc.Col(dcc.Graph(id="x-line"), width=5)
    ])
])

@app.callback(
    Output("cross-filter-scatter", "figure"),
    Input("score-radio", "value"),
    Input("score-radio2", "value"),
    Input("submit-button", "n_clicks")
)
def score_scatter(x, y, n_clicks):
    if not n_clicks:
        raise PreventUpdate
```

*This adds the "Submit" button and sets the "n_clicks" property equal to 0*

*The "n_clicks" property from the button gets passed as an input into the callback*

*If the button has not been clicked, then the app doesn't update (n_clicks=None)*

# **PRO TIP:** PERIODIC CALLBACKS

Data Tables

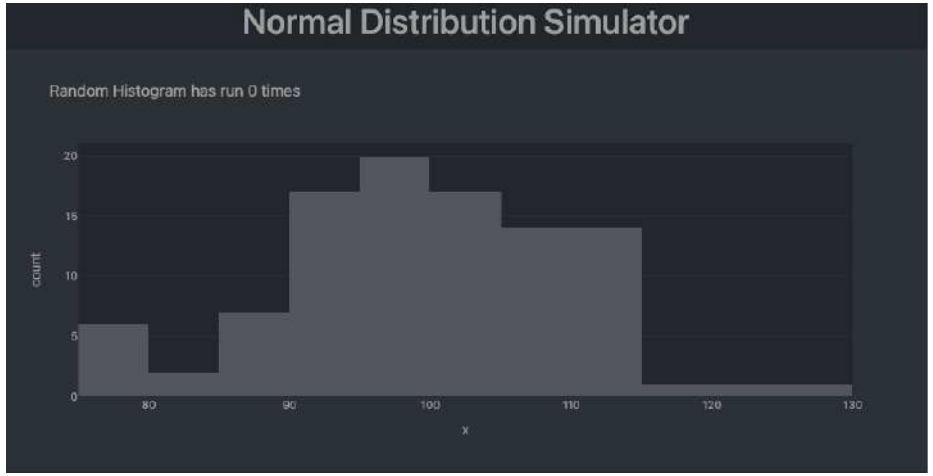
Advanced Callbacks

App Deployment

You can schedule **periodic callbacks** with the dcc.Interval() component

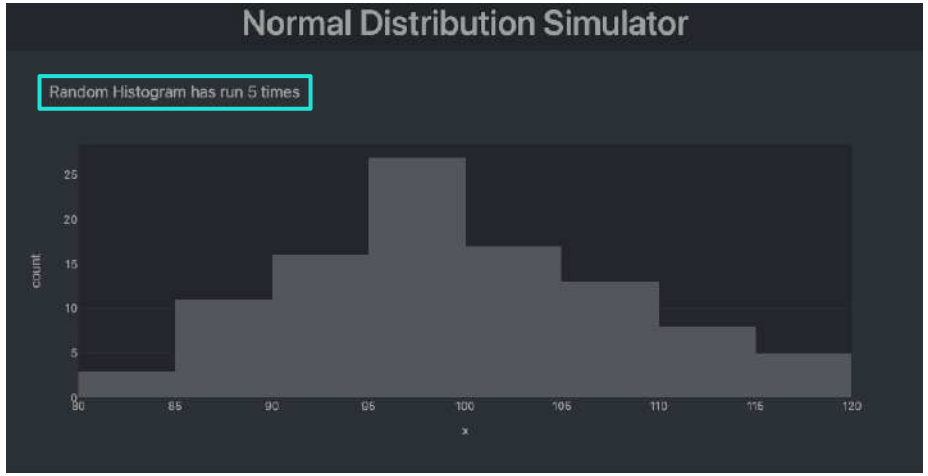
- This lets you update your app automatically without user input
- dcc.Interval(interval, n_intervals)

***Load state*** *(0 seconds)*



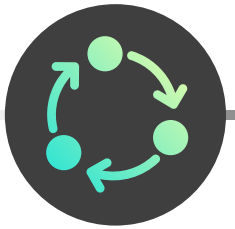

*This app is set to "refresh" every 2 seconds*

***10 seconds later***




*In 10 seconds, 5 callbacks have triggered every 2 seconds*

This can allow for **real-time updates** if your application is connected to a database or API, so instead of firing a random number generator, new data can be queried and appended to your DataFrame via pd.read_sql()



# PRO TIP: PERIODIC CALLBACKS

Data Tables

Advanced Callbacks

App Deployment

You can schedule **periodic callbacks** with the dcc.Interval() component

- This lets you update your app automatically without user input
- dcc.Interval(interval, n_intervals)

```
from numpy.random import default_rng
rng=default_rng(2023)

app.layout = html.Div([
    dbc.Row(html.H1("Normal Distribution Simulator", style={"text-align": "center"})),
    dbc.Row(dbc.Col(dcc.Graph(id="random-data-scatter"))),
    dcc.Interval(id="refresh-data-interval", interval=2000, n_intervals=0)
])

@app.callback(
    Output("random-data-scatter", "figure"),
    Input("refresh-data-interval", "n_intervals")
)

def score_scatter(n_intervals):
    mean, stddev = 100, 10
    fig = px.histogram(
        x=rng.normal(mean, stddev, size=100),
        title=f"Random Histogram has run {n_intervals} times"
    )
    return fig
```

*This will trigger every 2,000 milliseconds (2 seconds)*

*This takes "n_intervals" as the input, fires the callback function, and increments "n_intervals" by 1*

*This creates a histogram with randomized normal data, and show how many times it has run*

*(you don't need to use n_intervals in the function)*

# **ASSIGNMENT**: ADVANCED CALLBACKS

1 **NEW MESSAGE**

*April 2024 ,20*

From: **Leonard Lift** *(Ski Trip Concierge)*

Subject: **Updated Table App**

Hey there,

Can we update the table application we created earlier to include a few more features?

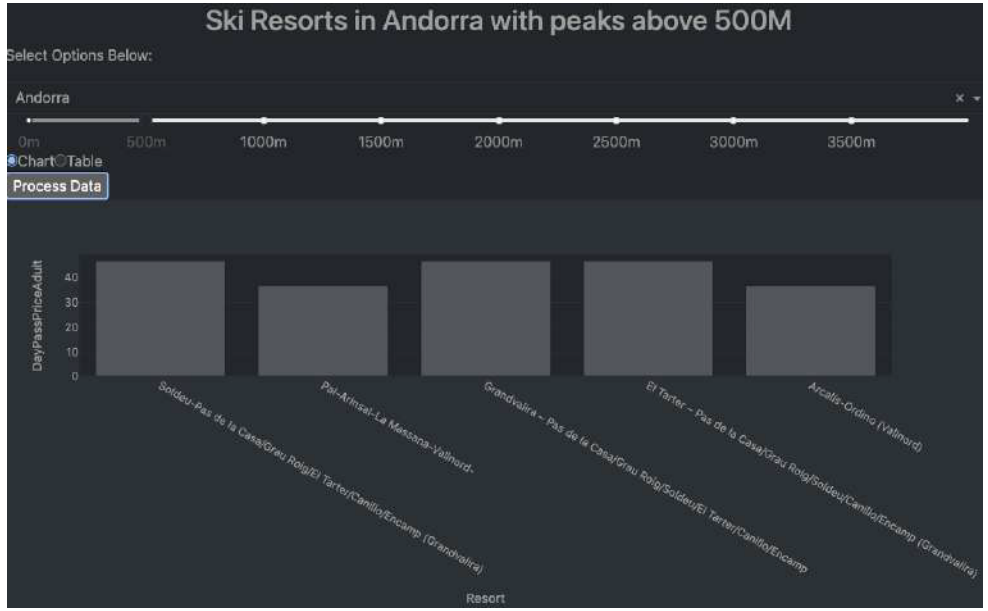
I'd like the user to be able to select whether a bar chart or a table is returned, and I don't want the app to run until the user clicks a button to confirm their choices.

Thanks!

*sectionO6_assignments.ipynb*

Reply Forward

## *Results Preview*



# DEPLOYING YOUR APP

Data Tables

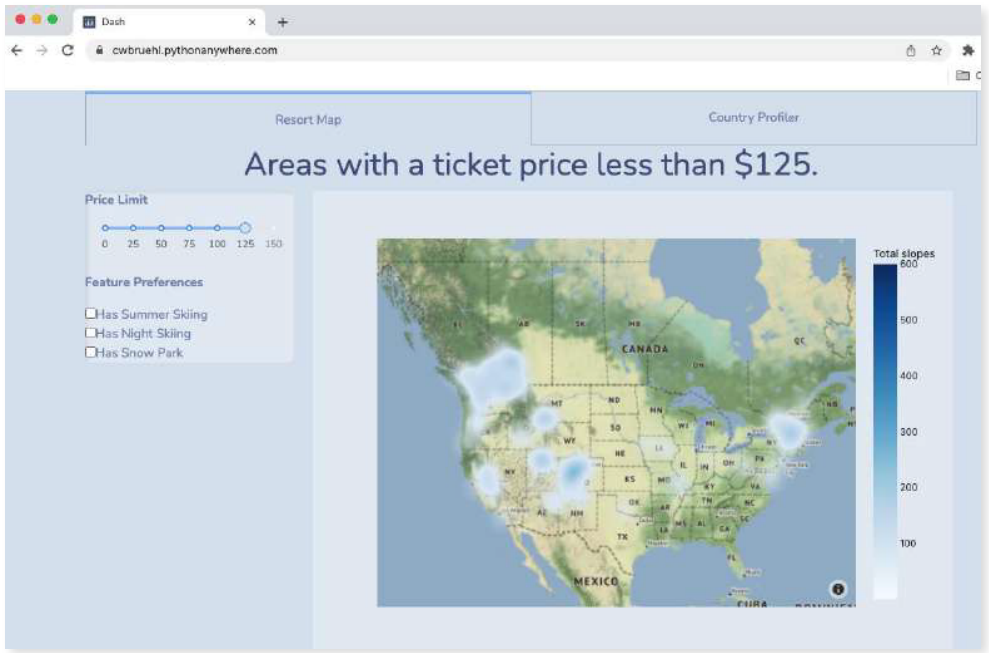
Advanced Callbacks

App Deployment

## **Deploying your app** requires access to a server

- For professional projects, you can use cloud services like AWS and Azure, or Dash Enterprise
- For personal projects, you can use PythonAnywhere or Heroku

*This is hosted on PythonAnywhere*




*Check out my deployed app at:* ***https://cwbruehl.pythonanywhere.com/***

# PYTHONANYWHERE

Data Tables

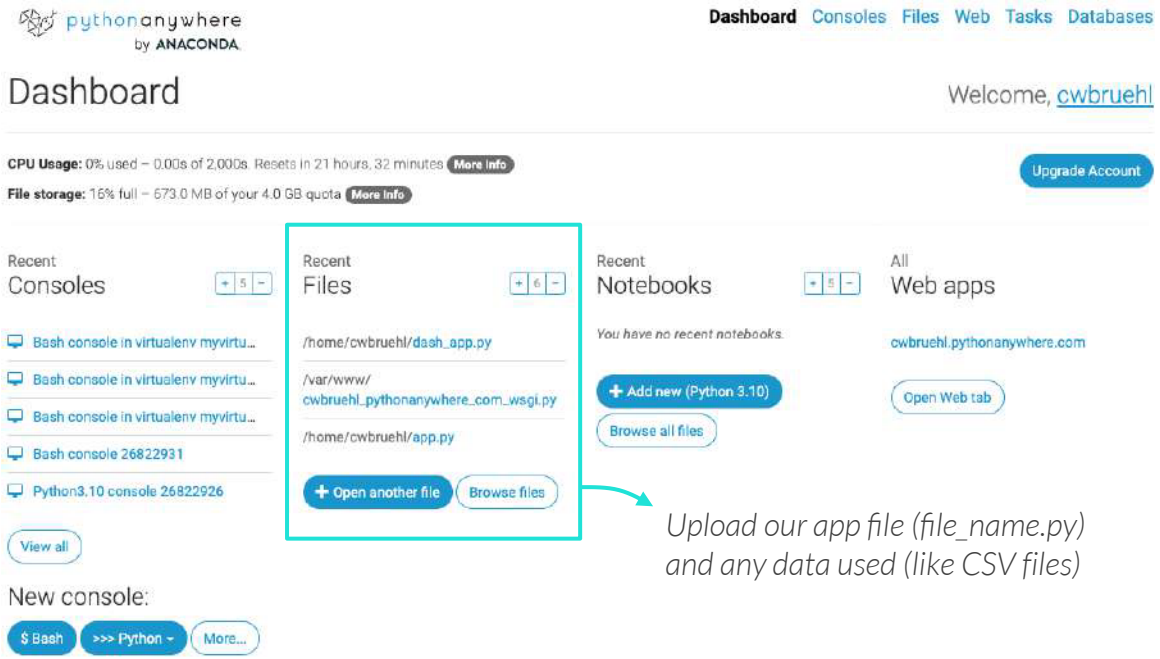
Advanced Callbacks

App Deployment

You can deploy your app on **PythonAnywhere** for free by following these steps:

1) Go to **pythonanywhere.com/pricing** and click Create a Beginner account

2) Register, then **upload your app** and **data files**



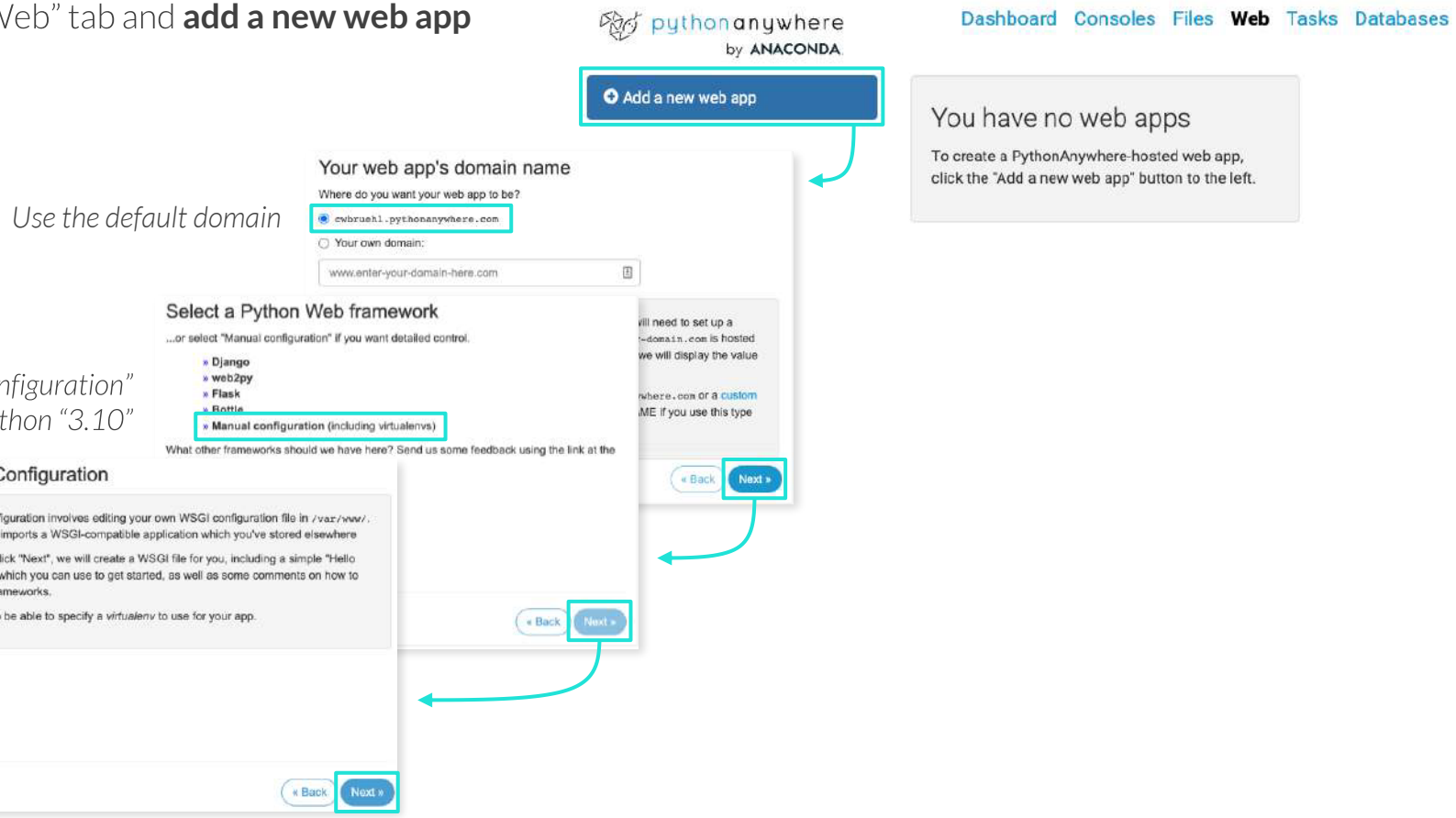

*Upload our app file (file_name.py) and any data used (like CSV files)*

# PYTHONANYWHERE



You can deploy your app on **PythonAnywhere** for free by following these steps:

3) Go to the "Web" tab and **add a new web app**

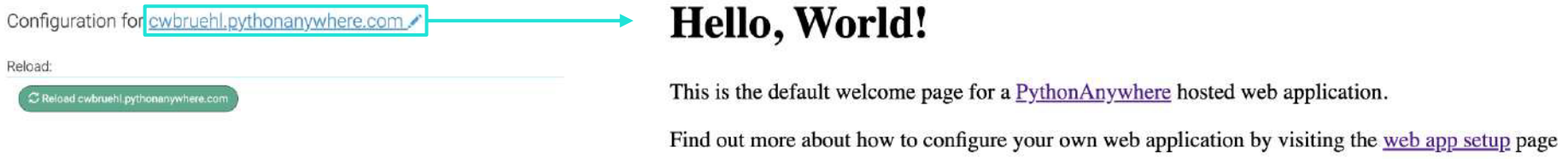


# PYTHONANYWHERE

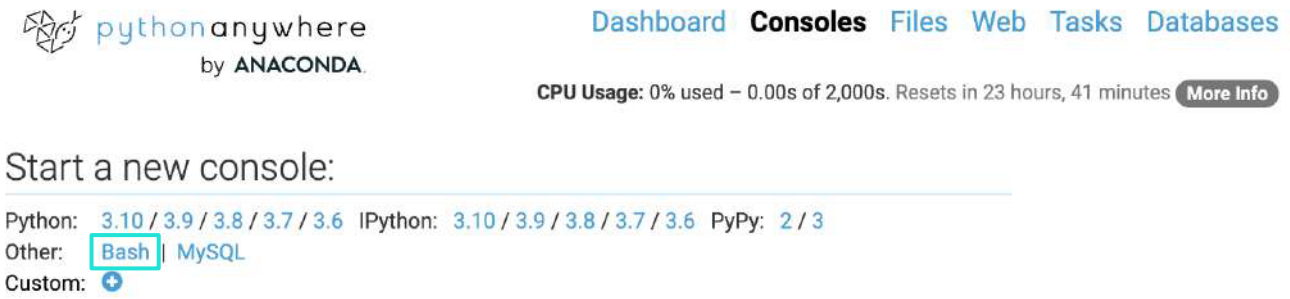


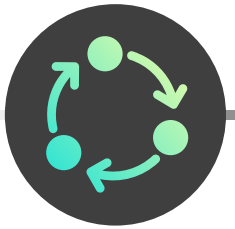
You can deploy your app on **PythonAnywhere** for free by following these steps:

4) Your default app is **live**!




5) Go to the "Consoles" tab and **start a new Bash Console**





# PYTHONANYWHERE



You can deploy your app on **PythonAnywhere** for free by following these steps:

6) Type “mkvirtualenv myvirtualenv --python=/usr/bin/python3.10” to **create an environment**

```
18:35 ~ $ mkvirtualenv myvirtualenv --python=/usr/bin/python3.10

^[created virtual environment CPython3.10.5.final.0-64 in 17314ms
  creator CPython3Posix(dest=/home/cwbruehl/.virtualenvs/myvirtualenv, clear=False, no_vcs_ignore=False, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy, app_data_dir=/home/cwbruehl/.local/share/virtualenv)
    added seed packages: Brotli==1.0.9, Flask==1.1.2, Flask_Compress==1.13, Jinja2==2.11.3, MarkupSafe==2.0.1, Werkzeug==2.0.3, attrs==22.2.0, certifi==2017.1
1.5, chardet==3.0.4, charset_normalizer==2.1.1, click==6.7, dash==2.7.0, dash_bootstrap_components==1.3.0, dash_bootstrap_templates==1.0.7, dash_core_componen
ts==2.0.0, dash_html_components==2.0.0, dash_renderer==0.11.1, dash_table==5.0.0, decorator==4.1.2, fastjsonschema==2.16.2, idna==3.4, ipython_genutils==0.2.0
, itsdangerous==2.0.1, jsonschema==4.16.0, jupyter_core==4.11.2, nbformat==5.7.0, numpy==1.24.1, pandas==1.5.2, pip==22.3.1, plotly==5.9.0, pyrsistent==0.19.3
, python_dateutil==2.8.2, pytz==2022.1, requests==2.28.1, setuptools==65.6.3, six==1.16.0, tenacity==8.1.0, traitlets==5.1.1, urllib3==1.26.13, wheel==0.38.4
  activators BashActivator,CShellActivator,FishActivator,NushellActivator,PowerShellActivator,PythonActivator
(myvirtualenv) 18:36 ~ $
```

7) Go to the “Web” tab and **connect your app to your environment**

Virtualenv:

Use a virtualenv to get different versions of flask, django etc from our default system ones. More info here. You need to **Reload your web app** to activate it; NB - will do nothing if the virtualenv does not exist.

/home/cwbruehl/.virtualenvs/myvirtualenv

*Replace “cwbruehl” with your username*

# PYTHONANYWHERE

Data Tables

Advanced Callbacks

App Deployment

You can deploy your app on **PythonAnywhere** for free by following these steps:

8) In the "Web" tab, **edit the WSGI configuration file** with the code below and click "Save"


```
1  import sys
2
3  # add your project directory to the sys.path
4  project_home = u'/home/cwbruehl/'
5  if project_home not in sys.path:
6      sys.path = [project_home] + sys.path
7
8  from project_app import app
9  application = app.server
```

*Your username*

*The name of the module (.py file) storing your application*

# PYTHONANYWHERE

Data Tables

Advanced Callbacks

App Deployment

You can deploy your app on **PythonAnywhere** for free by following these steps:

9) Make **tweaks to your code** to prepare it for deployment

```
from dash import Dash, dcc, html

app = Dash(__name__, external_stylesheets=[dbc.themes.MORPH, dbc_css])
```

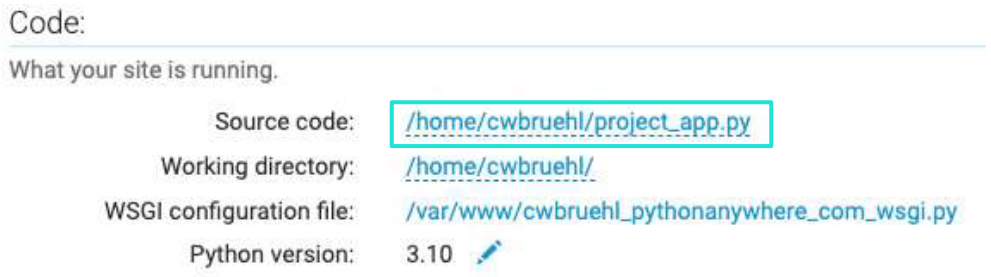
*Make sure you're using **Dash** instead of Jupyter_Dash*

```
return resort_name, elev_rank, price_rank, slope_rank, cannon_rank
```

***Remove app.run_server()** – the last lines of code should be the final callback function*

**Code tweaks can be done prior to uploading your code as well!** There are good arguments for editing it both before and after upload, but either way will get you a running the app!

10) Specify the **file path** to your app in the "Source code"



# PYTHONANYWHERE

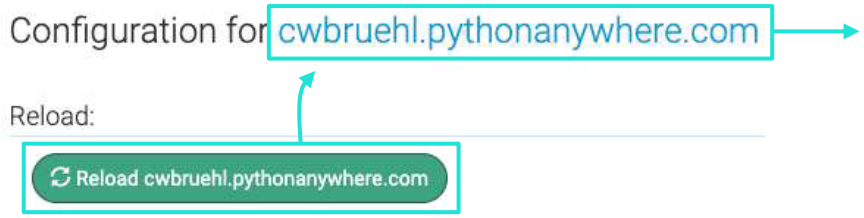
Data Tables

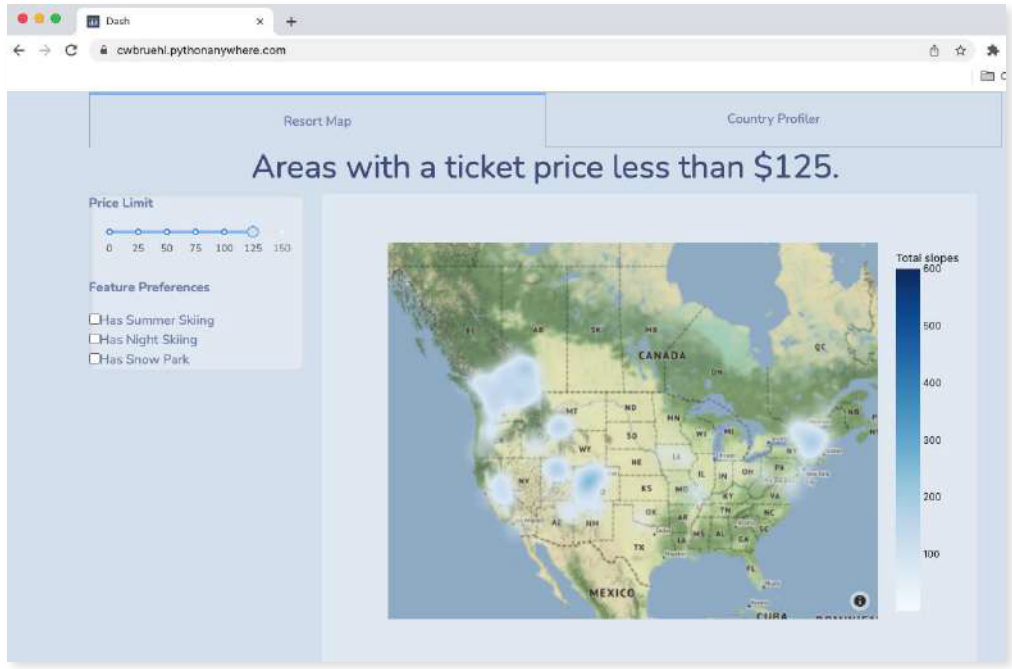
Advanced Callbacks

App Deployment

You can deploy your app on **PythonAnywhere** for free by following these steps:

11) Reload and **visit your app**!




**Got stuck somewhere?** Check out PythonAnywhere's help documentation – most of these steps are covered there! (they also have a great example of a Dash app deployment)

# KEY TAKEAWAYS

You can embed **data tables** into your Dash applications

- *This lets users inspect & analyze the raw data by sorting, filtering, or exporting it*

**Advanced callback functions** take user interactivity to the next level

- *Conditional, chained, and cross-filter callbacks let users explore the data in a wide variety of ways*
- *Manual and periodic callbacks let users control when callbacks are fired*

Dash applications are meant to be **deployed**

- *If you're building an application for work, consult your IT department about company or cloud-based servers*
- *Python Anywhere is a solid free option for deploying your app (Heroku is another popular option)*

# FINAL PROJECT

# PROJECT DATA: WORLDWIDE RESORTS

```
resorts.head()
```

| | ID | Resort | Latitude | Longitude | Country | Continent | Price | Season | Highest point | Lowest point |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | Hemsedal | 60.928244 | 8.383487 | Norway | Europe | 46 | November - May | 1450 | 620 |
| **1** | 2 | Geilosiden Geilo | 60.534526 | 8.206372 | Norway | Europe | 44 | November - April | 1178 | 800 |
| **2** | 3 | Golm | 47.057810 | 9.828167 | Austria | Europe | 48 | December - April | 2110 | 650 |
| **3** | 4 | Red Mountain Resort-Rossland | 49.105520 | -117.846280 | Canada | North America | 60 | December - April | 2075 | 1185 |
| **4** | 5 | Hafjell | 61.230369 | 10.529014 | Norway | Europe | 45 | November - April | 1030 | 195 |

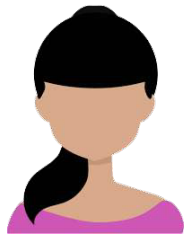
5 rows × 25 columns

MAVELUXE

```
resorts.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 499 entries, 0 to 498
Data columns (total 25 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   ID                    499 non-null    int64
 1   Resort                499 non-null    object
 2   Latitude              499 non-null    float64
 3   Longitude             499 non-null    float64
 4   Country               499 non-null    object
 5   Continent             499 non-null    object
 6   Price                 499 non-null    int64
 7   Season                499 non-null    object
 8   Highest point         499 non-null    int64
 9   Lowest point          499 non-null    int64
 10  Beginner slopes       499 non-null    int64
 11  Intermediate slopes   499 non-null    int64
 12  Difficult slopes      499 non-null    int64
 13  Total slopes          499 non-null    int64
 14  Longest run           499 non-null    int64
 15  Snow cannons          499 non-null    int64
 16  Surface lifts         499 non-null    int64
 17  Chair lifts           499 non-null    int64
 18  Gondola lifts         499 non-null    int64
 19  Total lifts           499 non-null    int64
 20  Lift capacity         499 non-null    int64
 21  Child friendly        499 non-null    object
 22  Snowparks             499 non-null    object
 23  Nightskiing           499 non-null    object
 24  Summer skiing         499 non-null    object
dtypes: float64(2), int64(15), object(8)
memory usage: 97.6+ KB
```

# **ASSIGNMENT:** FINAL PROJECT



**NEW MESSAGE**

*May 1, 2023*

From: **Deepthi Downhill** *(VP of Analytics)*

Subject: **Even MORE Ambitious Resort App**

Hey, thanks for the great work on the two dashboards.

However, I'm getting some feedback that having two separate dashboards is challenging to navigate. Can you make this a single app, with each view on its own tab? Try to improve the design a bit as well.

We also want to think EVEN BIGGER. The US and Canada were a great start, but we have access to data on ski resorts world-wide, and we should be able to leverage much of our existing code to include all of them.

Thanks!

*section07_final_project.ipynb*

Reply Forward

## *Key Objectives*

1. Build a multi-tab dashboard with a grid-based layout
2. Add multiple chart types and interactive elements
3. Write standard callback functions to connect them
4. Include a chained callback function and (if you're daring) a cross-filtering callback function

MAVELUXE