**Dashboarding Overview**

With real-time visuals on the dashboard, understanding business moving parts becomes easy. Based on the report type and data, suitable graphs and charts can be created in one central location. This provides an easy way for stakeholders to understand what is going right or wrong and what improvements are necessary. Also, getting the big picture in one place can help businesses make informed decisions which can improve performance. In general, the best dashboards answer critical business questions. Let's say you're assigned a task to monitor and report the performance of domestic US flights. Following are the yearly review report items: top 10 airline carriers in the year 2019 in terms of the number of flights, number of flights in 2019, split by month, number of travelers from California state to other states split by distance group. Let's look at two ways of presenting the report. Type 1, the report is presented through tables with inference from tables documented for reference. Type 2, here, we are presenting the same report in the dashboard format. As you can see, hovering over each chart will provide details about the data points. At the bottom in the sunburst chart, you can click on different numbers, drill down into levels, and get detailed information about each segment. Can you observe the difference in the presentation of the findings? What if we need to get the report on real-time data, not static data? Also presenting the result using tables and documents is time-consuming, less visually appealing, and hard to comprehend. A data scientist should have the ability to create and deliver a story around the finding in a way stakeholders can easily understand. With that in mind, dashboards are the way to go. Let's take a look at web-based dashboarding tool options available in Python. Dash is a Python framework for building web analytic applications. It runs on top of flask plotly.js and react.js. Dash is well suited for building data visualization apps with highly customized user interfaces. Panel works with visualizations from Bokeh, Matplotlib, HoloViews, and many other Python plotting libraries, making them instantly viewable, either individually or when combined with interactive widgets that control them. Panel works equally well in Jupyter Notebooks for creating quick data exploration tools or as a standalone deployed app in dashboards and allows you to easily switch between those contexts as needed. Voila turns Jupyter notebooks into standalone web applications. It's compatible with separate layout tools like Jupyter-flex or templates like voila-vuetify. Streamlit can easily turn data scripts into shareable web apps with three main principles: embrace Python scripting, treat widgets as variables, and reuse data and computation. There are other tools that can be used for dashboarding. Bokeh is a plotting library, widget, and app library. It acts as a server for both plots and dashboards. Panel, which is one of the web-based dashboarding tools, is built on Bokeh. Ipywidgets provides a wide array of Jupyter compatible widgets and an interface supported by many Python libraries. But sharing as a dashboard requires a separate deployable server like voila. Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Bowtie allows users to build dashboards in pure Python. Flask is a Python-backed web server that builds arbitrary websites, including those with Python plots that then function as Flask dashboards. In this video, you learned that dashboard simplifies the dynamic aspect of the business. Data can be presented by using different types of dashboards. There are different types of dashboarding tools.

**Introduction to Plotly**

Plotly is an interactive, open-source plotting library that supports over 40 unique chart types. It's available in Python and JavaScript. Plotly Python is an extension of Plotly JavaScript Library and includes chart types like statistical, financial, maps, scientific, and three-dimensional data. The web-based visualizations created using Plotly Python can be displayed in Jupiter Notebook, saved to standalone HTML files, or served as part of pure Python build web applications using dash. Here, we'll be focusing on the two sub-modules of Plotly; Plotly Graph Objects and Plotly Express. Plotly Graph Objects is the low-level interface to figures, traces and layout. The Plotly Graph Objects module provides an automatically generated hierarchy of classes, figures, traces, and layout called graph objects that are used for representing figures with a top-level class Plotly.graph\_objects.Figure. Plotly Express is a high level wrapper for Plotly. It's a recommended starting point for creating the most common figures provided by Plotly. Because if it's simple syntax, it uses graph objects internally. Let's see how to use plotly.graph\_objects submodule with a simple line chart creation example. First, import the required packages. Here we're importing graph objects as go. By writing the code, import plotly.graph\_objects as go. Then we're importing Plotly Express with import plotly.express as px command. Lastly, we need numpy to generate sample data. We're importing numpy, import numpy as np, then generate sample data with np.random.seed(10). We're setting random seed for reproducibility. Now let's create an array of 12 elements; x=np.arrange(12). Let's create random y values by using random module y=np.random.randint (50, 500, size=12). The plotly.graph contains the JSON object, which has a dictionary structure. Since we imported Plotly Graph Objects as go in the previous slide, go will be the JSON object. The chart can be plotted by updating the values of the go object keywords. We create the figure by adding a trace which is called Scatter, here. Let's view the code; fig = go.Figure( data=go.Scatter( x=x, y=y)). Next, the layout of the figure is updated using the update layout method. Here we are updating the x-axis, y-axis, and chart title. Let us view the code;

fig.update\_layout( title= 'Simple Line Plot', xaxis\_title= 'Month', yaxis\_title= 'Sales'). Then fig.show method is called to display the created plot. This is the plotted figure. Now we will create the same line chart using Plotly Express. As you can see in the example, the entire line chart can be created using a single command. Let's view the code; hashtag, entire line chart can be created in a single command; fig=px.line( x=x, y=y, title= 'Simple Line Plot', labels=dict, (x= 'Month', y= 'Sales') fig.show(). Visualization is automatically interactive. Plotly Express makes visualization easy to create and modify. It's time to play with the Plotly library. We'll use the airline reporting dataset from the data asset exchange to demonstrate how to use Plotly Graph Objects and express to create charts. Here's a quick overview of the airline reporting dataset. The reporting carrier on-time performance dataset contains information on approximately 200 million domestic US flights reported to the United States Bureau of Transportation Statistics. The dataset contains basic information about each flight, such as date, time, departure airport, and arrival airport, and if applicable, the length of time the flight was delayed and information about the reason for the delay. In this video, you learned that Plotly is an interactive open-source plotting library that supports over 40 unique chart types. Plotly graph objects is the low-level interface to figures, traces, and layout. Plotly Express is a high-level wrapper for Plotly. It uses graph objects internally.

**Introduction to Dash**

Dash is an open-source, user interface Python library for creating reactive web-based applications. It's both enterprise-ready and a first-class member of Plotly’s open-source tools. Dash applications are web servers running Flask and communicating JSON packets over HTTP requests. Dash’s front end renders components using React.js. It's easy to build graphical user interfaces using Dash as it abstracts all technologies required to make the applications. Dash is declarative and reactive. It can be rendered in a web browser and deployed to servers. It provides a simple reactive decorator for binding code to UI. They are inherently mobile and cross-platform-ready. Let's say you plan to create an application to answer business questions. As a first step, you need to determine the application’s layout, decide which chart to use, and where to place it. For example, this is called the layout part of Dash. The second part is to add interactivity to the application. There are two components of Dash. First is the core components. We can import the core components as DCC using this import statement. Next is HTML components. We can import HTML components as HTML using this import statement. Let's see them one by one. The dash\_core\_components describe higher-level interactive components generated with JavaScript, HTML, and CSS through the React.js library. Some examples of core components are creating a slider, input area, check items, and date picker. You can explore other components using the reference link at the slide’s end. The dash\_HTML\_components library has a component for every HTML tag. You can compose your layout using Python structures with the Dash Dash HTML Dash components library. The dash\_HTML\_components library provides classes for all HTML tags and the keyword arguments describe the HTML attributes like style, class name, and ID. No knowledge of HTML or CSS is required, but it can help style the dashboards. In this video, you learned that Dash is an open-source user interface Python library for creating reactive, web-based applications. It's easy to build graphical user interfaces using Dash as it abstracts all technologies required to make the applications. There are two components of Dash: Core and HTML components. The dash\_core\_components describe higher-level interactive components generated with JavaScript, HTML, and CSS through the React.js library. The dash\_HTML\_components library has a component for every HTML tag.

**Make Dashboards Interactive**

Let us understand how to connect Core and html components using Callbacks. A callback function is a Python function that is automatically called by Dash whenever an input component's property changes. callback function is decorated with @app.callback decorator. So what does this decorator tell Dash? Basically, whenever there's a change in the input component value, the callback function wrapped by the decorator is called followed by the update to the output component children in the application layout. Let's look at the callback function skeleton. First, create a function that will perform operations to return the desired result for the output component. Decorate the callback function with @APP.callback decorator. This takes two parameters. One output, this sets results returned from the callback function to a component id. Two input, this set input is provided to the callback function to a component id. From here, we connect input and output to a desired property. We will see this in action with an example using the airline data. Use the case here to extract the top ten airline carriers in the provided input year. In terms of the number of flights based on the input year, the output will change. First, we import the required packages as seen before, we'll import pandas Dash core and html components. The new entry here is Dash dependencies. From Dash dependencies, we're importing input and output that we will use in the callback function. We read the airline data into the Pandas data frame. We load our data frame at the start of the app and it can be read inside the callback function. We will start designing the Dash application layout by adding components. First, we'll provide the title to the Dash app using the html heading component H1 and style it using the style parameter. Next, we're adding an html division and textInput core component in dash.Dash(). The inputs and outputs of the application are simply the properties of a particular component. In this example, our input is the value property of the component that has the id input-yr. By default, the value has 2010. We will update this value in our callback function. Lastly, we will add a division with a graph core component. The core component has bar plot as id, which we will update inside the callback function. Note the component IDs. We will add a callback decorator app.callback input to the callback will be the component with id input-yr and property value. Output to the callback will be the component id, bar-plot and property figure. Component\_id and component\_property keywords are optional and provide clarity. Next, we will define the callback function get\_graph. The entered year will be the input. Using the year, we extract the required information from data and the application layout graph updates. Lastly, we will run the application. This is the output of the code our initial input year is 2010. As the year is updated, the graph is getting updated in parallel. The second example is a callback with two inputs. It's similar to one input callback except for a few changes. We will add a division with one more text input with the component id input-ab. Then add the new input with component id input-ab to the decorator inside a list. Next we'll define callback function get\_graph. This takes the entered year and the entered state as input parameters. Computation extracts the information and the application layout updates with the graph. This is the output of the code. Our initial input year is 2010 and the state is Al, which is Alabama. As the year and state update, you'll observe that the graph gets updated in parallel. In this video, you learned that a callback function is a python function that is automatically called by Dash whenever an input component's property changes. The at app callback decorator decorates the callback function in order to tell Dash to call it. Whenever there's a change in the input component value. The callback function takes input and output components as parameters and performs operations to return the desired result for the output component.

**Understanding the Lab Environment**

Skills Network labs cloud IDE provides a hand-on environment in your web browser for completing course and project-related labs. It utilizes there an open source IDE platform that can run on a desktop or the Cloud. When you open the lab, the screen will have two sides. On the right side, you'll see the labs instructions. On the left side is the Integrated Development Environment, IDE, where you'll complete the following; use terminal, write code, and so on. Let's have a closer look at this IDE window. At the top, there is a core menu that comprises of sub menus to help you create new files, save, run, and debug a terminal sub menu and so on. On the left side is a project-specific menu where you can find options to launch your application. To open a new terminal, click the terminal in the core menu and then click new terminal. Here, we have completed it. Great, let's start with the lab. We will work on the airline reporting carrier on time. The performance dataset source from the data asset exchange encompasses details on nearly 200 million domestic US flights reported to the United States Bureau of Transportation Statistics. It includes essential details about each flight, such as the time, date, departure airport, arrival airport, duration of flight delay, and information pertaining to the cause of the delay. You will analyze flight delays and a dashboard. You'll be creating five line graphs with dashboard components for monthly average carrier delay, monthly average weather delay, monthly average national air system delay, monthly average security delay, monthly average late aircraft delay by reporting airline for the given year. One, in the terminal, install all the required packages as mentioned in the instructions, copy the PIP commands. Two, paste it into the terminal and then press Shift plus Enter to execute them. Do this for all three PIP commands one by one. It will take a while. Let's now create a new script file to write the code. One, click file in the core menu bar and then click New File. Two, a pop out prompt will appear on the screen. Here, mention the name of this file as indicated in the instructions, flight\_delay.py and click "Okay". Three, here it is. The file is ready and open in the IDE. First we need to import the libraries and pull the dataset into the script file. One, the instruction side of the screen mentions all the required libraries. Additionally, the code mentioned to fetch the dataset with pandas is airline\_data equals pd.read\_csv. Two, copy and paste them into the script file. Three, as shown here. Now it's time to design the layout of the app. One, we will first create the dash app by calling the dash function, app equals dash.dash function. Two, to add a layout division using HTML, you can utilize the dcc.input function tag inside the layout division along with the div function and input components. Three, using the display as flux for two outer divisions to get graphs side-by-side in a row. Four, right or copy the code from the instructions and paste it into the script file below the previous code. Now that we have the skeleton, let's include the components. One, the title will go in the html.H1 function as a string, and you can style the text. Two, inside the first HTML division, which we refer to as the dcc.input tag, the input component can be customized with various attributes. For instance, you can assign the ID as input year, set the value as 2010, specify the type as number, and even customize the style attributes such as height and font size. One, when utilizing this dial parameter, you can actively add different graphs to the divisions. We assign display as flex to place two plots side-by-side in a row. Use dcc.graph tag for including the plot and give each plot an ID. For the year input by user, we need to compute the airline data and make charts and plots. One, let's call this function compute\_info. Pass the dataset and the input year as arguments, the function should return all the average delays for that year. Two, within this function, we will aggregate the specific delay by grouping the dataset for the month and the reporting airlines features. You can compute all the delays as individual plots. But best practice would be to create a function. As we need our dashboard to update in real time based on the year value input by the user, we will now create a callback function. First, we will create a callback decorator. One, update the output component ID parameter with the IDs provided in the dcc.graph function component for all five components we created and set the component property as a figure, like this. Two, and then update the input component ID parameter with the ID and set the property as value. Three, then create a callback function that uses the input provided to perform the computation. Four, create a graph in this function and return it as an output. Five, lastly, call the run.server function on the app. Six, good work. Seven, to run this application, use the command python3flight-delay.py in the terminal. One, you will get the port number after you run the application. Two, now you need to launch it. Click the Skills Network Toolbox. Three, then click the Launch Application. Four, enter the port number, in this case 8050 in the application port box. Five, and then click your application. Six, the output should look like this. In this video, we have walked through the lab where you developed a dashboard with dash framework on flight delay time statistics.