# Communicating Data Insights

by Janani Ravi

Start Course

Bookmark Add to Channel Download Course



# Course Overview

## Course Overview

Hi, my named is Janani Ravi, and welcome to this course on Communicating Data Insights. A little about myself, I have a Master's Degree in electrical engineering from Stanford and have worked at companies such as Microsoft, Google, and Flipkart. At Google, I was one of the first engineers working on real-time collaborative editing in Google Docs, and I hold four patents for its underlying technologies. I currently work on my own startup, Loonycorn, a studio for high-quality video content. Providing crisp, clear actionable points of view to senior executives is becoming an increasingly important role of data scientists and data professionals these days. In this course, you will gain the ability to summarize complex information into clear and actionable insights. First, you will learn how to sum up the important descriptive statistics from any numeric dataset. Next, you will discover how to build and use specialized visual representations such as candlestick charts, Sankey diagrams, and funnel charts in Python. You will then see how the data behind such representations can now be fed in from enterprise-wide sources such as data warehouses and ETL pipelines. Finally, you will round out the course by working with data residing in different public cloud platforms, even in a hybrid environment. That is with some of it on premise and some of it on the cloud. When you're finished with this course, You will have the skills and knowledge to

pull together data from disparate sources and use nifty virtualizations to convey crisp, actionable points of view to a senior executive audience.

# Communicating Insights from Statistical Data

## Module Overview

Hi, and welcome to this course on Communicating Data Insights. In this module, we'll focus on communicating insights from statistical data, and for this, we'll work with the Matplotlib visualization library in Python. We'll see a brief overview of the different visualization packages available in Python, and discuss their strengths and weaknesses. We'll also discuss how you can pick the right visualization type based on your use case. We'll then view box plots and violin plots using Matplotlib and see how we can view frequency distributions using histograms. We'll then move on to visualizing composition data using pie charts. Using pie charts, we'll see the proportion of the different households living in different geographical locations in California. If you have time series data, and if you want to see whether there a exists a periodic relationship in this data, autocorrelation is a great visualization to use. We'll visualize the number of visitors to the Grand Canyon over a series of months and explore whether there are any seasonal effects in the number of visitors. And finally, we'll work with stacked plots and stem plots in Matplotlib.

## Prerequisites and Course Outline

Before we move on to the actual course contents and demos, let's take a look at the prereqs that you need to have so that you make the most of your learning. This course assumes that you're familiar programming in Python and you know how to use visualization packages. You should also be familiar with basic SQL queries, nothing really advanced. It would be helpful if you had some familiarity working on cloud platforms, such as Azure. This is not a requirement though. And the only math that you need is high school math and stats. Let's briefly look at what we'll cover as a part of this course. We start off by seeing how we can draw insights from statistical data, and we do this using the Matplotlib library. We'll then see some of the really interesting charts that Plotly has to offer to visualize business data. We'll then move on to using the Seaborn package for visualization, which we'll use to visualize relationships and distributions. In order to communicate

data using insights, you need all of your data in one place, and that one place could be a data warehouse. In this module, we'll see how we can integrate data in multi-cloud environments to a single location, an Azure SQL data warehouse. And finally, we'll talk about how we can integrate data in hybrid environments. We'll specifically look at how we can move over data that is located this is located on an on-premise machine to the Azure cloud.

## Visualizations and Use Cases

Before we get started with communicating data insights using visualization, let's see a quick overview of the visualization libraries that are available in Python. Now visualization is an extremely important step in exploratory data analysis. That's how you understand and get an intuitive feel for your data. Visualization can be used to understand what relationships exist before you use this data for modeling. In short, visualization techniques are an important precursor to higher-level data analysis using modeling techniques such as machine learning. Visualization, when combined with interactivity, helps with exploration and experimentation, if you have interactivity built into your charts and graphs that allows you to drill down in your data in more granular detail. If feel that visualization gets its due importance. However, it's easy to underestimate the importance of interactivity in your visualization, and often, interactivity is lacking from many visualization tools. If you feel your data is complex and hard to understand, you might want to use a visualization tool that includes interactivity that enables exploration and dramatically increases your understanding of the data. Now, Python offers a whole host of libraries that you can use for visualization. Matplotlib, Seaborn, Bokeh, and Plotly are just a few examples. Each of these have their own community, but by far, Matplotlib is the most widely used. Each of these libraries come with their own signature strengths. Matplotlib is powerful and allows for very granular customization. Seaborn is extremely easy to use, especially when you're working with statistical data. Seaborn is built on top of Matplotlib. Bokeh is great for interactivity, and Plotly is great for custom charts and for collaboration. Bokeh and Plotly are interactive by default. Before we move on to plotting actual charts and graphs, let's see the visualization type and the corresponding use case for the common chart types. Pie charts are used to represent parts of a whole. The sections, or the wedges of a pie chart, give you the proportion corresponding to the whole pie. Line charts are a great visualization tool when you have time series data, and you want to view changes over time. Bar graphs are what you'd use to compare the same quantity for different groups or categories. Histograms are great for univariate analysis, when you have just one variable and you want to see how that variable is distributed across buckets or bins. Stacked bar charts allow you to compare parts of a whole, as well as compare across groups. If you're

looking to understand bivariate relationships in your data, that is relationships that exist between two variables, a scatterplot is a great choice that allows you to view co-movement, as well as allows you to detect outliers in your data. Box plots, on the other hand, are a great visualization tool for descriptive statistics. It allows you to view mean, median, quartiles, the interquartile range, as well as outliers. Violin plots give you the same information as box plots do. They're similar to box plots, but they also show you the probability density of the variable, allowing you to see how the data points of the variable are distributed. Rug plots are a special kind of visualization commonly used in Seaborn. This is can be through of as a one- dimensional scatter plot, also called a zero bin-width histogram. If you want to view the frequency distribution of your univariate data in the form of a smooth curve, you'll use the KDE plot. KDE stands for kernel density estimation, and this is a smoothened probability distribution of all of your data points. The violin plot that we discussed earlier is simply a combination of the box plot and the KDE plot. There are a number of specialized visualizations that we'll encounter in this course, and one is the Sankey diagram. This allows you to view the flow of information between nodes. The Sankey diagram was first used to described Napoleon's invasion of Russia and the number of troops that he had with him going in and coming back. Another specialized plot that we'll work with to view business data is the funnel plot. This gives you sequential step-by-step processes showing you the loss at each step, for example, a funnel of number of visitors to your site, to purchases. We'll also work with candlestick plots that allow us to visualize stock price movements.

## Getting Started with Azure Notebooks

In this demo, we'll see how you can use Matplotlib to visualize statistical data. We'll calculate a few descriptive statistics and plot them on charts. We'll be writing all of our Python code on Azure Notebooks. These are hosted Jupyter Notebooks on the Azure Cloud, and you can access them at notebooks.azure .com. Sign in with your Azure subscription account, specify your username, this is a personal account, not an organization account, put in your password, and you'll be signed in. All of your notebooks are contained within projects. Head over to the My Projects tab, and there you'll be given an option to create a new project, which is exactly what I'll do. I'm going to call this project CommunicatingDataInsights. Click on the Create button, and once this project has been created, I'm going to create a new folder to hold the datasets that I'm going to work with. Click on this drop-down, hit new folder, and name your folder. This is just the datasets folder. All of the file sets we are going to work with are on my local machine, so I'm going to click through to the datasets folder, using this drop-down, and upload from computer. This brings up a dialog where there is this blue button here allowing me to choose files from my

local machine. Select the XML, CSV, and JSON files that we're going to work with, and hit Open. Hit Upload, and all of these files will be uploaded to Azure Notebooks, and you'll have them on the cloud to work with. Head up one level to our Projects directory and click on this drop-down to create a new Python Notebook. Here is where we'll start writing code. Give this notebook a meaningful name, I've called it VisualizingStatisticalData, and choose the kernel which you want to execute your code. All the code in this course will be written using Python 3.6. Once you see the new notebook listed here, click on it and this will open up on a new tab. Wait for the Kernel ready sign. You're ready to get started with coding.

## Visualizing Statistical Data

Let's take a look at the Python version that we're working with,! python --version, hit Shift+Enter to execute code within this site. We are working on Python 3.6. Azure Notebooks comes preinstalled with most of the data science and visualization libraries that we'll use. But in order to ensure that we have the latest version of these libraries, I'm going to run a pip install -U command for Matplotlib and any other visualization libraries that we are going to use from here on. The demos of this course use Matplotlib version 3.1 .0. Next, scroll down and set up the import statements for the Python packages that we'll use, NumPy, Matplotlib, pandas, and the stats module in SciPy. We'll plot our visualizations using the higher-level pyplot API in Matplotlib. The Matplotlib version is 3.1 .0, and let's read in the dataset that we'll work with. This is a dataset that contains the monthly price for vegetable oil from 2006 to 2018. I'm going to read this into a DataFrame called oil_data. You can see we have the year, month, and prices for crude palm oil, soybean oil, coconut oil, and so on. The describe function on a pandas DataFrame will give you a quick statistical overview of all of the oil data that we have in here. You can see that there are a total of 156 months of information, and you can see from the average and the standard deviation for each kind of oil, that their prices are very different. Let's confirm this. I'm going to invoke the mean function on the pandas DataFrame, which will calculate the mean of all numeric values, and I'm going to display these values in a sorted order. Ascending equal to false will display these values in the descending order, and I don't want the information for the year column, only the oil prices. You can see that the average price of fish oil across all these months is by far the highest, and lard is the cheapest, on average. I'll now use a Matplotlib bar chart to compare all prices for each kind of oil across all of these months. Every Matplotlib visualization is contained within a figure and axis, and you can configure these, their sizes, and other custom details. I've specified a rotation angle for xticks, where xticks refers to labels on the x axis. This will allow xlabels to be clearly displayed in a non-overlapping manner. I have now plotted the bar graph for each kind of

oil using plt.bar, and passing the values for each kind of oil. And here below, I've specified a title for the chart as a whole and the labels for the x axis, as well as the y axis. Hit Shift+Enter, and here is the bar graph comparing the prices of oil across the years. Now observe that we haven't specified the kind of aggregation that we wanted to apply to oil prices for this bar chart, and the default display is that the bar graph displays the highest price across the entire dataset. Observe that this orange part that represents groundnut oil is at about $2500. Let's confirm that this bar chart indeed displays the max value. Calculate the max value of groundnut oil. You can see that it's 2555. That squares with what we saw in the bar graph. And let's calculate the year and the month when groundnut oil was actually at this maximum value. And here is the record from our dataset in the month of May, year 2012. Using exactly the same formula, let's calculate the min value of groundnut oil as well. This is 0, indicating that there are a few null or missing values in our dataset. That's fine for this demo. Let's calculate the mean value of groundnut oil across all of these months and years. That's about $1400. And let's calculate the median as well. As you can see, there are helpful functions on pandas DataFrames for all of these descriptive statistics. With all of these interesting values calculated, let's view the information on the price of groundnut oil over the years using a scatter plot. On the x axis, we have the years for which we have information. On the y axis, we're going to plot the raw price of groundnut oil. On the same scatter plot, I'll plot two horizontal lines representing the max value and the min value of groundnut oil across the years. These two horizontal lines will be in the red color. I'll plot two more horizontal lines in the green and lime green color representing the mean and the median of groundnut oil prices. Set up a legend, specify a title, and the x and the y axis labels. Hit Shift+Enter, and let's take a look at what this chart looks like. And you can see that this is actually a very interesting visualization. It shows us how the prices of groundnut oil move across the years. Within a year, we have the max value, min value, the mean, and the median all plotted on the same chart. You can see that the year 2012 there was a spike in the price of groundnut oil. Prices were high through all months of the year.

## Box Plots and Violin Plots

In this demo, we'll see how we can visualize data in Matplotlib using box plots and violin plots. Set up the import statements of the Python libraries that we'll use, NumPy, pandas, and the pyplot module from Matplotlib. Some of the code that we write here generates warnings where we convert integral format numbers to floating point form. These warnings tend to interfere with the demo, which is why I've set up this filter to ignore warnings. You can rest assured that these warnings can be safely ignored. We'll work with the same oil data that we saw earlier. This time,

I'm going to read it into a pandas DataFrame and drop all records which have missing values using the drop any function. We'll take a look at a sample of this dataset, if you want to refresh your memory as to how it looks. Otherwise, let's just move on, and let's take a look at the box plot visualization of oil_data. Now we won't work with all of the oil here. I'm going to select just two columns, GroundnutOil and FishOil, and place it in a new DataFrame. The pandas library that we use to work with data is closely integrated with the Matplotlib visualization library, so you can simply call the boxplot function on pandas, and it'll use Matplotlib under the hood to display a box plot of these values. The box plot is a great tool for statistical visualization. It gives you a number of details in one go. The horizontal line here at the center represents the mean price of groundnut oil and fish oil. This box that you see is the interquartile range. At the lower end is the 25th percentile, and at the higher end of the 75th percentile. These two horizontal lines at each edge are referred to as whiskers, and this is called a box-and-whisker plot. These represent 1.5 times the interquartile range away from q3, that is the 75th percentile, and away from q1, that is the 25th percentile. Any datapoint that lies outside of these whiskers is considered to be an outlier, and they're plotted separately. You can see that for certain months, there are outlier values in the price of groundnut oil. You can configure the box plot that you want to see in pandas as well. I want to see the box plot only for lard prices, and I want the box plot to be by year. You can see in this visualization how lard prices change by year. Some years they are high; some years they are low. There is a real variation in price for the year 2008. Recent years seem to be fairly steady. I'll now extract the column names from my DataFrame, which contains the various kinds of oil, into a new array called columns. I've skipped over the first two columns for year and month. I'll now extract all of the prices of oil for all of these oils into a NumPy array, rather than use it directly from a pandas DataFrame. I'll now plot a box plot of this information using Matplotlib directly, not using pandas, which uses Matplotlib under the hood. This box plot takes in the oil data array, which is the NumPy array of all oil prices. Along the x axis we have the different kinds of oil specified by columns. Observe that I've used a list to specify the sequence of the columns as well. And here is the resulting box plot showing you how oil prices vary for different kinds of oil. You can see that groundnut oil has many outliers in price, as does sunflower oil. Let's see how we can use Matplotlib to customize this box chart and make it colorful. Here are the colors that we want to use for the different boxes. Specify the same box plot visualization as before. We pass in the oil_data_array. Patch_artist is set to True. This is what we need for customization. This will allow us to store the patches that make up our box plot in this bp variable. The labels here are the names of the different kinds of oil. We then run a little for loop to iterate over the box representation for each kind of oil. As you can see, the individual elements that make up our box plot can be accessed using Matplotlib, and this actually allows us to perform very fine-

grained customization. Here I've set the facecolor of each box to a color from the array that I'd created earlier. The caps here refer to the whiskers of our box plot. Every box plot has two whiskers, at the bottom and at the top. I've used indices to access the bottom whisker for each box plot and assigned it a color. The rest of the customization you already know. And here is our colorful box plot representation. Each box representing a different oil has a different color, and the lower cap here is the same color as the box. Here are all of the individual elements of the box plot that you can access and customize when you set patch_artist equal to True, boxes, whiskers, caps, medians, and fliers. Here are nine box plots. So we have nine boxes, nine medians, nine fliers, and the number of whiskers and caps are double because they are on either end of each box. The violin plot shows us exactly the same information as a box plot, so often you might want to use the violin plot in place of the box plot. The one significant difference here is that instead of using the box representation for the distribution of the data points, you get a probability density function representing how the data is actually distributed. What you see on each side of the vertical line is a kernel density estimation curve, which gives you a better idea of how data points are distributed, thus has more information than a plain box plot. Let's improve this violin plot a little bit. I'll show median information here as well. Showmedians is true. I'll also specify the oil types on the x axis so that we know the prices of which oil a particular visualization corresponds to. And here is the resulting violin plot. We now have the median information using the center line, and we also have the shape of the distribution of prices.

## Histograms

In this demo, we'll see how we can visualize univariate distributions of data using histograms. Once again, we'll work with the same oil prices dataset that we've seen so far, and we'll drop all of the records that have missing values. Here is the oil data, in case you've forgotten how it looks. But if you remember, you can just move on. I'll now perform a grouping and aggregation operation using pandas. I want to find the average oil price by month for fish oil. So I'm going to group by month and calculate the mean. Observe that the result here has the group column set to month. We get the average fish oil prices across years for all months of the year. This will allow us to see whether there are any seasonal effects on the fish oil. Now let's visualize how fish oil prices are distributed using a histogram. This histogram will display the prices of fish oil. The color of the bars will be yellow, and the color of the edges of the bars will be red, and we'll have a total of 20 bins, or 20 bars. And here is our bright yellow histogram showing us the frequency distribution of fish oil prices. You can see along the x axis, prices have been bucketed into 20 bins. Along the y axis, the height of the bars represents the frequency distribution, how many records

from our dataset have fish oil prices in this range. The highest bar here is at about $ 17.50. Most of the records in our dataset have fish oil prices in this bucket. Histograms represented using Matplotlib can be thought of as made up of three components. N is the frequency distribution, bins refers to the edges of bins along the x axis, and patches refers to the boxes, or bars, drawn for the histogram. The combination of n and bins will give us the actual frequency counts for each bucket. So there are only three records where the monthly price of fish oil has been between $0 and $620. If you want a cumulative frequency distribution rather than a frequency distribution for just individual buckets, you can pass in cumulative=True to the plt.hist function. Observe here that along the x axis, we have the buckets or the bins for the histogram, and you can see from the increasing height of the bars that every bar represents the cumulative count of records from the particular bucket and all previous buckets. Let's say you wanted to use this histogram representation to compare the frequency distribution of two kinds of oil on the same chart. That's possible as well. I'll first plot a histogram of the fish oil prices, this is the same histogram that you've seen before, on the same plot I'm going to plot another histogram for the prices of groundnut oil. I've changed the color of the bars for this histogram. They are lime green in color. The edgecolor is red, and the alpha is 0.6. Alpha refers to the opacity of these bars. This opacity specification is needed so that you can see both frequency distributions. Here are two frequency distributions overlaid one on top of the other, allowing for easy comparison.

## Pie Charts

In this demo, we'll work with pie charts in Matplotlib and see how subsets of data make up the whole. For a little variety, let's work with a slightly different dataset here. This is the California housing dataset, and the original source is here at this URL that you see here onscreen. Read the contents of the housing.csv file into a pandas DataFrame, and let's drop all records which have missing values. And let's take a look at what this data looks like. These are prices of houses in the state of California. We have lat, long, location, median age of the house, total rooms, total bedrooms, and so on. The numbers seem a little large here because the house prices and the total rooms and population, etc., refer to towns. So these are summary statistics for all homes in a town, median income, median house value, and so on. Let's explore this data. We have ocean_proximity for a particular town and households for a town. Let's extract this into a separate DataFrame called households_data. Let's get more information from what we have. I'm going to group the households data by Ocean_proximity and sum up the number of households in each category. The result grouped DataFrame will give us an idea of the number of households in our dataset that lie less than one hour from the ocean, are inland, are on an island, and so on.

And we can now visualize this using a pie chart. Let's take a look at the columns of this group data first. The columns are ocean_proximity and households. Let's now view the number of households in each kind of geographical area using a pie chart. I access the households from the group data. The labels are ocean_proximity, that is the actual group, and I want to display it using one decimal point. And the result here is this nice little pie which shows you the proportion of households for each kind of location. You can see here that most of the households lie less than one hour from the ocean, and island is such a small proportion that you can't even see it here on the pie. The default colors for our pie chart are kind of boring. I'm going to customize our pie chart with a few custom colors which I think are bright and cheerful. Let's view the same pie once again, but this time I've specified colors for the different sections of the pie. Here is the same pie chart that we saw earlier, but now each section has a different custom color. This pie chart is rather small, and for some of you, it may also be distorted so that the two axes, x and y, are not equal. Let's fix that. I'm going to plot the same pie chart with the same colors once again, but this time, I have set plt.axis equal to equal. This means my x and y axes will both be equal. This pie will actually be circular in shape. Plt.axis equal is a way to enforce that if it's not already the case, and the pie is also larger and clearer to view. Of all of the details that you're displaying this pie chart, let's say there is a certain section or category that is significant or more important, and you want to highlight it in some way. You can explode that section of the pie. Here is an explode tuple with five fields corresponding to the five sections of our pie, and only one of these fields has a non-zero value equal to 0.2. That is the section that will be exploded out of the pie and highlighted. We'll set up the same pie chart with the same group data as before, households and ocean_proximity, and we'll pass in this explode tuple as an input argument. In addition, I've also accessed the pie wedge at index 4, that is our fifth wedge, and set its edgecolor to be blue. And here you can see in the result that one section of the pie has been exploded out. This allows us to highlight those households which are near the ocean. In the default ordering that Matplotlib has for this particular pie, this is the fifth wedge, the last one. Getting the wedge right is something that you'll have to play around with for your particular dataset.

## Visualizing Autocorrelation

In this demo, we'll see how we can use Matplotlib to visualize autocorrelation relationships in your data. Autocorrelation can be thought of as the correlation of any series with itself. Autocorrelation is typically used with time series data see if there are any periodic effects. The dataset that we'll work with here that we'll read into a pandas DataFrame are Grand Canyon visits, number of visitors to Grand Canyon. This is data that I have put together manually using

this URL as the source. If you look at the sample of records in this DataFrame, you can see that this gives you the number of visitors by month across a number of different years starting with 2011. Invoking the describe function on the NumVisits column will give you a quick overview of this data. You can see that we have information for 84 months, and you can see the average number of visitors was 426, 000 across these years. While working on this demo, I've found that the autocorrelation visualization in Matplotlib doesn't really work well with these large numbers, so I'm going to divide NumVisits by 1, 000 so that all visits are expressed in terms of thousands of individuals. These are numbers that Matplotlib can handle. This is time series data, where we have the number of visitors per month. This is to the Grand Canyon. Let's visualize this using an autocorrelation plot to see whether there are any patterns in the number of visitors by month. Simply pass in the NumVisits column to plt.acorr. Observe that maxlags here is 6. This will plot the correlation of our original time series data with itself with 0 lags, 1 lag, 2 lags, all the way up to 6 lags. Here is the resulting autocorrelation visualization. Let's see if we can interpret this. When number of lags is equal to 0, that is just the correlation of the series with itself. This value will always be 1, perfectly positively correlated because it's the same series. On either side, we have a lag of 1, 2, 3, and so on. You can see that the correlations seem to be weaker as we increase the number of lags. Because maxlags was set to 6, we did not see whether there were any yearly seasonal effects. This time, I'm going to plot the same autocorrelation visualization for monthly visits, but with the maxlags set to 13. Twelve months make a year; this will allow us to see if the number of visitors in a particular month of the year is closely correlated with the number of visitors in the same month of the next year. And if we take a look at the shape of the resulting autocorrelation graph, you'll see that this is indeed the case. Every 12 lags, the correlation value is high, indicating that there are very strong seasonal effects in the number of visitors to the Grand Canyon. You can confirm this trend by plotting the same autocorrelation graph with the maxlags of 48. This will give you correlation data across four years, and you can see the strong seasonal effects are visible here as well.

## Stacked Plots and Stem Plots

In this demo, we'll see an example of using stacked plots and stem plots to visualize our data, once again working in Matplotlib. We'll work with data from national parks once again. This has been culled from the original source that you saw earlier. This data is a little different though. It contains the year, the number of visitors to Badlands, the Grand Canyon, and the Bryce Canyon. This is data for the yearly number of visits to these national parks, starting from 1961. Into a variable named x, I'm going to extract just the values present in the Year column. X is equal to

np_data and the Year column, and into the y variable, I'll extract the number of visitors to the three national parks, but I'm going to vertically stack this information. So in the lower portion of the stack, we'll have Badlands data, then Grand Canyon, then Bryce Canyon. Here is what the resulting vertically stacked array looks like. We need vertically stacked data to plot stacked plots, and that why we've set it up this way. Let's set up the labels in the order of the stacked data, Badlands first, Grand Canyon, and then Bryce Canyon. Matplotlib allows you to visualize a stacked area plot using plt.stackplot. On the x axis, we have the years for which we have the number of visitors information available, and on the y axis, we plot our stacked data, vertically stacked data. And here is our resulting stacked area chart. Blue represents the Badlands, orange the Grand Canyon, and green the Bryce Canyon. You can see that the number of visits made to Badlands has remained more or less constant all the way through the year 2000. The Grand Canyon has experienced a surge in the number of visitors in recent years, as has the Bryce Canyon. If you don't like the default colors that you view in your stacked plot, you can always change them. Let's have them displayed in sandybrown, tomato, and skyblue. Pass in the colors array to the colors input argument, and with just a wee bit of customization, you can see that the resulting stacked plot looks much more bright and cheerful. Instead of viewing the number of visits to each of these national parks in terms of absolute values, what if you wanted to view the difference in the number of visitors as compared to the previous years? That's possible as well. Once all of the records in your pandas DataFrame are ordered, and ours is, you can use this very useful function called diff, that we calculate the difference from the previous record for each of these column values. Let's take a look at the resulting DataFrame with the differences in the number of visitors year on year. You can see that the first record here, for the first year, 1961, is NaN, or not a number. There was no previous year to compare with. For all of the other years, these are differences in the number of visitors as compared with the previous years. So these differences can be negative or positive, depending on whether the number of visitors increased or decreased. When you want to view changes in your time series data, the stem plot is extremely useful. Call plt.stem and pass in the year on the x axis and the changes in the number of visitors for the Grand Canyon on the y axis. The title of this plot is Change in Number of Visitors. We'll see this year on year. And this is what the stem plot looks like. You can see that for most years, these stems are above the 0 line, indicating that the number of visitors increased year on year. Around the year '99, 2000, there were a few consecutive years where the number of visitors fell. Matplotlib allows for very granular customization of your visualizations, and the same is true for your stem plot as well. Here the marker format is a green dashed line. The line format is a red dashed line, and the base format is in blue. And here is our customized stem plot showing the changes in the number of visitors to the Bryce Canyon over-customized. It's a little hard to see

what's going on. But this gives you the idea that very granular customization is possible of the look and feel of your charts.

## Summary

And with this demo, we come to the very end of this module on communicating insights with statistical data. We started this module off with a brief introduction to the different visualization libraries available in Python. We then discussed the different kinds of visualizations that you might choose based on your use case. We then got hands on with Azure Notebooks, which are hosted Jupyter Notebooks on the Azure Cloud. We saw how we could use Matplotlib to visualize statistical data. We then studied box plots and violin plots to view data distributions. We then saw how we could visualize the frequency distribution of univariate data using histograms. We also saw how we could plot multiple histograms on the same chart. We then moved on to composition charts. We saw how we could view the composition of the different kinds of households using pie charts. Autocorrelations are a useful way to view time series relationships, which have a certain periodicity. We saw how we could visualize autocorrelation in time series data with Matplotlib. And finally, we worked with stacked plots and stem plots. In the next module, we'll install and work with the Plotly package, and we'll see how we can use Plotly to communicate insights from business data.

# Communicating Insights from Business Data

## Module Overview

Hi, and welcome to this module on Communicating Insights from Business Data. In this module, we'll work with the Plotly visualization package in Python, and you'll see that Plotly offers a series of very interesting charts that you can use for your visualizations. We'll start the module off by looking at bubble charts and bubble maps. Plotly bubble charts allow you to represent information using colors, as well as the size of the bubbles. Bubble maps are nothing but bubble charts plotted on a geographical map to represent geographical information, and we'll see an example of that as well. We'll then move on to heatmaps, which can be used to represent information using a color matrix. We'll then see the support that Plotly offers for time series data.

We'll plot stock price movement in the form of a line chart, and we'll also customize this line chart using a range slider. Plotly also offers built-in support for visualizations for candlestick charts, which are very commonly used to track stock price movements. Business information is often viewed in terms of a funnel, how many people visited your site, how many people bought, how many people returned. That's an example of a funnel. In this module, we'll also see how we can build funnel charts using Plotly. And finally, we'll round this module off by taking a look at Sankey diagrams to represent flow information.

## Loading XML Data

In this demo, we'll see how we can work with data that is originally in the XML format. We'll load it into Python and convert it to a form that is useful to us. XML data has a hierarchical structure in the tree format, and we'll use this API here, xml.etree .ElementTree, which is a simple and efficient API available in Python for parsing and working with XML data. Here is a brief overview of the XML file that we are going to be working with. We'll work with the same California housing data that we saw in an earlier demo. Notice that every row tag has information for a single record. Nested tags under the row tag give us the individual field information, longitude, latitude, total_rooms, total_bedrooms, and so on. I'm going to use this ElementTree Python library in order to parse and read in this housing.xml file, and I'll store it in the tree variable. Let's take a look at the root of this XML tree, and you can see it is the California_Housing_Date node. Everything looks good. All of the information, that is, our records that we want to work with, is present in the row tag. You can simply call root.findall (row), and you can see that there are about 20, 000 records in this dataset. We'll now run a for loop to iterate over every node in this root, and from each node which is a row node, we'll extract all of the information that we need. We'll then access the nested nodes that contain individual field information and access the text of each of these nodes. I'll initially just print them all out so that we know that we are extracting the right information from our XML data. And you can see from the result here, that yes, indeed, we are. I'll read in this XML data and store it in the form of a pandas DataFrame because that's by far the easiest way to work with our data. Here are the column names for the data that we're going to read in, long, lat, median age, median_income, Ocean_proximity, all columns that you're familiar with. I'll now create a new pandas DataFrame that will hold the values in these columns. Specify the columns for this DataFrame. This DataFrame is currently empty. We'll run the same for loop to extract information from our XML data, and we'll use that information to fill in our DataFrame. For each of the records that we access in the nested XML node, set up a pandas series object that represents one record or one row. The one detail that you need to watch out for here is that you

get the order of the fields and the order of your columns exactly right. They should match up. Append this one record to your DataFrame, and this for loop will ensure that all of the records in your XML file will be converted to a tabular format. And here is our DataFrame. The shape of this DataFrame should convert to you that we have all of the records in the original XML file, all 20, 640 of them. With our data in the DataFrame format, we can write this DataFrame out to a CSV format. I'll call this housing_converted.csv. If you now run an ls command in the datasets folder, you'll see this file there, housing_converted.csv.

## Bubble Charts

In this demo, we'll see how we can visualize data using bubble charts in Plotly. The Plotly library isn't installed by default on Azure Notebooks, so you'll need to install it separately. Pip install -U plotly will get the latest version, and this is what we'll work with, version 3.9 .0. Set up the import statements for pandas and modules that we'll use from Plotly. Graphs in Plotly are plotted using the graph_objs library, and we want these graphs to be available offline. In order for your Plotly visuals to display within your Jupyter Notebook window, you need to enable notebook mode. Invoke this init_notebook_mode, and set connect equal to True. In Plotly, the visualization, the line or the data points are often referred to as a trace. I'm invoking the go.Scatter trace here in order to see a simple scatter plot visualization of some made-up data. You can have multiple traces in a chart, a line, a scatter plot all together, which is why you specify traces in the form of a list. Call offline.iplot to plot this very simple scatter plot of made-up data, and here you can see the four data points that we plotted. Plotly charts are interactive. If you hover over this chart on the top right, you get an interactive menu that you can work with. You can download this plot as a PNG, you can zoom into individual bits of your graph, you can pan, you can move the graph left and right to view your data in detail. You can also select portions of your graph to zoom in. And by default, if you hover over the individual data points, you'll get more granular information. We'll try that. Observe that by default, when you hove over a data point, you'll get the value associated with that data point using tool tips, as well as highlights. Scatter plots can be thought of as bubble charts because you can represent additional information using the size of the data points, or the bubbles. The z variable here holds values that I'm going to represent using the size of the bubbles. Here is the same scatter plot that we had set up earlier, our bubble chart. X and y values remain the same. Notice that the marker size has now been set using our z variable. Everything is highly configurable here. You can configure different attributes of the marker using a dictionary. I've only specified the size attribute. And here is the same bubble chart as before, but now the bubble sizes hold information. If you hover over the bubble, you'll see the z values associated with

that bubble. I'm now going to represent even more information in this bubble chart. These values are present in the i variable, and I'll use the color of the bubbles to represent these i values. I'll set up the scatter plot once again. Here is the same scatter plot on our toy dataset, and notice that we've added even more configuration options to the marker. The size of the bubble will represent information stored in the z variable. The color the bubble represents information in the i variable. The color scale I used is Portland, and I want to show the color scale as well. Specify the trace that we want to display in this plot, and call offline.iplot. And here is our bubble chart representing four bits of information. I'll just hover over this large yellow bubble here. You can see that the tool tip displays the z value of the bubble, which is 400, and the color of the bubble gives you its i value, which is around 6. And here you thought that scatter plots could only represent information in two dimensions. Now that we've understood how a scatter plot works, let's read in our housing data from our housing.csv file into a pandas DataFrame. We are familiar with this dataset; it needs no introduction. There are 20, 000 records here. Visualizing all 20, 000 records can be problematic, so I'm only going to select 7% of the records at random. We'll now use a bubble chart to visualize these 1445 records. Here is our bubble chart, created using go.Scatter, that will represent some interesting information. Along the x and the y axis, we'll display the median_income and the median_house_value across townships. We'll go with the same bubbles or markers mode that we worked with earlier, and we'll configure our markers. The size of our markers will represent the total number of rooms within the geographical area, or townships. Now in our dataset, total rooms are really large values. The size reference will divide all of the values by 500 so that they'll scale down to smaller numbers. The proportions will be maintained though. We'll also use the color of the bubble chart to display information. The color will represent the median age of the houses. You can also specify customizations to the layout of your bubble chart. I want its height to be 600 pixels and width to be 900 pixels. I want it to have a title as well. When you have data, as well as the layout customization, you need to instantiate a figure object and pass in both of these before you plot this figure. Hit Shift+Enter, and here is the bubble chart, and there is some interesting information that you can glean from here. You can see that in these townships, as the median income increases, the house value also tends to rise. The size of the bubbles gives us the total number of rooms in that township or geographical area. It could mean that the houses there are larger, or there are just more houses. The color of the bubbles gives you the median age of a house in that township, and you can see that there are certain locations where houses tend to be really old, around 50 years. And there are other locations where houses on average tend to rather new, under 10 years.

## Bubble Maps for Geographical Data

In this demo, we'll work with bubble maps in Plotly that will allow us to view geographical information plotted on a map. Let's see how we can use bubble maps here using a simple example first before we move onto a more complex one. The type of Plotly visualization that displays a bubble map is a scattergeo visualization. Plotly allows you to specify chart types in a dictionary format as well. Specify the latitude and longitude of the locations that you want to plot using a list, a list for longitude, a list for latitude. Make sure that each of these lists has the same number of elements. I'm going to plot these points using markers the size of 10. You'll see that each kind of plot in Plotly has its own attributes for layout specification. Here we don't want to display the legend, and we want to display land. Specify the attributes of this figure using a dictionary this time, and let's plot this bubble map. Observe that there are three bubbles here corresponding to the locations that we had specified using lat and long. The first of these is the USA, the second is Null Island located at 0 lat and 0 long, and the third is a lat long location in India. Now that we understand how the bubble map can be configured and how it works, let's plot some real data. We'll work with our California housing data once again. Since the original dataset is huge, I'm going to sample a few records for us to visualize. I'm going to sample just 10% of the original records, giving us a total of about 2, 000 points that we'll display. We've seen earlier that the ocean proximity information is available in the form of strings. It's a categorical variable. I'm going to convert this categorical variable using the sklearn label encoder to a numeric format. Instantiate the label encoder and call fit transformer on the ocean_proximity column. Each category in the original column will be converted to numeric labels. Let's take a look at a sample of these labels. You can see that less than 1 hour from the ocean is represented using label 0. When a home is located inland, it's represented by the label 1, and so on. A scattergeo plot requires numeric labels, which is why we performed this conversion. Let's set up the scattergeo plot where we'll now get the lat long information from the housing_data DataFrame. We'll also use the size of the bubbles in this bubble map to represent information. It'll represent the median house value. We'll divide it by 1, 000 so that we're not working with very large values. We'll use the area of the bubble to give us an idea of the median value of a house in a township. We'll also use color information. The color of the bubbles will tell us how close a particular township is to the ocean. Let's set up the figure for this scattergeo plot, our bubble map. We won't show a legend; we want to show land, and the land should be yellow in color. Instantiate the figure using the data, as well as the layout information, and here is our bubble map. This displays the entire world by default, but all of our lat long locations are concentrated in California, which means you'll need to use the zoom and pan options using your mouse. You can use the

scroll button on your mouse to zoom in, and you can select and drag this map so that you can view the right location. If you hover over the bubbles, you'll get additional details. You can see that these greenish-colored bubbles represent townships that are located inland. Bubbles that are close to the ocean are red in color, and they're also larger, indicating those houses are more expensive.

## Heatmaps

In this demo, we'll see how we can work with heatmaps in Plotly. A heatmap is a graphical representation of data where the individual values contained in a matrix are represented in the form of colors. Let's first consider a very simple matrix. The matrix simply contains 8, 64, 3, and 37. The original matrix is a 2D matrix that has columns and rows, and the values are the z values, and I'm going to pass them into a heatmap. Let's plot this heatmap and take a look at how heatmap uses colors to represent information. You can see that the color scale ranges from blue to red. At the bottom right here, we're representing the number 64 using the red color. Here, x corresponds to the column that is 1, y corresponds to the row that is 0, and z is the value 64. This cell on the top left here represents the number 3 in our original matrix. This is the z value at column 0 and row 1. Now that we have a basic understanding of the heatmap, let's use it to visualize some real data. I'll read in the housing.csv file once again. Once again, I'll use just a fraction of the records present in the original dataset. I'll go with just 10%, which corresponds roughly to 2, 000 records. I'm going to perform a simple calculation here to calculate the median within this smaller dataset that I'm working with. This is the median of the median house value in a township. We'll use this information to categorize our records into two categories, above and below the median. We'll categorize every record in our dataset as having house price values which are above the median or not. We've created two categories in our original data, one where house prices are above the median, another where they're below the median. I'm now going to plot the information that we have to work with using a heatmap. Along the x coordinates, I'm going to use the ocean_proximity categorical variable. You know that there are five possible values for how close a town is to the ocean, and along the y axis, I'll plot whether the prices in that town are above the median or below the median. The z value, or the color of our heatmap, will represent the median house value. With this trace set up, plot the data, and let's take a look at our heatmap. You can see that the two rows correspond to false and true. Housing values are above the median in our dataset and below the median, and you can see that there are five columns corresponding to where our house is located. Let's hover over individual cells to get more detail. Here, houses are located less than one hour from the ocean, and are below the

median value. The median house value here is 165, 000, which is below the median of our dataset. And here, houses are located less than one hour from the ocean, but they're above the median value.

## Interactive Time Series Charts

In this demo, we'll see how we can use Plotly to visualize time series data. We'll change the data that we've been working with for a little variety. We'll use the stock market dataset available here at Kaggle. This contains five years of open, high, low, close prices for a stock, ideal for time series visualizations. If you take a look at the shape of the DataFrame that we loaded in the contents, you can see that we have stock market prices for about 1250 days. We'll use the go.Scatter object to visualize this time series data. It'll automatically plot a line chart because we've specified date along the x axis. So date is along the x axis, and the closing price of the stock is along the y axis. The variable I've used for the scatter object is trace_close because this represents the closing price. Go ahead and plot this data, and let's take a look at what our line chart looks like. You can see that this particular stock has done pretty well. It has been trending upwards. You can hover over the line in order to view the details of individual data points, where the stock was at at specific days of the year. I'll now set up another visualization of this time series data, this time for the open price of the stock. The scatter object I assigned to the trace_open variable and the color of the line is orange. And this is a representation of how the opening price of this stock has moved over the last five years. Plotly allows us to display multiple traces in the same visualization. I'm going to update my data list to include trace_close, as well as trace_open, and I want to view this in one time series chart. I specify my layout, instantiate the figure with this data and this layout, and here is my resulting time series visualization. Plotly has automatically added a legend because we now have two traces within the same visual. Legends in Plotly are interactive, and you can click on a particular line in the legend to choose to display it or not. I'm going to click on Stock Open. This turns off the Stock Open line, and only the Stock Close line is displayed. If you click on Stock Open once again, that line will reappear. Plotly offers a whole range of customizations that you can apply to the charts that you create. I'll show you one example of a customization that we can apply to this time series data chart. That is the use of a range slider. This only requires a change to the layout of my chart. The traces that I want to display remain the same. I'll configure the x axis to display a range slider, and the type is date. When we set up the figure, the only change is this new layout. We are going to display the same data as before, stock open and closing price. Observe that at the bottom here we have a range slider that allows us to zoom into a particular interval of time if we want to. Let's use this range slider. You can hold any

of the knobs to the left or to the right and drag it in, and you can see that our chart zooms in. It's a pretty cool bit of interactivity, and we added this to our chart with almost no additional code.

## Candlestick Charts for Stock Price Movements

In this demo, we'll see how we can use candlestick charts to view stock market data. If you haven't heard of candlestick charts before, they're actually pretty cool, and they're specifically designed to represent financial price information. The general shape of a candlestick chart is kind of like what you see here onscreen, like a box plot. A single candlestick represents stock price information for a single day, and it encapsulates four items of information, opening price, closing price, high, and low. The colors of each of these candlesticks are also meaningful. Two different colors to represent up movement of a stock on a day, or down prices move on a particular day. Some kind of red color is typically used to represent down price moves. Here the closing price will be lower than the opening price. Up moves are represented using some kind of greenish color. Here the closing price will be greater than the opening price. The box that you see here in the middle is the body of the candlestick. The edges of the box represent the open and close prices. Now close will be below open if the price moved down, close will above open if the price went up during the day. The body of the candlestick represents the price range between open and close, and the high and low values of the day are easy to infer from the shape. They are these whiskers here. Let's see how we can use Plotly to plot these candlestick charts. We'll work with the stock price dataset that we saw earlier, stockmarket.csv, loaded into a pandas DataFrame. Plotly offers a built-in class for candlestick charts called go.Candlestick, and pass in all of the information that it needs. Along the x axis, we'll represent the date information for stock price moves. Every candlestick needs to know the open, close, high, and low price, which we pass in. These are column values in our DataFrame. Go ahead and plot this trace data, and what you'll get as a result is a candlestick chart. It kind of looks like a line chart. That's because we have so many data points that the candlesticks are very, very tiny. You can hover over these tiny candlesticks and view information about a particular day. This tiny little candlestick here gives us the stock price for June 9th, 2014. Close is above open, indicating that the stock price went up. This candlestick is also green. I'll now hover over a tiny red candlestick where the stock price fell on June 18th, 2014. You can see that the default display for this candlestick chart has a slider at the bottom. You can use this slider to zoom in on a particular period to view the candlesticks in more detail. Or I'll do something a little better. Just for the purposes of this demo, I'll plot just 10 days of data. I'm going to plot this candlestick chart for just the first 10 records in my dataset. And with fewer data points

to visualize, you'll be able to see the candlesticks in more detail. The green candlesticks are when the stock price went up, and the red candlesticks are when the stock price fell.

## Funnel Charts and SVG Paths

The next visualization that we'll use to view our business data is the funnel chart. Let's first understand what it can be used to represent. Funnel charts are used to visualize progressive reduction of data as it passes from one phase to another. The broad end of the funnel is on top, that's when you have all of the initial data that you have to work with, and it progressively gets narrower as you move downwards. Funnel charts are widely applicable in the world of business. You can use funnel charts to evaluate the recruitment process, how many candidates came in, how many cleared the first round, second round, and so on. Or you can use funnel charts to calculate the efficacy of steps in sales processes. Funnel charts can also be used to track ecommerce order fulfillment or the analysis of promotional campaigns, how many people saw the ad, how many people clicked, how many people bought, and so on. As you might imagine, it's really useful to have a funnel chart view of your data. Now it turns out that Plotly doesn't really allow you to use funnel charts out of the box. However, you can use Plotly elements to draw your own custom funnel chart using Scalable Vector Graphics, or SVG. This is an XML-based vector image format for 2D graphics. SVG is an open standard, it's widely used, and offers interactivity and animation. When you use SVG, all of your elements are composed of path elements, and the path element is the XML element used to define a path, a line from one point to another. It may not be a straight line though. You'll define how a path moves in SVG using a series of commands and coordinate points. Here is a sample representation of a path that defines a triangle. The commands here are the alphabets, and they define the shape of the path. You can think of these commands as instructions to someone holding a pen. The numbers that you see here are the coordinate points on the axis. Here is a summary of the commands that you can use to draw a path. M is move from one point to another, L is draw a line from one point to another, C means draw a curve from one point to another, Q is for a quadratic Bezier curve, and Z is the command to close a path. Let's quickly visualize how this path command works. Move the cursor to point 1, 1. Our cursor is already there; we moved it there. Then let's say the next command is L 1, 4. This will draw a line from 1, 1 where our cursor is, to 1, 4. If the next command is L 4, 1, this will draw a line from 1, 4 to 4, 1. And our last command is a Z command. It simply closes this path. This command will draw a line from our current cursor position back to the beginning. And this completes our triangle.

# Drawing Funnel Charts

In this demo, we'll see how we can construct funnel charts in Plotly using SVG paths. There is no built-in Visualization object for a funnel chart. The dataset that we are going to work with to visualize this funnel is a dataset that I made up, so it's completely fake, and it's stored in app_info.csv. You can think of this as a funnel that you might have for an app on a Play Store. You have impressions, clicks, downloads, and then purchases, and you can see that we start off with almost a million impressions, but only 10, 000 people purchased our app. Let's extract the columns for phases and values and set about constructing our funnel. I'm going to store the phases in the Phases variable, and the values in the Values variable. Both of these are pandas series objects, essentially lists, and I'm going to have a list of colors as well to represent each phase. Our dataset was made up of 4 phases, so num_phases is equal to 4. I'll now set up a few variables and show you what each of these variables corresponds to in the final funnel. The plot width is the width of the widest portion of the funnel. This is equal to 200 pixels. The section height variable holds the height of a single section, or a phase. The section gap is the gap between sections. Because we are going to hand draw these phases, we'll have to do a few mathematical computations. The unit_width is the plot_width divided by the maximum of the values. Unit_width is the width used to represent a single individual. Phase_widths represent the width of each phase, or section. This is unit_width multiplied by the value for that phase. You can take a look at the phase_widths here. We start off with 200, then go to 100, then 20, then 2. I'll now calculate the total height of my funnel chart and store it in this variable height. Take into account every phase's height, as well as the gaps between sections. This gives our funnel a height of 230 pixels. We are going to draw every section of the funnel using SVG paths. The coordinate points that we'll use to plot our SVG path are calculated mathematically in this way. You can see that this particular point here references the top-right portion of the first section of my funnel. This is the phase width of the first phase at index 0. Half of that is the x coordinate, and the y coordinate is the height. Imagine that 0 is the center line through this funnel. The bottom of this section has an x coordinate, which is equal to half the phase width of the next section. This is the point that you see highlighted using a circle. The y coordinate for this point is the height of the total funnel minus the height of a single section. What we get as the result are the four points used to define the first section of our funnel. I'll now draw the first section of the funnel using the SVG path commands. Here is what the resulting path looks like for the first section. This path sets up and connects the coordinate points that we just calculated, and finally closes the path. We'll now define a single section of the funnel using a dictionary format. The type is path, pass in the path commands, specify the color, and specify the line color as well. Pass the section dictionary

into the layout object go.Layout, pass this layout into a figure object, we don't represent any additional data, and plot this figure. And this gives us one section of our funnel. Now you should know that even though the code seems complicated, this is pretty boilerplate code, and you can just fill in the values for your dataset and use the same code to build your funnel. That's the whole idea of this example. So set up a list for shapes, path_list, and y_labels. We'll fill this in for every section of the funnel using this for loop. We'll need to perform a slightly different mathematical calculation for the last phase of our funnel. That's a square-shaped one. The coordinates of the remaining sections of the funnel can be calculated using the same math that we saw earlier. Pass in these coordinate points to an SVG path specification that draws a section, print out the points, and the path appended to a path list. Append the height location for the y label for that particular section, and reduce the value of height in preparation for the next iteration of the for loop, which will calculate the coordinate points for the next section. Hit Shift+Enter, and the result will give you the SVG paths that you need to use to build up your funnel. We'll run a for loop once again through all of the phases of our funnel to set up the shape dictionary object. This has the path and color specification for each section of our funnel. Here is what shapes looks like at the very end of the for loop. In addition to drawing the sections of the funnel, I'm going to set up the x and the y axis in a scatter plot. On the left side of this funnel, at these specific y coordinate locations, I'm going to have the phases displayed. On the right side of this funnel I'm going to display the value associated with each phase. The labels on the left and the right side of my funnel make up my data, set this up in a list form, and we'll finally use the go.Layout object to draw the sections of our funnel. The sections will be drawn using the section values in the shapes list. And here is the y and y axis that we've specified. Pass this data and layout into a figure object and plot this figure. And this gives us our funnel chart. Remember you can build your own funnel chart using this same boilerplate code. Just fill in your phases and values. We've spoken earlier that SVGs are interactive. You can select portions of your funnel chart to zoom into a single section. Here, if you leave your mouse, you'll zoom into the clicks portion of your funnel. If you want to get back to your original view, you'll get this helpful little tip here which says double-click. Double-click, and I move out.

## Sankey Diagrams

In this demo, we'll see how we can use Sankey diagrams in Plotly to view flow information. Sankey diagrams are a very interesting type of visualization which allow you to indicate flows. The arrows in the diagram indicate the direction of flow, from which node to which node. In this Sankey diagram, the flow of information, people, anything, is from the left to the right. The width of the

arrow indicates the magnitude of the flow. Thicker the arrow, greater the flow, or greater the values passing between these two nodes. Putting all of these visual indicators together, Sankey diagrams are used to depict a flow from one set of values to another. These colorful nodes that you see represent points which can be sources or destinations of flows. The gray crisscrossing network that you see represents the connections between these nodes. Wherever you want to represent information or people flow, you'll use Sankey diagrams, for example, transfers within a system, many-to-many mapping between two domains, or multiple paths through a set of stages. All of this can be represented using Sankey diagrams. In this demo, let's first use a Sankey diagram to represent career paths that students might take. Imagine that a number of students complete their PhD. That is the start of their career. And the other elements here in this list are other options open to them once they're done with their PhD. Let's visualize how these students make these career choices using a Sankey diagram. Invoke go.Sankey, and let's pass in our list of nodes. These are the career paths for students. We then need to specify source and target nodes for the flow. Each node is represented using a number, and this number is the index of that node in this career path list. Corresponding to each source and target, we have a value, which represents the number of students who've taken that particular career path, from a source node to a target node. Specify the layout for the Sankey diagram. We'll only give it a title, Sankey Diagram for Career Path after PhD. Instantiate a figure, pass in the data and the layout, and here is what the Sankey diagram looks like. What's really cool here is how little code you had to write in order to get to this. Let's understand the flow of students from one node to another of this Sankey diagram. A source, target, and value say that there are 53 students going from node 0 to node 3. In our career path list, node 0 represents the first element where the student has just completed his or her PhD. Node 3 represents a career outside science. And you can see that the flow connecting these two nodes is rather thick, representing 53 students. Let's look at one more example here so you've really understood the Sankey diagram. The flow from node 2 to 3, where 2 represents research staff, and 3 represents a career outside science. Seventeen students have made this transition. The connection is a little smaller. You can get these same details by hovering over the flows in a Sankey diagram. Or you can hover over individual nodes and see the inflows and the outflows.

## Summary

And this last demo on Sankey diagrams in Plotly brings us to the very end of this module on communicating data insights using business data. We started this module off by looking at bubble charts and bubble maps. Bubble charts allowed us to represent information using the size

and color of the bubbles as well, and bubble maps allowed us to use these bubbles in a geographical map. We then saw an example of how we could use heatmaps to visualize values in a matrix. We saw that heatmaps use the color of the matrix cell to represent information. We then saw the support that Plotly has for time series data. Time series data by default is represented using line charts, but you can add in nice, neat interactive customizations such as the range slider. We then studied and understood candlestick charts that allow us to visualize stock price movement information, open, high, low, and close. We saw that if a built-in chart type is not available, you can draw your own custom charts in Plotly using SVG, and you saw how we can do that for funnel charts. And finally, we rounded off this very interesting module by taking a look at Sankey diagrams to depict flow. In the next module, we'll shift our attention a little bit to visualizing relationships, and for that, we'll use the Seaborn Python package.

# Visualizing Distributions and Relationships in Data

## Module Overview

Hi, and welcome to this module where we'll continue our study of communicating data insights, this time by Visualizing Distributions and Relationships in Data. And for this, we'll use the Seaborn Python package that's built on top of Matplotlib and is far easier to use than Matplotlib. We'll start the module off with some univariate analysis where we'll visualize a single variable using histograms, KDE plots, and rug plots. We'll discuss what exactly the kernel density estimation is, the fact that it's a mathematical technique to smoothen your raw data to a probability distribution function. We'll then move on to bivariate analysis, and we'll use visualization techniques such as scatter plots, jointplots, and hexbin plots for bivariate data. We'll then move on to regression plots to view relationships. If you're planning to perform regression analysis on your data, this is a natural first step. We'll then see how we can view pairwise relationships using pairplot and PairGrid, and then move on to specialized plots used to display categorical data.

## Visualizing Univariate Distributions

In this demo, we'll see how we can use the Seaborn visualization package to visualize univariate and bivariate distributions of data, univariate when there's just one variable and bivariate when

we want to view the relationships between pairs of variables. Here on a new Azure Notebook, I'm going to do a pip install of the Seaborn library so that I have the latest version to work with. Set up the import statements for the Python packages that you'll use. Import seaborn as sns; that's the common short form. I'll also ignore warnings in my code. That's because there'll be a few warnings talking about integer to floating point conversions. They're totally okay to ignore. The seaborn version that I'm using for this code is 0.9 .0. We'll work with some new data here in this demo, the weight and height dataset, available here at kaggle.com. If you look at a sample of the DataFrame that we just read in, you can see that we have two columns in this dataset, gender information, and the weight in pounds. The shape of the DataFrame will tell you how many records this dataset has. We're working with 10, 000 records. We'll first visualize the weight information that we have, using a distribution plot that allows us to view a frequency distribution in the histogram format. Seaborn's API for visualizations is very intuitive and easy to use, but seaborn is built on top of Matplotlib, so you can specify the size of your figure, the title, ylabel, etc., using Matplotlib. If you take a look at this frequency distribution of weight, you can see that we have two peaks. It's pretty clear here that these correspond to the two genders represented in our dataset, males and females. On the x axis is the buckets into which the weight range has been divided, and the y axis gives us a count of the number of individuals in each bucket. If you want to specify the number of bins you want your x axis to be divided into, you can do that using the bins input argument. Here, the number of bins is equal to 70. Observe here that in addition to the Histogram, that is our bars, we also have a smooth curve representing the probability distribution of our data. If you want this visualization with only the probability distribution but not the histogram, set hist equal to False. And the resulting visualization will just have the probability distribution curve but not the histogram bars. This probability distribution curve is the KDE, or the kernel density estimation curve. KDE is a mathematical technique often used to get a smooth probability distribution from a histogram of raw data. Assume that you have a set of data points and you'll plot these data points in the form of a frequency distribution or a histogram. You can apply KDE, or kernel density estimation, to figure out the probability distribution of this set of points. The area under this smooth probability distribution curve must sum to 1. KDE is a standard mathematical technique that you would use to figure out the probability distribution. Let's say you want to see the distribution, but you don't want the KDE curve; you want the rug plot. KDE should be set to False; rug should be set to True. This gives us an original histogram with the bars, and observe these little lines here at the bottom. Each line represents an individual data point. This is the rug plot. If you want to view just the rug plot, you can call sns.rugplot and pass in your weight data as well. This will give you just the data points, plotted in the form of a rug plot. You can also customize the colors in which you want to view your visualizations. I'm going to set color

codes to True and plot a KDE plot that will just show me the kernel density estimation curve. I'll customize this visualization with shade equal to True and color equal to green. And here is a KDE curve with a background and the curve itself shaded.

## Visualizing Bivariate Relationships

In order to visualize bivariate relationships in data, I'll work with a new dataset, the auto- mpg.json dataset, originally available from this link here. This dataset contains mileage information for cars, along with a number of features for each car, such as acceleration, cylinders, displacement, and so on. This dataset is not clean. It contains a few missing values such as dashes and question marks. I'll replace those with NaNs, and I'm going to drop the missing values using dropna. We'll now use this cleaned dataset for our visualizations and to view insights. One of the best visualizations to use to explore relationships between two variables is the scatter plot. Here I'm going to plot a scatter plot with horsepower on the x axis and miles per gallon on the y axis. You can see how seaborn works directly with pandas DataFrames and its columns. For most of the visualizations, you simply pass in the complete DataFrame and specify the columns whose values you want to view. When you look at the scatter plot, it's very clear that there is an inverse relationship between horsepower and mileage. As horsepower rises, the mileage of the car falls. Let's now use the hue, or the color, of these data points in this scatter plot to represent additional information. This time I'll use the hue to represent the number of cylinders that a car has. Because you're using color to represent a third bit of information in this scatter plot, seaborn has helpfully set up a legend for you as well. The color of these points makes it pretty clear that cars with fewer cylinders tend to have better mileage. If you have more cylinders, mileage tends to be worse. In addition to using the hue to represent data, I'll also specify some information represented by the size of the bubbles of the scatter plot. Hue represents cylinder information as before. The size of your bubbles represents the weight of the car. I've also specified the range of sizes to use, from 30 to 300, and hue norms. Just like with the bubble chart, in Plotly, seaborn can use other dimensions of your scatter plot in order to represent information. But because seaborn is not interactive, the graphs tend to be more confusing because you can't hover over and view details of individual points. Towards the bottom right of our chart, the size of the bubbles tends to be quite large. Those are the heavier cars. A jointplot in seaborn is a combination of a scatter plot that you can use to visualize bivariate relationships, and a histogram for univariate distributions. You'll see both of these in a single visualization. In addition to seeing how horsepower and miles per gallon are related, you can also see the univariate distribution for horsepower and MPG separately. A jointplot in seaborn can be used to visualize a hexbin

representation of your data. Here you bin all of your data points into hexagonal bins, and the color of each bin will give you the density of data points at that location. Darker colors indicate that more data points are bucketed in that particular hexagonal bin. There are several cars in our dataset with a horsepower of around 150 and a mileage per gallon of around 15.

## Pairwise Relationships

The seaborn library in Python offers some very interesting plots that you can use to quickly take a look at the pairwise relationship that might exist between variables in your data. We'll continue in this demo working with the auto_mpg dataset that we saw earlier. I'm going to read this in into a DataFrame, and I'm also going to replace the missing values with NaNs and drop all records which have missing fields. Let's take a look at the pairwise relationships that exist in this data. I'll work with just a subset of columns that are present here so that all of the visuals fit on one screen. So I'll only choose acceleration, horsepower, weight, and mpg. If you want to view all of the pairwise relationships that exist between all of these variables, simply use the sns.pairplot and pass in your entire DataFrame. You'll get a matrix representation of your charts, 4x4 matrix for every pair of relationships. Along the main diagonal, you can see this frequency distribution histogram. These are univariate relationships, acceleration versus acceleration, horsepower verses horsepower. That's just a univariate relationship. Bivariate relationships such as acceleration versus horsepower are all expressed using scatter plots, by default. Every column and row is labeled. Here is a scatter plot showing a linear positive relationship, weight versus horsepower. If you want to view pairwise relationships for just a subset of your data, use the pairplot and specify the columns in the vars input argument. So we'll only look at pairwise relationships between displacement, horsepower, and weight, but in this pairwise relationship mapping, I want to view the information for cars with different numbers of cylinders in a different hue. Let's take a look at the resulting visualization, and you'll see what I mean. Observe that the univariate distribution is now represented using a KDE curve, and we have five different KDE curves corresponding to the number of cylinders present in the car. The scatter plots representing bivariate relationships such as weight versus horsepower also have their points colored in a different color based on the number of cylinders. In an earlier module, we studied the heatmap to represent information using color. Now the heatmap in the real world is often used to represent correlation data, that is correlations that might exist between the pairs of variables in your dataset. If two variables in your dataset move together in the same direction, that is, as one increases the other does as well, they are said to be positively correlated. If they move together such that if one increases the other decreases, they are said to be negatively correlated. The core function in the pandas

DataFrame will give you a correlation matrix, and you can pass this into sns.heatmap to visualize this correlation matrix in the form of a heatmap. Observe that weight, number of cylinders, displacement, and horsepower are strongly positively correlated. On the other hand, cylinders, displacement, and horsepower are negatively correlated with miles per gallon, the mileage of the car.

## Regression Plots

If you're looking to understand the relationships that exist between your variables, regression analysis is a common technique. But before you get to regression analysis, you can visualize your data using regression plots. We'll work with the same dataset as earlier, the auto_mpg.json file, which I'll read into a pandas DataFrame. This has missing values, so I'm going to replace the missing values with NaNs and drop all records with missing fields. The lmplot is a convenient API that seaborn offers to fit a regression model on your data and visualize it all in one go. The x axis will have the horsepower; y axis will have the miles per gallon. The lmplot draws a scatter plot representation of the data points and fits a regression line with a confidence interval as well. There is a clear negative linear relationship between miles per gallon and horsepower. We knew this already. The lmplot interface also makes it very simple to categorize your data and fit different regression models on each category. So you'll use the lmplot as before. Hue equal to origin will categorize and color the cars that we're representing based on their origin. You can see from the resulting scatter plot and regression lines that three separate regression analyses were performed for each category. These correspond to the three possible origins of the car, within state, out of state, and out of country. You can customize any plot in seaborn using a different color palette. Here I've used the Set2 color palette and changed the aspect ratio of my chart as well. And here is what the resulting visualization looks like. You can have seaborn fit regression lines on categorical data as well. On the x axis, we're representing the number of cylinders. This is clearly categorical. And on the y axis we have the miles per gallon. You'll see that we have a regression line through categorical data. Let's take a look at another example where we'll categorize our underlying data points and then fit a regression model. Here I want three different regression analyses based on where a car originated, and I want each of these to be in a different column. So for in-state, out-of-state, and out-of-country cars, we have three separate regression plots set up in separate columns. In addition to the lmplot, the regplot function in seaborn also performs regression analysis, but the lmplot is more powerful, which is why we've chosen to use that here. The lmplot offers many more features, as we'll explore here. Let's go back to the pairplot that we worked with earlier to view pairwise relationships. This time, I've specified kind is

equal to reg, so all of our scatter plots will be fit with a regression line. As you can see, seaborn plots are closely integrated and give you a whole host of visualizations to choose from.

## Strip Plots and Swarm Plots

When you're exploring relationships in your data, it's quite possible that some of your variables are categorical in nature. That's when you choose strip plots or swarm plots to visualize these. We'll continue working with the same dataset as before, the auto_mpg.json file. I'll read in, clean it, and here is the resulting dataset all ready for us to use. Let's just see what a strip plot is and what it represents. I haven't specified any categorical values, just the miles per gallon. And you can see that this is just a scatter plot showing you how the data points are distributed. The data points are set up to be non-overlapping so that you can view them individually. Let's now set up a strip plot to view our scatter plot representation, where one of the variables is categorical. On the x axis, we have the number of cylinders, on the y axis, the miles per gallon. Jitter equal to True will jitter the data points so that they don't overlap with one another. You can see all of the data points individually. This categorical scatter plot makes it quite clear that cars with a large number of cylinders tend to have poor mileage. Just like with regular scatter plots, with strip plots, you can use the color of the data points to represent additional information such as the origin of a car. And here is the same strip plot as before, but this time, the data points have been colored based on the origin of the car. A legend has also been set up for you automatically. Another way to view the same information as in the case of the strip plot is to use the swarm plot. This represents the same data, but the points are distributed along the categorical axis so that you can view the distribution of data better. If your dataset is huge, you might find that the swarm plot does not really scale well. All of the customizations possible with strip plots are possible with swarm plots as well. Here I'm going to color the points based on origin, and I've set dodge to True. Dodge here will ensure that the original data points have been separated by category, where the category is the origin of the car.

## Box Plots, Violin Plots, and Factor Plots

And finally, we come to the last demo of this module where we'll see how we can use seaborn to view box plots, violin plots, and factor plots. Let's get back to our California housing prices dataset for this demo. Read this in to a pandas DataFrame, housing_data. We already knew that box plots are great for viewing the presence of outliers, as well as a statistical distribution of your data. Let's take a look at a box plot of the median house value based on how close a particular

house is to the ocean, ocean_proximity on the x axis. You can see here that houses that are inland tend to have on average a lower value. There are of course many outliers here. And homes that are on an island tend to have a higher median value. Let's categorize all of the records in our dataset into two categories, above the median and below the median. Let's calculate the median of the median house value and use this for categorization. I'm going to add in a new column here called above_median. If you take a look at this dataset, this column will have been added to the very right. Above_median is true if the housing value for that town is above the median. It's false otherwise. Let's plot the box plot once again, but this time, I'm going to use the hue to divide my data into categories. Observe that we now have for each category on the x axis, two box plots, one representing towns where the median house value lies above the median of our dataset, one below. We've already seen how we can plot a violin plot using Matplotlib. Here it is using seaborn. Instead of boxes, the violin plot uses the KDE curve to represent the distribution of the data in addition to the range of the data. This probability distribution function makes it very clear that most of the houses in our dataset lie inland. For our concluding discussion on factor plots, let's go back to the automobile dataset. This is the clean dataset that we'll work with. I'll use the miles per gallon data in order to plot a violin plot. On the x axis I want the model year, and on the y axis I want the miles per gallon for the car. This violin plot representation shows us a very interesting trend in our data. The recent models of cars tend to have better mileage. You can see that the violin plots move steadily up the scale on the y axis. Here is an example of a slightly different categorical plot that you can use in seaborn. This is the bar plot, where on the x axis we'll plot the categorical cylinder value, and on the y axis the miles per gallon for all of these cars with a certain number of cylinders. And by default, you'll get a bar chart representation of the average MPG values for cars with different numbers of cylinders. On average, it seems like cars that have four cylinders tend to have the best average mileage. Another categorical plot that you can use in seaborn is the point plot. We'll represent the same information in terms of points which are connected to one another, the average miles per gallon and the number of cylinders on the x axis. And finally, let's take a look at a catplot for categorical data. On the x axis we have the number of cylinders, and on the y axis we've specified kind is equal to count, so we'll have frequency information, the number of cars in our dataset that belong to each of these categories.

## Module Summary

And this demo brings us to the very end of this module where we saw how we could use the seaborn Python package to visualize relationships in data. We started off by looking at univariate data and visualized it using histograms, KDE plots, and rug plots. In the real world, though you'll

often have many variables in your dataset and you might want to explore the relationships between these variables, there are several plots in seaborn available for this, ordinary scatter plots, jointplots, hexbin plots as well. If you're thinking about performing regression analysis on our data, a quick way to explore these relationships using seaborn is to use regression plots. These allow you to categorize and view your data in the form of a scatter plot. It also fits a regression line on your data. Seaborn also gives you specialized charts that will allow you to quickly see the pairwise relationships that exist in your data. You can customize these pairplots to show regression lines, KDE curves, and so on. And finally, we rounded this module off by looking at specialized plots to view categorical data, strip plots, swamp plots, box plots, violin plots, and factor plots. In the next module, we'll move away from data visualization and talk about an important step that comes before you can extract insights from data, integrating from multiple sources to a single destination, so you have it all together. The next module will focus on integrating data in a multi-cloud environment.

# Integrating Data in a Multi-cloud Environment

## Module Overview

Hi, and welcome to this module on Integrating Data in a Multi-cloud Environment. When we perform data analysis and visualization to communicate insights, we make the implicit assumption that all of the data that we have to work with is available in one location. But in the real world, this is not necessarily the case. The reality is that your data may be located across multiple locations, even multiple cloud providers. The reality today is that we live in a multi-cloud world. In this module, we'll get introduced to Microsoft's flagship data warehouse on the Azure cloud, the Azure SQL Data Warehouse for business analytics. In order to bring all of our data together in this data warehouse, we'll use Azure Data Factory. This will give us extract, transform, and load pipelines to move our data to the Azure cloud. We'll discuss what it means to have data silos in your organization, the problems that it can cause, and we'll discuss techniques that can be used to eliminate these silos. We'll implement a prototype of one of the techniques that we'll discuss to eliminate silos. We'll connect up silos using the Azure Data Factory ETL pipelines and bring data together to the Azure SQL Data Warehouse. We'll perform data integration across multiple cloud platforms, from Amazon's S3 to Azure.

## Data Silos and Possible Solutions to Silos

Before you can extract insights from your data, you need to make sure that all of your data is brought together into one location where it can be used. A major hurdle that you'll need to overcome first is the elimination of data silos. What is a data silo? An isolated repository of enterprise data unconnected to other repositories and unavailable for use by most users in the organization. There might be a team or two in your org who has access to this data, they know how to use it, but others don't, and that's a problem. So what are the common forms of data silos? If you have standalone relational databases that are on an on-premise location or if you've stored data in block storage such as persistent disks, or if you have data in data warehouses with semi-structured data not easily accessible to all. Now data silos in an organization are bad news. They pose many serious problems, and they have to be eliminated if you want to harness the power of your data. Data silos mean that there is no place which is a single source of truth. If you have different repositories holding the same data, but one is updated, the other isn't, you'll have the problem of working with stale, out-of-date modified data. Data silos are a major hurdle that need to overcome before you can communicate insights with data. It's hard to connect the dots when you have silos, and you might end up with political turf battles over ownership of data. Data silos mean that the same data might be present in multiple locations, which means storage costs can be significant. Also, it's hard to audit data located in multiple locations, and there are other gray areas to dealing with data in silos. Once an organization an identified that it has a data silo problem, it can work towards fixing it. You solve either by integrating the silos in some way or by using data lakes as a single source of truth. A data lake is a fairly new term, but it has become increasingly important, and you'll find that different cloud providers have their own data lake offering. This is a single repository for all enterprise data, whether it's structured or unstructured, batch or streaming data, raw or transformed data, whether stored on cloud or on premise. Data lakes, you should know, are not the same as data warehouses that have been around for a while. This is a structured data store used for analytical processing and reporting. This usually holds transformed data fed in from disparate sources via ETL pipelines. ETL here stands for extract, transform, and load operations. ETL pipelines are programs or scripts with business logic which automatedly extract data from disparate sources, transform data so that it's in an aggregated form that satisfies your schema, and then loads it into a data warehouse. Now there are two broad approaches that you could choose to go with to solve the data silo problem in your organization. Use ETL pipelines plus data warehouses, or go directly to the data lakes architecture. ETL pipelines integrating data with your data warehouses serve to connect up silos, whereas with data lakes, you eliminate silos entirely. ETL stands for extract, transform, load

pipelines, whereas with data lakes, you have a single repository for all enterprise data. Let's say you already have a bunch of data in disparate sources, and you can't really move them all in one go to the cloud. You might have ETL pipelines that connect all of your data to a single data warehouse where it can then be analyzed. But if you're starting from scratch and you want an architecture that eliminates silos from the get-go, you'll directly go to a data lake, which will hold raw data in unprocessed form. You don't have to transform it first. If you have legacy data in silos spread across your organization, it might be a good idea to go for the ETL pipeline plus data warehouse combination. It's fine for hybrid, on-cloud, and on-premise setups. But if you are cloud first and cloud only, data lakes is better. The first option, though not the most ideal one, is less disruptive. It allows for integration and phase migration of legacy data, whereas with data lakes, you have to have a one-off migration. Let's talk about some of the Azure data products that you can use to solve your data storage and silo problems. The Azure SQL Database is a managed relational database service on the cloud. The Azure SQL Data Warehouse is Azure's flagship data warehouse offering that competes directly with Google's BigQuery and Amazon's Redshift. If you're looking for a data lake architecture setup, you can use Azure Data Lake, a platform-as-a-service offering that ties Azure Data Lake storage and Data Lake analytics. The Azure Data Factory is a managed service meant for building complex hybrid ETL pipelines that allow you to integrate data silos. Once you've chosen your data storage option and you have all of the data together in one place, you can use a technology such as Power BI to visualize your data. In this module and the next one, we'll work with the Azure Data Warehouse, the Azure Data Factory, and Power BI. Data lakes are a relatively new concept, and it's rather hard at first to understand the differences between data lakes and data warehouses. Let's compare and contrast the two. Data lakes allow you to store raw data in native form. Data warehouses typically work with structured or semi-structured data and ingest data in the transformed form. Data lakes can be considered to be completely schemaless. You can store blob data or binary data such as images and video files as well. If there is a schema defined for your data, it's usually schema on read for a data lake. For a data warehouse, you have a predefined schema, which is checked before you write data out to a data warehouse. Data lakes can hold entirely unstructured data, whereas with data warehouses, you have structured or semi-structed data. Data lakes are meant to be a one-technology-fits-all option. All of your data in the enterprise will be stored in a data lake, whereas with data warehouses, you'll only store that data that you require for analytics, that is data for online analytical processing. Data lakes are optimized for fast ingestion and cheap storage, whereas data warehouses are optimized for structured retrieval and analytics and visualization.

## Setting up an Azure SQL Data Warehouse

In this demo, we'll create and set up an Azure SQL data warehouse, which will hold data from disparate sources. You can access the main Azure portal by going to portal.azure .com. You'll need to have an Azure subscription to work with. Sign in with your Azure subscription account. Here is my username, and this is a personal account, not associated with an organization. Specify the password, and go ahead and sign in. This will take us directly to the home page of the Azure portal. On the left, you'll find a link to create a new Azure resource. The compute and storage technologies that you'll use on Azure are essentially resources. The resource that we want to create is under Databases, the Azure SQL Data Warehouse. This will bring up a page where you can specify the name of the database that you want to use within the warehouse. Loony-cdi-sql-warehouse is the name of my database. Resources that you create on the Azure cloud are grouped together using resource groups. Let's create a new resource group here. I'll call it loony-cdi-rg. A resource group is nothing but a logical grouping of resources that you use for a task. Deleting a resource group will get rid of all of the resources that you've created. Your data warehouse will live on a database server, and here is where you configure the required settings for your server. If you have an already existing server, you can choose that one, or you can choose to create a new server, as I'll do here. I'll call it the loonycdiserver. The server name is a prefix for a URL that allows you to uniquely identify your server. Loonycdiserver.database .windows .net is a unique URL for this server. On this page, you can also set up a username and password to access the server. I'll have a simple dbuser, specify a password, confirm it, and you're all set. Click on the Select button here to set the server up. This server refers to the virtual machine instance that holds my data warehouse, and it's located in Central US. That is a default location. Go ahead, create this data warehouse by clicking on this Create button here. And wait for a few minutes till your data warehouse is successfully deployed. Clicking on this notification icon will give you all of the recent events that occurred. You can see that our deployment succeeded. Select the Go to resource button here, and here we are on the data warehouse that we just created.

## Uploading a File to Azure Blob Storage

Now that our SQL data warehouse has been set up, in this demo, we'll use Azure Data Factory pipelines to load data into our data warehouse. Across this module and the next, we'll load in data from multiple different sources. The first bit of data that we'll load in is in the form of a CSV file. This is some made-up information about customers that I've set up, customer id, name, phone number, and address. Let's create a table in our data warehouse to store this information. Click on the Query editor link off to the left navigation pane. Using the query editor requires you to sign in

as an authenticated user. Dbuser is the SQL-authenticated user that we had set up earlier. Sign in with the user and password. On the left is the object explorer that shows you the tables, views, and stored procedures set up for this warehouse. Before we get started writing queries, I'm going to minimize all of the left panes here so that I have more room for my query editor. I'll run a simple CREATE TABLE SQL command here to create a table named customer_details, the customer_id, customer_name, phone, and address as its fields. Click on the Run button here in order to execute this query, and this table will be created within your data warehouse. This newly created table should now be available and accessible using the Tables node in the object explorer, and there you see it, our customer_details table. This table doesn't have any data yet. We can run a simple SELECT * command and confirm that this is indeed the case. We need to load in data into this table. We'll now load in data into this table using the CSV file uploaded to the Azure blob storage container. Let's go ahead and create a new resource, and I'll open this in a new tab so that we have two tabs to work with. In order to create a blob storage container on the Azure cloud to hold our data, we'll need to create a new resource, and this resource is a storage account. Select the Storage category and choose Storage account. The storage account also needs to be associated with a resource group, and I'll choose the loony-cdi-rg that we had set up earlier. Let's specify a name for our storage account. This is the loonycdiblobstorage. We'll choose to go with the default values for all other settings. Click on the Review + create button in order to set this up. Here is a summary page for you to view the details of the storage account that you're about to create. If you're okay, go ahead and hit Create. And wait for a couple of seconds till your deployment succeeds. You'll be presented with this helpful little blue button that allows you to go directly to your resource. Within this storage account, there are different kinds of storage containers that you can create. Select the Blobs option here because we want to create a blob storage container. Blob storage is just object storage for any kind of structured or unstructured data. Click on the + Container button here to create a new blob storage container, and I'll call this the loony-cdi-container. Go ahead and click on OK, and this container will be created. Click through and hit the Upload button here in order to get data from our local machine onto this blob storage container. Clicking on this icon on the right will bring up an explorer window on your local machine. Select the customer_details file. Hit Open and select Upload. Once the upload is complete, this file is now available for you to access on the Azure cloud platform. In the next clip, we'll run an Azure Data Factory pipeline to get the contents of this file loaded into our SQL Data Warehouse.

## Loading Data from Blob Storage to SQL Warehouse Using Azure Data Factory

We'll start this clip off by creating a new Data Factory resource. Go to Create a resource under Analytics. Select Data Factory. The Azure Data Factory allows us to create extract, transform, load, ETL pipelines to load data into our cloud resources. I'll call this Data Factory the loony-cdi-df, and I'll create this within an existing resource group. This is the loony-cdi-rg that we had set up earlier. And really, that's all you need to do. Accept the default values for all of the other settings, this Data Factory will be located in East US, and click on Create. In a few seconds, your resource will be deployed. Click on the Go to resource button, and here you are on your Azure Data Factory page. Within this Data Factory, click on Author & Monitor, and this will take us to a page where we can create a new pipeline to copy data over from our blob storage container to our SQL data warehouse. We are going to set up multiple pipelines here within this data factory, so give this pipeline a meaningful name, CopyDataFromBlobToAzureDataWarehouse. Click on Next, and you'll need to create a connection to the source where your contents are located. A connection here will set up what Azure calls a linked service. This is what allows us to link storage and compute resources to our pipeline. Our data is located in Azure Blob Storage. Search for it, select it, and hit Continue. You might use this linked service more than once. We'll call it the BlobStorageLink so that we can identify it. And let's specify the kind of subscription that we have to Azure. We have a Pay-As-You-Go subscription, and that's what I've selected here. The next step is to specify the name of your storage account. This is where you'll select the storage account that contains the blob storage container with your file. This is the loonycdiblobstorage. Before we use this linked service, let's test it to make sure that the connection is successful. Click on Test connection, and when you see this happy, green checkmark, you can move ahead and hit Finish. This completes the configuration of our source linked service. Select the blob storage link and hit Next, and this will take you to a screen where you can select the file from where you want to load in contents. Hit Browse, select the loony-cdi-container, and within that, select the customer_details.csv file. Since this is a file and not a folder, we can uncheck this checkbox that says copy file recursively and then hit Next once again. The next screen allows us to specify our file format settings so that it's read in correctly. Azure has rightly detected that this is a text-format, comma-delimited file. If you move up this little splitter pane that you see here at the center of your screen, you'll be able to expand the pane that allows you to preview your file contents before you load it in. If you click on the schema tab here, you'll see the schema that Azure has auto-detected on your file, and you'll also be presented with the option to change the data types for individual fields. The customer_id field, I want to be in the Int64 format, so I've changed this data type. The remaining data types are as is. Click on Next, and let's move on. The source of our Copy Data pipeline has been set up. The next step is to configure the destination. Create a new connection, and this time the connection is to the Azure SQL Data Warehouse.

Select this as our destination linked service, hit Continue, and let's configure it. SqlWarehouseLink is the name of this linked service. Once again, we use the Pay-As-You-Go Azure subscription that we are logged in with. We need to specify a server name to connect to. This is the loonycdiserver that we had set up earlier. Your server might be configured with several databases. Choose the right one, loony-cdi-sql-warehouse is the name of my database, and use the fields here to specify the username and password of the SQL-authenticated user for this database, who's authenticated to, right to this server. Let's confirm that we haven't made any mistakes here by clicking on Test connection. And there is our pleasant green checkmark. Hit Finish, and let's move on. Hit Finish on this pane, select the SqlWarehouseLink as your destination, and hit Next. Here is where you'll specify the table where you want the contents of your data to be loaded. Choose the table that we just created, the customer_details. Hit Next. Now it's quite possible that your source data columns and the fields in your destination table do not match precisely. Here is where you can specify how you want your source data to be loaded into your destination table. This looks good as far as we are concerned. I'm going to hit Next and go on to the next page where I'll configure the settings for this copy pipeline. I'll stick with the default option for fault tolerance. I'll abort activity on the first incompatible row, so if there's any row in my dataset that's incompatible with the table where I'm loading data, this pipeline will stop. Under Advanced Settings, you can configure other options for your pipelines, such as the degree of copy parallelism, whether you want to enable PolyBase for more robust transfers, or not. Choose the default options and hit next. And finally, here is a summary page where you can quickly review all of the configuration settings that you've specified. Scroll down, make sure everything looks good, and then hit Next. And this is where you'll deploy this pipeline. Once the pipeline has been deployed, you're probably curious about how it's progressing. You can view its status by clicking on this Monitor button here. You can see that this pipeline is currently in progress. You can hit the refresh button here, or you can click through and view further details on this pipeline. And it seems from here that our pipeline has successfully completed copying over our data. Let's confirm this by querying our SQL data warehouse. Close this tab and head over to the tab where you have the query editor open. Let's execute this same SQL query. SELECT * from customer_details, and this time, our table has data. We see the results here on this pane. Our copy data pipeline has completed successfully and loaded the contents of our CSV file into our SQL warehouse.

## Loading Data from S3 Buckets to the SQL Data Warehouse

In this demo, we'll see how we can load data from other cloud platforms onto our Azure SQL Data Warehouse using Azure Data Factory. The data that we're going to work with is this CSV file,

customer_orders, which contains order details for customers on an e-commerce site. This a made-up dataset. I'll head over to the query editor on my SQL data warehouse, and I'll run a simple query to create a table called customer_orders, which contains three columns, customer_id, order_id, and product_id. Check the table's node to whether this table has been created. Yes, indeed, it has. This newly created table is empty, and we can confirm this by running a simple SELECT * command, SELECT * from customer_orders. You can see that there are no results. We're going to load the contents of this table using an Azure Data Factory pipeline, but the source of the data will be on another cloud platform, specifically AWS. Go to aws.amazon .com /console, and sign in with your AWS account. You need to have an AWS subscription. Specify your username and password, and you'll be signed in onto the Management Console. I'll now create an S3 bucket, which is the equivalent of the blob storage containers that we used on Azure. Scroll down. Under Storage, you'll find an option for S3. Click through, and you'll be taken to the main S3 buckets page. Click on this blue button here which says Create bucket, and specify a name. Loony-cdi-bucket is my name. Click on the Create button here, and this bucket will be created. Click through to this bucket, and let's get our CSV file uploaded into this bucket. The Upload button here will bring up a dialog that will allow you to add files from your local machine. Click on Add files. This will bring up our explorer window. Select the customer_orders file and hit Open. As you can see, the process to upload a file to an S3 bucket is very similar to how we uploaded a file on Azure Blob Storage. Because we're working on a completely different cloud platform, not Azure, we'll need to configure security credentials so that our Azure Data Factory pipeline can access the contents of this bucket. This requires the creation of an access key that our Azure pipeline will use to authenticate itself. Click on Access keys, and click on this button here to create a new access key. This key will be downloaded to your local machine in the form of a CSV file. Make sure that you keep it secret and you don't share it with anyone. When we set up our copy pipeline on Azure Data Factory, we are going to paste in the contents of this file as the secret key that the pipeline needs to access your S3 bucket. We are done setting up our contents on the S3 bucket. Switch tabs and head over to your loony-cdi-df, our Azure Data Factory. Click on the Author & Monitor link in order to create a new copy data pipeline. The steps that you'll follow to configure this pipeline are exactly what you did before, so I'm going to run through this rather fast. Specify a meaningful name for this pipeline. You can expand and make this screen full screen if you want to. Hit Next, and let's configure the source. Our source data lies on a different cloud platform. I'll need to create a new connection to configure this new source. Search for Amazon S3, and you'll find that an option exists within the pipeline linked service options. Click on Continue, and let's configure our S3 linked service. I'm going to call this the S3BucketLink. You can add a meaningful description. But what you really need to authenticate yourself is the access

key, ID, and secret access key. Both of these will be present in the CSV file that we downloaded from AWS. Paste in the access key, ID, and the secret access key, and let's test to see whether our connection is successful. Hopefully you made no mistakes copying in your key ID and secret access key and test connection will be successful. Hit Finish and select the S3BucketLink as your source connection and move on to the next screen. The Browse button here will allow you browse your S3 bucket exactly like you did with your Azure blob storage container. Select the loony-cdi-bucket, select customer_orders.csv, and move onto the next screen. Given that it's a file, I'll uncheck copy file recursively and move on from this screen. This is a fairly simple file, so you'll find that the file format settings on the next screen are correct. Azure has autodetected them correctly. And here is a preview of the data that we're going to copy in. The Schema tab is where you can view and modify your schema. I'm going to convert this customer_id, which is of string type, to be of Int64 so as to match the customer_id data type in my other table, that is, the customer_details table. In addition, I'll change the order_id and the product_id to also be of type Int64. With our source configuration complete, let's move on to specifying the destination for our copy. This is the SQL warehouse link. We only need to change the table where we want this data to be loaded. This is the customer_orders table. The next screen is where we specify column mappings for our data. Everything looks good here. We can hit Next and move on. We'll stick with the default settings for the copy operation. Hit Next and view a summary of the copy that we're just about to perform. Everything looks good. I'm going to hit Next once again so that I can deploy this pipeline. Once you get the green checkmark, go ahead, click on Monitor so that you can monitor the status of this copy pipeline. This pipeline is currently in progress. Let's click through and see where we are at in this copy. Click on this little icon here, and that'll allow you to drill into the details of this pipeline. On this page, this little spectacles icon will give you granular information about the state of your transfer. So far, one file has been read from our S3 bucket and 20 rows have been written to our SQL data warehouse. It seems like our pipeline is complete, and yes indeed, it is. I'll switch over to the query editor tab and run this query yet again to see if the contents have been loaded successfully. Hit Run, and here you can see from the results that we have all of the data copied over from our S3 bucket into our SQL data warehouse table.

## Summary

And with this demo, we come to the very end of this module on Integrating Data in a Multi-cloud Environment. A great place for business analytics data is a data warehouse, and in this module, we were introduced to one on the Azure cloud, the Azure SQL Data Warehouse, to store our business data. Any business organization that has been around for a while is likely to have data in

different sources. These sources might be on-premise machines or scattered across different cloud providers. In today's world, no organization likes to tie itself to a single cloud platform. However, to extract insights from your data, you require your data to be in one location, which is why the Azure Data Factory for integrating data sources is so useful. We spoke about data silos in detail and how it can hurt your organization. We discussed different kinds of silos, and we discussed two ways to solve this problem, using data lakes or a combination of ETL pipelines plus a data warehouse. Our hands-on demos focused on using the Azure Data Factory as an ETL pipeline and the Azure SQL Data Warehouse as a final destination for our data. We integrated data from a variety of different sources, the Azure Blob Storage container, we also integrated data stored on AWS S3 buckets. In the next module, we'll focus on integrating data in a hybrid environment, from our on-premise machine to the Azure cloud.

# Integrating Data in a Hybrid Environment

## Module Overview

Hi, and welcome to this module on Integrating Data in a Hybrid Environment. The truth is today, no one is tied to one cloud provider or a single location for their data. They may have data on premise or with different cloud providers. There has to be a way to bring all of this data together to a single destination. In this module, we'll continue to use Azure Data Factory pipelines to bring our data to the Azure SQL Data Warehouse. This time, we'll see how we can integrate data which is stored on premise on a SQL Server on our local machine. You'll see that our ETL pipelines require some additional special setup before this is possible, specifically, the installation of additional software, that is the integration runtime on your local machine so that your cloud resources can talk to your on-premise resources. And finally, once we have all of our data in a single location, we've effectively connected up all of the data silos that exist. That's a huge accomplishment. We'll then see how we can use Microsoft's Power BI to visualize this data.

## Loading Data to an On-premises SQL Server

In this demo, we'll see how we can use Azure Data Factory to integrate data from our on-premise SQL Server machine to our SQL data warehouse on the cloud. The reality is, today our data might

be spread across multiple cloud platforms and also on premise, so there needs to be a way to get it all together, and that's what we'll see here in this demo. The data that we're going to load in from our on-premise SQL Server is this products.csv file. It contains a product ID, product name, and the price of a product. Let's create a table to hold this data on our SQL data warehouse using the query editor on our browser window. This is the products table with the three columns that we just saw. Execute this query, and this new table should now be available within the Tables node of our data warehouse. There you see it, dbo.products. This is a newly created table. Let's satisfy ourselves that this is completely empty. SELECT * from products will give you no results. I'm currently working on a Windows machine, and I already have a local SQL Server set up, and I connect to it using SQL Server Management Studio. Using the toolbar at the bottom, click on the search icon and search for ssms. We'll connect to our SQL Server using this tool. You can see that ssms has picked up the SQL Server Express database that I have on my local machine. I already have an authenticated user set up, sa. This is a default user, and I'm going to connect using this default user. I'll use the Security node on the Object Explorer on the left to create a new SQL authenticated user. Select Logins here. You can see here that we already have a SQL authenticated user; that is sa. Select the Logins node and right-click, and this will give you an option to create a new login. Click on New Login, and this will bring up a dialog. The login user that I want is the loonydbadmin. I want this new user of this database to be a SQL Server-authenticated user. Specify a password for this user and confirm password as well. We want this user to have full access to our SQL Server database. Go to Server Roles and check all of the options that are available here. I do this just to keep our lives simple. You can select specific permissions if that's what you want. Click on OK, and you can see here within Object Explorer that this new user has been successfully created. Close this SSMS window, and let's restart SQL Server Management Studio and log in as the new user, loony-dbadmin. Specify the password for this use and hit Connect. Since this is the first time that we are logging in as this new user, you'll need to change your password, but now we are all set up. Right-click on the Databases node and select New Database. I'll create a new database here within my on-premise SQL Server. Loony-cdi-productsdb is the name of the database. Hit OK, and this database will have been created for you. Expand the Databases node, and you'll find it in there. Let's load in the CSV file that we have on this machine to create a new table within this database. Select the loony-cdi-productsdb node, right-click, choose Tasks, and choose Import Flat File. This will bring up a wizard that we'll walk you through, loading in your CSV file contents to create a table on SQL Server Express. Specify where your file is located, click on the Browse button, and select the file using the explorer window. Click on the Next button, and you'll be taken to a screen where you can see the preview of the contents that you're going to load in. Hit Next once again, and you'll be able to see schema

details. The schema looks good to me. I'm going to hit Next. Here is a summary of the import operation that we are about to perform. Click Finish, and the contents of this file will be loaded into a table within your database. Go back to your Object Explorer pane and expand the Tables node here, and you'll find that the new table, dbo.products, has been created. This is the table that holds the contents of the CSV file that we just imported.

## Configuring a Self-hosted Integration Runtime

With data loaded into our on-premise SQL Server, we're now ready to create and deploy an Azure Data Factory pipeline to load data from on premise to the cloud. Click on the Copy Data option here. I'll call this new pipeline CopyDataFromSqlServerToAzureDataWarehouse. I want to deploy this pipeline right away. Run once now is the default option that's checked. Click on Next, and let's configure the source of our data. Once again, we'll need to create a new connection. We have our data in a SQL Server which is on premise. Choose the SQL Server option. I'll call this new linked service the SqlServerLink. Now because we're integrating with an on-premise SQL Server machine, the steps here are a little different, and you'll need to pay a little bit of attention. Let's fill in the server name first. If you scroll down below, you'll need to specify the server name of your local desktop machine. This is the server name that pops up on SSMS when you try to connect to your on-premise machine. Next, let's specify the database name within this server, loony-cdi-productsdb. This is the database that we had created using SSMS. I'm going to authenticate ourselves to this database using SQL Authentication. The username is the loonydbadmin user. Remember that this user needs to have read access to the database whose contents you're trying to read. Specify the password for this user, and let's continue. I know that I have the server name, username, and password set up correctly. Let me test the connection to see whether it works, but it does not. Clearly, something more is needed. And here is where things change a little bit. The problem lies with the integration runtime that we've chosen to connect to our server. So long as we work only on the cloud within our Azure subscription, you can go with the AutoResolveIntegrationRuntime option. But when you have an on-premise machine, you'll need to configure your own integration runtime. The integration runtime is the compute infrastructure used by Azure Data Factory to provide data integration capabilities across different network environments. This default integration runtime no longer works for us, so we are going to have to create a new one. Choose the New option, and let's go on with its configuration. This will bring up a pane where you can choose first the kind of integration runtime that you're looking for. The self-hosted integration runtime that we've selected here can run copy activities between a cloud data store and a data store in a private network, and that's exactly what we want. Click on Next, and

you'll be taken to a page where you can give this integration runtime a name. I've simply called it IntegrationRuntime. Once the runtime has been successfully created on your Azure portal, you need to set this runtime up on your local machine so that your cloud resources can connect to your on-premise resources. You can choose the express setup or the manual setup option here. When I tried to use the express setup, I faced a network connectivity issue, which is why I've chosen to go with the manual setup. There should be no difference between the two except that the express setup is a little faster. Clicking on this link will take us to a page where we can install the integration runtime that we need on our local machine. Select the Download option here, and the integration runtime executable will be downloaded to your local machine. Click on Next, and this will download the MSI installer package that you need to get going. Close this tab, we're done with the Microsoft Download Center, and right-click and open up this MSI. This will bring up a wizard that will walk us through the steps of installing the integration runtime on our local machine. English is the language that we'll choose. Click on Next, accept the terms of the license agreement, and let's keep moving. Specify the folder where you want this integration runtime installed. This is fine as far as I'm concerned. Hit Next, and hit Install. This will install the integration runtime on your local Windows machine. You might have to wait for a minute or two till this installation is complete. Hit Finish, and you'll now need to register this integration runtime with your Azure cloud platform. This is what will let Azure Data Factory know how to identify and access this integration runtime to copy data. The integration runtime needs an access key to authenticate itself to your Azure portal. Switch back to the window where you were setting up the integration runtime and copy over the first of these keys. Copy the key and paste it into your dialog and hit Register. Once your authentication key has been verified, this integration runtime can connect to your resources on the Azure cloud. This is a self-hosted node. It's hosted on our local Windows machine. And it has picked up the name of our Windows machine as the node name. If you want this integration runtime to be accessible on the intranet, you can select this checkbox. Hit Next to continue with this wizard. I'm going to enable remote access without an SSL certificate. It's just a demo after all. Hit Finish, and your integration runtime will be registered with the Azure cloud. You can now launch the configuration manager for this integration runtime and take a look at its configuration settings. Thanks to the Data Factory authentication key that we had used to register this runtime, you can see that this self-hosted node is connected to our Data Factory on the cloud. Everything looks good. We're all set up here. Close this dialog, and let's move on with configuring our Data Factory pipeline. Click on Finish here. We've set up the new linked service to our on-premise SQL Server. It's now using the integration runtime that we just configured. You might see that your integration runtime is unavailable initially. It should still be good to use. Maybe wait for 2 or 3 minutes. Leave all of the other settings as is, scroll down to

the bottom, and let's test our connection to see if it succeeds. And yes, indeed, it does. Thanks to our self-hosted integration runtime, our on-premise SQL Server is now connected to our Azure Data Factory on the cloud. Hit Finish, and we're ready to move on with our configuration.

## Copying Data from On-premises SQL Server to SQL Data Warehouse

The remaining steps for configuring this Azure Data Factory pipeline are no different from what we've seen earlier. Select the SQL Server link as the source, and hit Next. Using the integration runtime, Azure Data Factory has picked up the one table that exists in our local SQL Server, the products table. Select this table. You can preview the data and make sure this is indeed the data that you want to copy. The next button will move us on to configuring the destination for this pipeline. Once again, we choose the SqlWarehouseLink. We want to move this data into our Azure SQL warehouse. Azure has autodetected the right table that matches our data. Hit Next, and let's specify the column mapping. The default column mapping looks good. We can hit Next and move on. Because we're moving from an on-premise location to the cloud, I'm going to enable staging for my data. Staging allows us to stage the data temporarily at some location before moving it into the SQL warehouse. You can specify the staging account linked service using this drop-down here. I'm going to use the BlobStorageLink here to stage my data. So our data will first be copied over to the Azure blob storage container and then to our data warehouse. We'll choose this container that we already have to stage our data, and under Advanced Settings, I'm going to turn on PolyBase. If you have a lot of data to copy from another network, allowing PolyBase makes your transfer far more efficient. Hit Next to review the copy pipeline. Notice that because we've enabled staging, SQL Server will first go to Azure Blob Storage and then to our Azure SQL data warehouse. Scroll down and verify all of the other settings and hit Next. This will deploy your pipeline and start the process of copying your data over. Let's monitor our pipeline to see whether it runs through successfully. Click through to the details of this pipeline, and here, if you click on the spectacles icon, you can see how much of the data transfer is now complete. The data that we transferred here wasn't that large. In a few minutes, you'll find all of the data will have been transferred over successfully. Let's go back to our query editor and check the contents of the products table. Execute this query, and you can see all of the data present on our local SQL Server is now available in your data warehouse on the cloud. Now that you've eliminated the data silos that exist in your organization, you can run complex join operations to extract insights from your data. Here we have a SQL query which joins all three of the tables that we uploaded to the

Azure SQL Data Warehouse. Phew, this was a long process involving several pipelines, but our data is now all in one place.

## Visualizing Data Using Power BI

In this demo, we'll see how we can use Power BI to visualize our data. We'll use Power BI Desktop that's freely available to connect to our Azure SQL data warehouse on the cloud. In order to install Power BI Desktop, you can visit this URL, or you can simply search for download Power BI on your favorite search engine. This is a link to Power BI download on the Microsoft Download Center. Scroll down and select the Download button, and this will install the Power BI MSI. Select PBIDesktop and hit Next. Once you have the installer on your local machine, you can simply double-click it or right-click and hit Open in order to start the installation process. As with all Microsoft products, this will bring up a helpful little wizard that will walk you through the steps of the installation. Make sure you accept the license agreement so you can keep moving. Power BI will install under Program Files by default. Hit Next and click on the Install button here to start the process of installing Power BI. In less than a minute, Power BI should be installed and ready for you to use. If you hit Finish, this will open up Power BI on your local machine. Using the ribbon on top, select the Get Data option. This is what we'll use to connect to the source of the data that we want to visualize. Click on the More option here so that you can see cloud sources as well. Select Azure and select the Azure SQL Data Warehouse. You'll of course need to authenticate yourself to this data warehouse. Click on Connect, and this will bring up a dialog that will walk you through the authentication process. In the Server field here, you need to specify the full URL path to your server. On the Azure portal on the main page of my loony-cdi SQL warehouse, I have my server name, loonycdiserver.database .windows .net. I'm going to copy the server name over and fill this server name in in my connection dialog. Click on OK here, and let's authenticate ourselves. Let's try using our current Windows credentials and hit Connect. This is clearly an error because our Windows account isn't authenticated to our Data Warehouse server. But if you remember, we have SQL authentication set up, and that's what we need to use. Select the Database option, and specify the username and password for your SQL authenticated user. Hit the Connect button, and let's see if we're successful. Nope. Remember, all resources on your cloud are secure by default. There are no firewall rules which enable the IP address of my local machine to connect to the Azure cloud, and this is what we need to fix. Switch back to the SQL Data Warehouse page on our Azure portal and click on the link which says Show firewall settings. This will bring up the firewall settings page for your database server. Click on Add client IP to enable your local IP to access the server. Azure has autodetected the IP address of your local machine and filled in all of

the details with an enable traffic rule. This IP address now has access to our database server on Azure. Hit Save, and we've now added this firewall rule successfully. Let's switch back to Power BI and hit Retry on this connection dialog, and this time, our connection to the Azure SQL Data Warehouse will be successful. You can expand the warehouse node in this dialog and view all of the tables that we have set up earlier. If you select the node corresponding to each of these tables, you'll be able to view a preview of the data stored in this table right here within Power BI. Here is our customer_orders table, and here is the products table that we loaded in from our on-premise SQL Server. I want to load in data from all three tables. I've selected all three tables. Click on the Load button here, and all of the data will be loaded in and available for us to visualize and work with. All three tables have been loaded into Power BI. You can expand the node that corresponds to each of these tables to view the details. We now have all of our data integrated into one place. We can now use this data to extract insights and set up dashboards. The Power BI user interface for charts is simple and easy to use. Simply select the chart type, here I've selected the bar chart, and it'll be added to your edit bay. Select the field that you want to represent on the x axis of this bar chart. I've chosen product_name from the products table. You can then select a field from another table or from the same table. I've chosen product_price from the same table. Product_price is now represented by the bars themselves, the bar height. By default, the bars will represent the result of the sum aggregation on your product price field. You can also use the nobs to resize and reposition this bar graph. You can change what the bars represent by clicking on this little drop-down here. The default aggregation is the sum. Let's switch this over to the average aggregation. The bars now represent the average price for each product in our dataset. As you can see, Power BI is intuitive and simple to use. The hard part was getting set up and connected to our SQL warehouse, and now that we've done that successfully, I'll leave you to explore all of the cool features that Power BI offers.

## Summary and Further Study

And this demo brings us to the very end of this module on Integrating Data in a Hybrid Environment. The reality is that today, your data might be located in different locations on premise or with other cloud providers, and you have to have a way to bring it all together to a single destination for data analytics. Our destination was the Azure SQL Data Warehouse, and the integrated data that was present on-premise on our local SQL Server using the Azure Data Factory and its pipelines. Once we had all of our data stored on the Azure cloud platform, we used Power BI to visualize this data. And with this, we come to the very end of this course on Communicating Data Insights. We started off by extracting insights from data, statistical insights,

business insights, and visualizing relationships. And we ended this course by integrating multiple sources of data to a single destination. If you're interested in Azure and you're interested in data, here are some other courses on Pluralsight that you might find interesting. Representing, Processing, and Preparing Data will talk about data preparation techniques, and Experimental Design for Data Analysis will focus on data modeling using the Azure ML Studio. And that's it from me here today. I hope you had fun. Thank you for listening.

Course author

Janani Ravi

Janani has a Masters degree from Stanford and worked for 7+ years at Google. She was one of the original engineers on Google Docs and holds 4 patents for its real-time collaborative editing...

Course info

| | |
|---|---|
| Level | Advanced |
| Rating | ★★★★★ |
| My rating | ★★★★★ |
| Duration | 2h 27m |
| Released | 21 Jun 2019 |

Share course