**Initial exploration**

Welcome to this course on exploratory data analysis! I'm Izzy, and I'll be your coach for chapters one and three of this course. My friend and colleague George will guide you through chapters two and four.

**Exploratory Data Analysis**

Let's say we've got a new dataset about books. Is this good data? What questions can it answer for us? It's only after we understand what our data contains that we can think about how the data might be useful to us. Exploratory Data Analysis, or EDA for short, is the process of cleaning and reviewing data to derive insights such as descriptive statistics and correlation and generate hypotheses for experiments. EDA results often inform the next steps for the dataset, whether that be generating hypotheses, preparing the data for use in a machine learning model, or even throwing the data out and gathering new data!

**A first look with .head()**

Let's begin by importing a dataset and reviewing some useful pandas methods for initial exploration! We'll import the books data from a csv file using pd.read\_csv and save it as a DataFrame called "books". Taking a look at the top of the DataFrame using the head function, we can see that our data contains columns representing book names, authors, ratings, publishing years, and genres.

**4. Gathering more .info()**

01:26 - 01:44

pandas also offers a quick way to summarize the number of missing values in each column, the data type of each column, and memory usage using the .info method. It looks like there are no missing values in our dataset, but it does have a variety of data types.

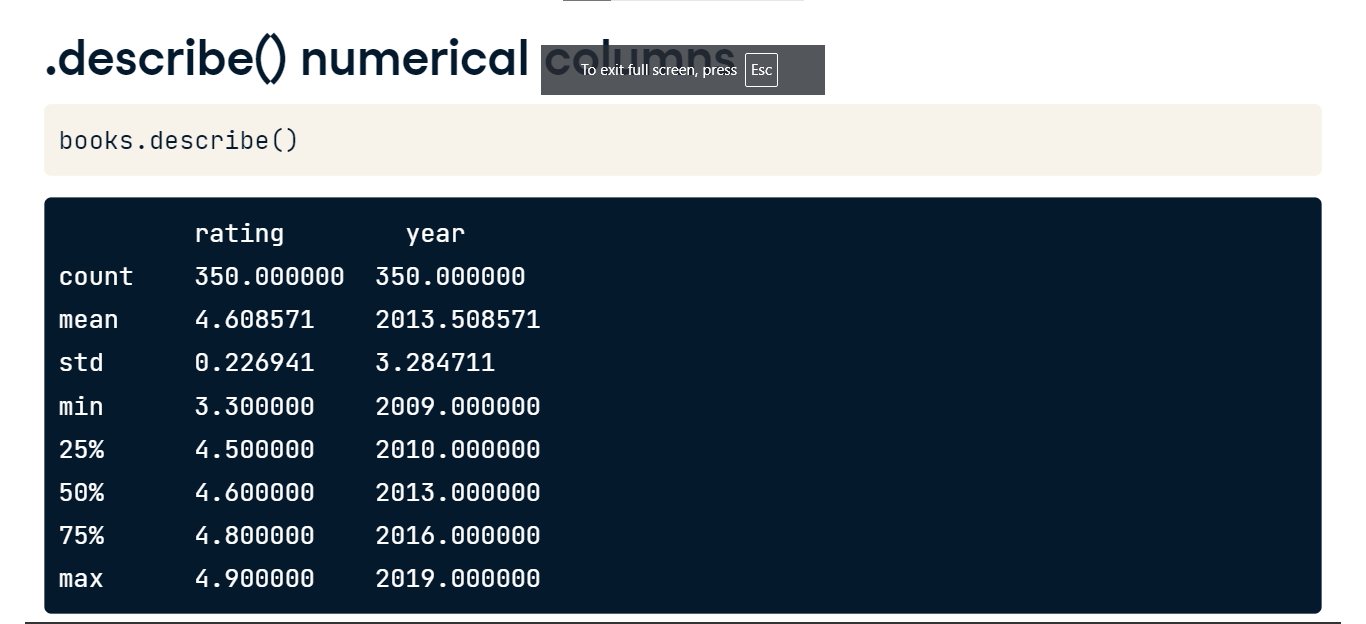
**A closer look at categorical columns**

A common question about categorical columns in a dataset is how many data points we have in each category. For example, perhaps we're interested in the genres represented in our books data. We can select the genre column and use the pandas Series method .value\_counts to find the number of books with each genre.



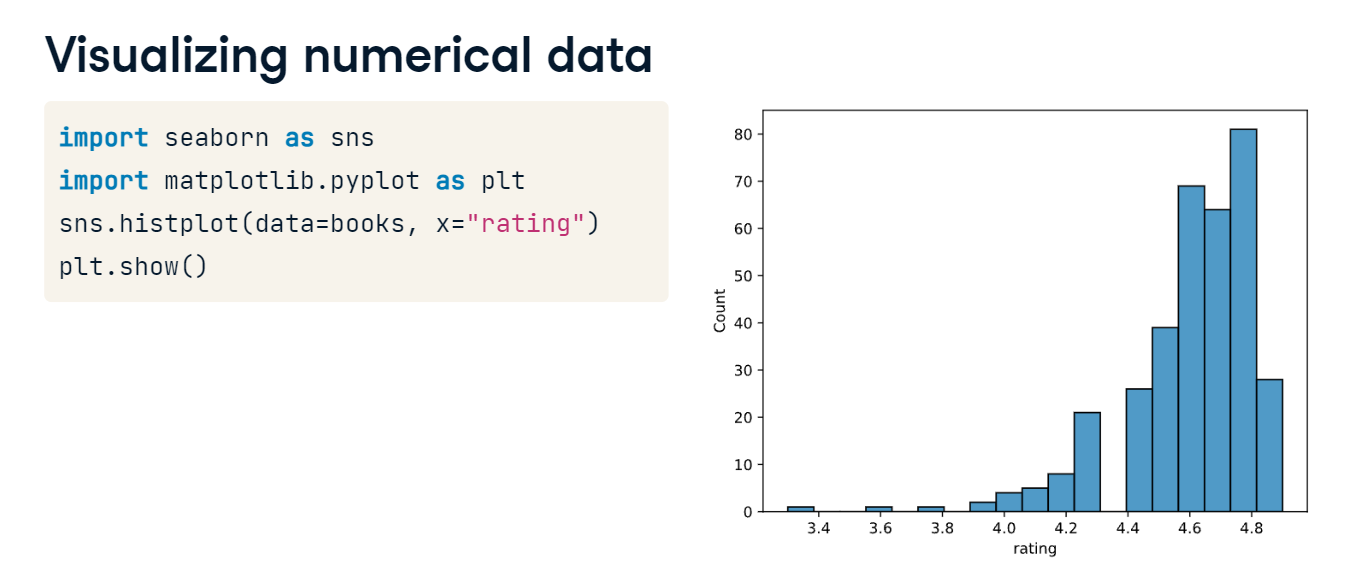
**.describe() numerical columns**

Gaining a quick understanding of data included in numerical columns is done with the help of the DataFrame.describe method. Calling .describe on books, we see that it returns the count, mean, and standard deviation of the values in each numerical column (in this case rating and year), along with the min, max, and quartile values.

****

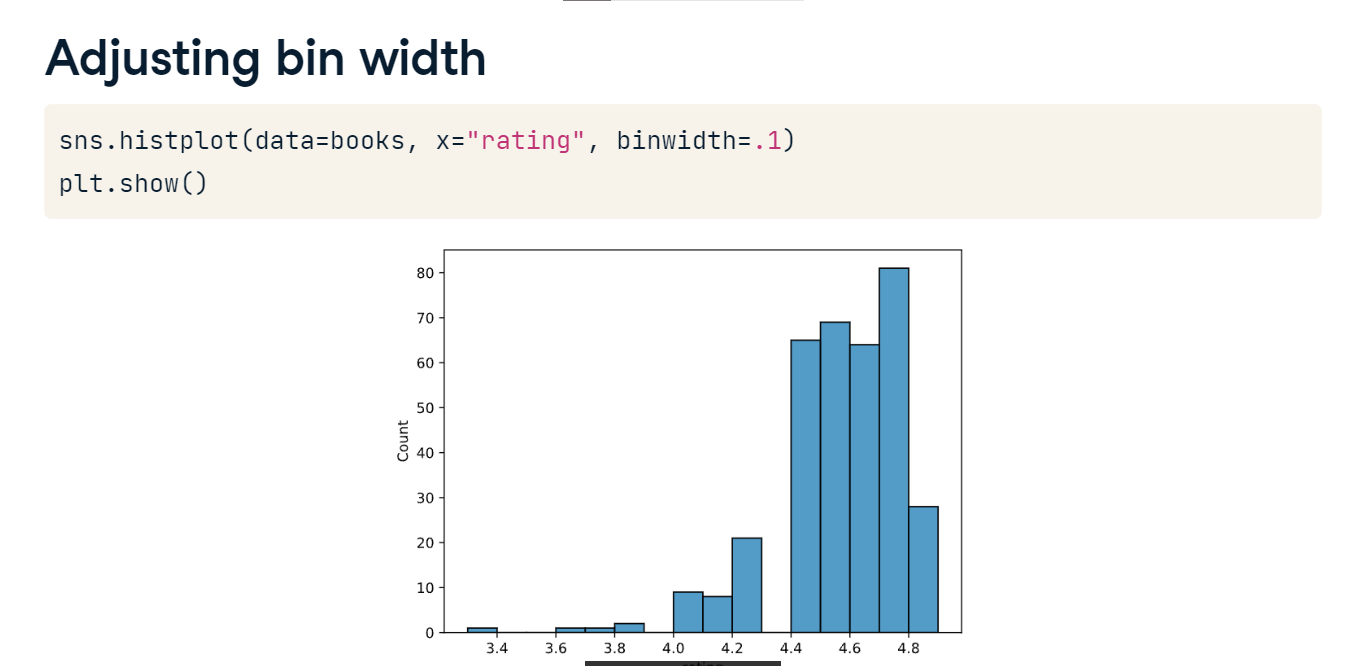
**Visualizing numerical data**

Histograms are a classic way to look at the distribution of numerical data by splitting numerical values into discrete bins and visualizing the count of values in each bin. Throughout this course, we'll use Seaborn to explore datasets visually. Seaborn is imported as s-n-s. We'll also import matplotlib.pyplot aliased as plt. To create a histogram, we'll use sns.histplot and pass the books DataFrame as the data argument. Next, we indicate which column we'd like to use as x by passing the column name rating to the x keyword argument. After running plt.show to display the plot, we see that most books received ratings above 4.4, with very few getting ratings below 4.0. However, the bin size here is a little awkward. Ideally, we would have a bin for each tenth of a rating, such as a single bin for scores greater than 4.5 to 4.6 inclusive.



**Adjusting bin width**

We can set a bin width of 0.1 using the binwidth keyword argument. That's better!



**Data validation**

Data validation is an important early step in EDA. We want to understand whether data types and ranges are as expected before we progress too far in our analysis! Let's dive in.

**Validating data types**

We learned in the last lesson that dot-info gives a quick overview of data types included in a dataset along with other information such as the number of non-missing values. We can also use the DataFrame dot-dtypes attribute if we're only interested in data types. But what if we aren't happy with these data types? Here, the year column in the books DataFrame is stored as a float, which doesn't make sense for year data, which should always be a whole number.

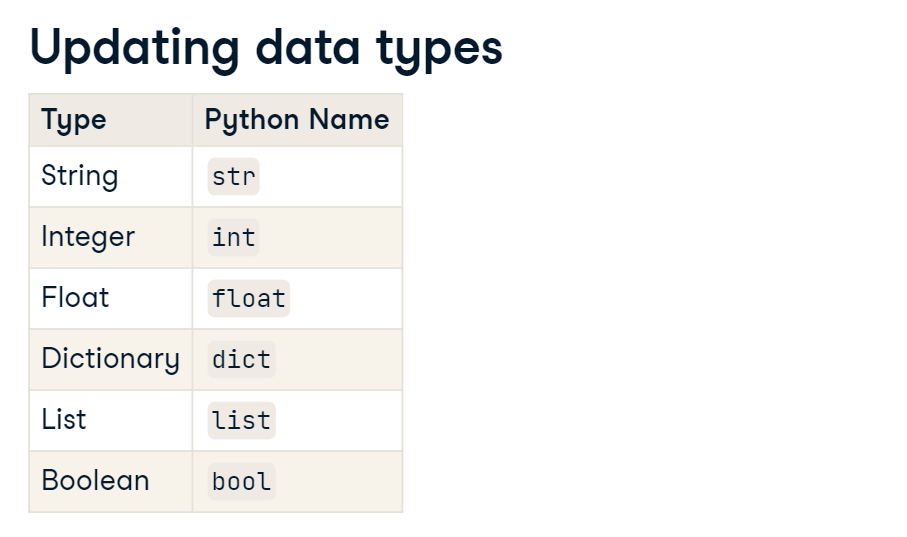
**Updating data types**

Luckily, the dot-astype function allows us to change data types without too much effort. Here, we redefine the year column by selecting the column and calling the dot-astype method, indicating we'd like to change the column to an integer. Then we use the dot-dtypes attribute to check that the year column data is now stored as integers - and it is!



**Updating data types**

Common programming data types as well as their Python names are listed here. It's the Python name that we pass to the astype function, as we did with int on the previous slide.



**Validating categorical data**

We can validate categorical data by comparing values in a column to a list of expected values using dot-isin, which can either be applied to a Series as we'll show here or to an entire DataFrame. Let's check whether the values in the genre column are limited to "Fiction" and "Non Fiction" by passing these genres as a list of strings to dot-isin. The function returns a Series of the same size and shape as the original but with True and False in place of all values, depending on whether the value from the original Series was included in the list passed to dot-isin. We can see that some values are False.



**Validating categorical data**

We can also use the tilde operator at the beginning of the code block to invert the True/ False values so that the function returns True if the value is NOT in the list passed to dot-isin.



**Validating categorical data**

And if we're interested in filtering the DataFrame for only values that are in our list, we can use the isin code we just wrote to filter using Boolean indexing!



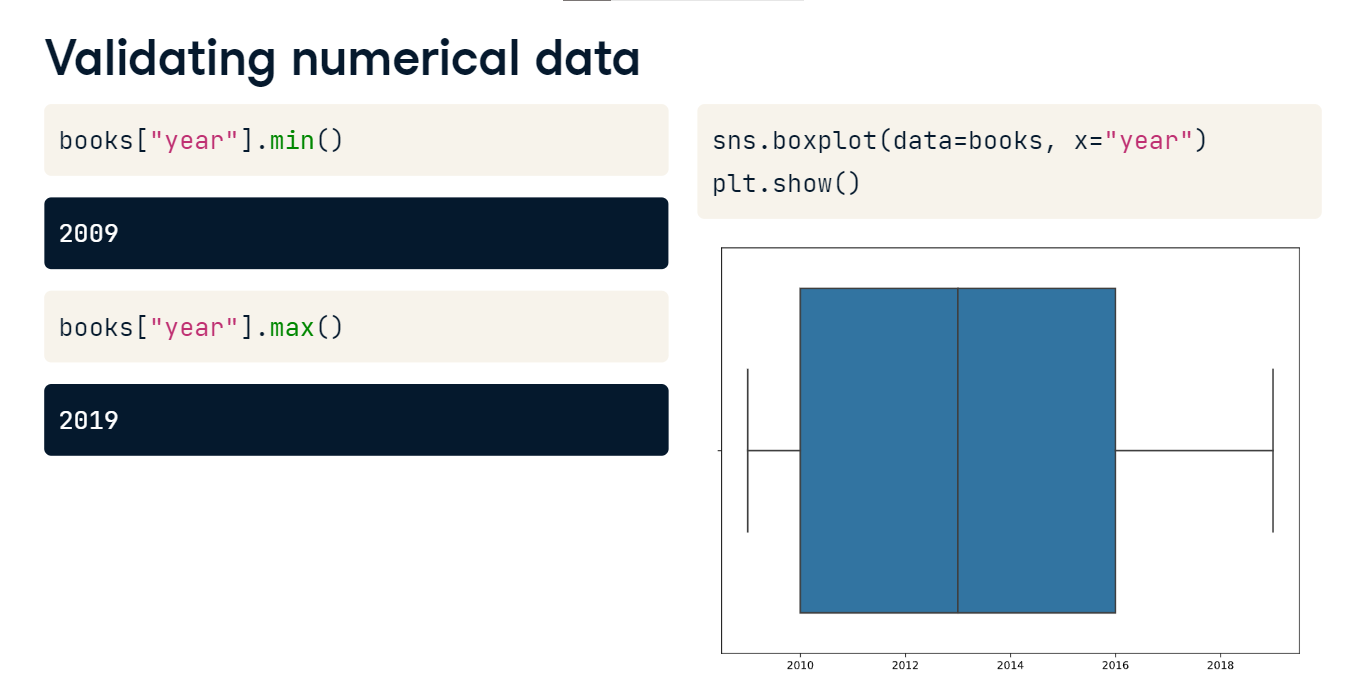
**Validating numerical data**

Let's now validate numerical data. We can select and view only the numerical columns in a DataFrame by calling the select\_dtypes method and passing "number" as the argument.



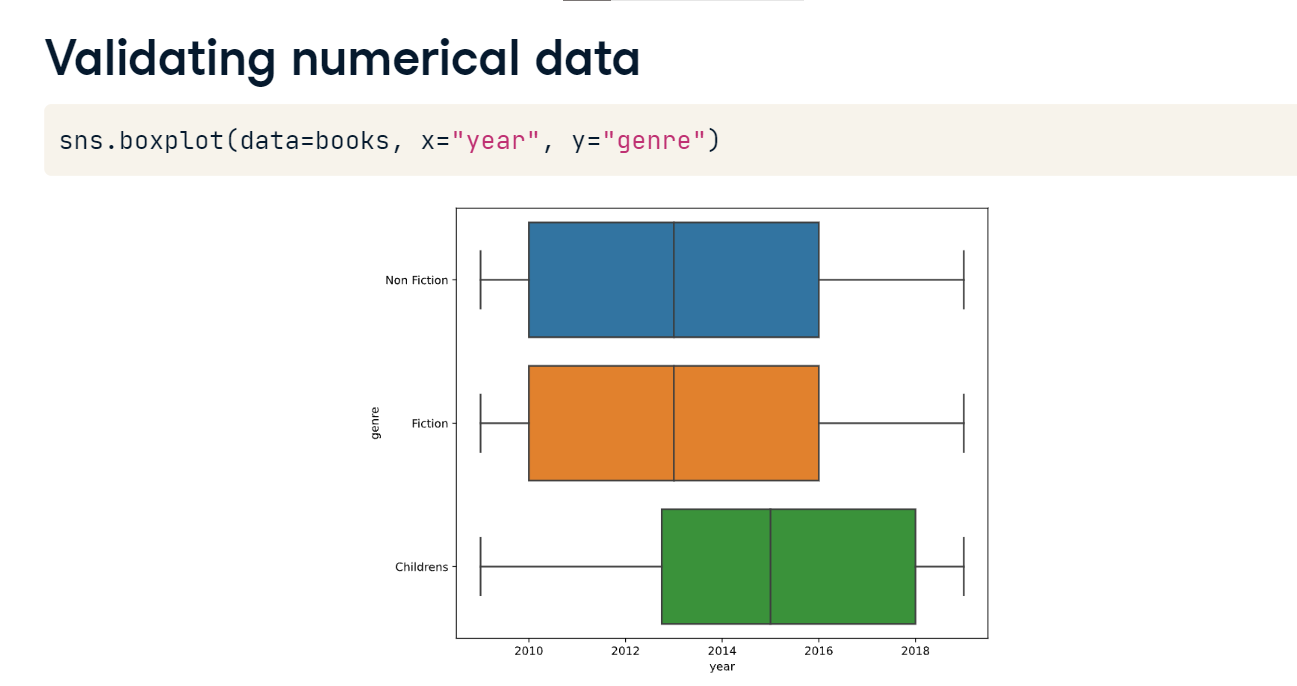
**Validating numerical data**

Perhaps we'd like to know the range of years in which the books in our dataset were published. We can check the lowest and highest years by using the dot-min and dot-max functions, respectively. And we can view a more detailed picture of the distribution of year data using Seaborn's boxplot function. The boxplot shows the boundaries of each quartile of year data: as we saw using min and max, the lowest year is 2009 and the highest year is 2019. The 25th and 75th percentiles are 2010 and 2016 and the median year is 2013.



**Validating numerical data**

We can also view the year data grouped by a categorical variable such as genre by setting the y keyword argument. It looks like the children's books in our dataset have slightly later publishing years in general, but the range of years is the same for all genres.

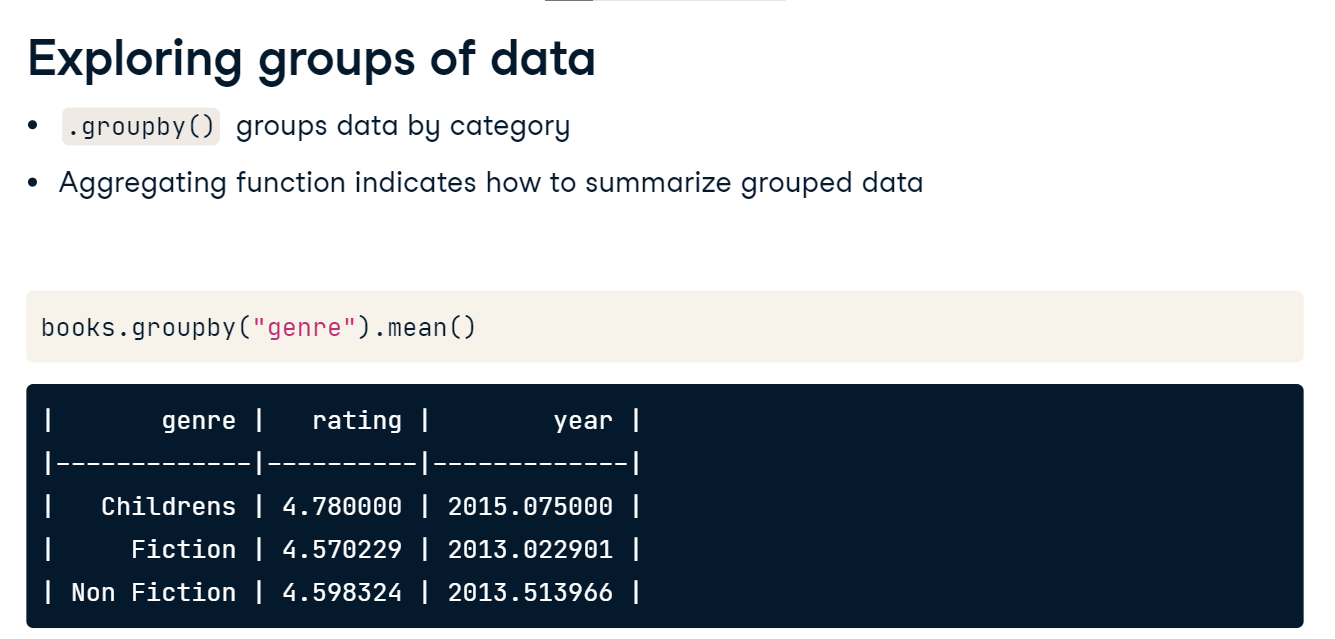


**Data summarization**

We ended the last video by exploring data by genre, noticing that children's books in our dataset have slightly later publishing years in general.

**Exploring groups of data**

We can explore the characteristics of subsets of data further with the help of the dot-groupby function, which groups data by a given category, allowing the user to chain an aggregating function like dot-mean or dot-count to describe the data within each group. For example, we can group the books data by genre by passing the genre column name to the groupby function. Then, we chain an aggregating function, in this case, dot-mean, to find the mean value of the numerical columns for each genre. The results show that children's books have a higher average rating than other genres.



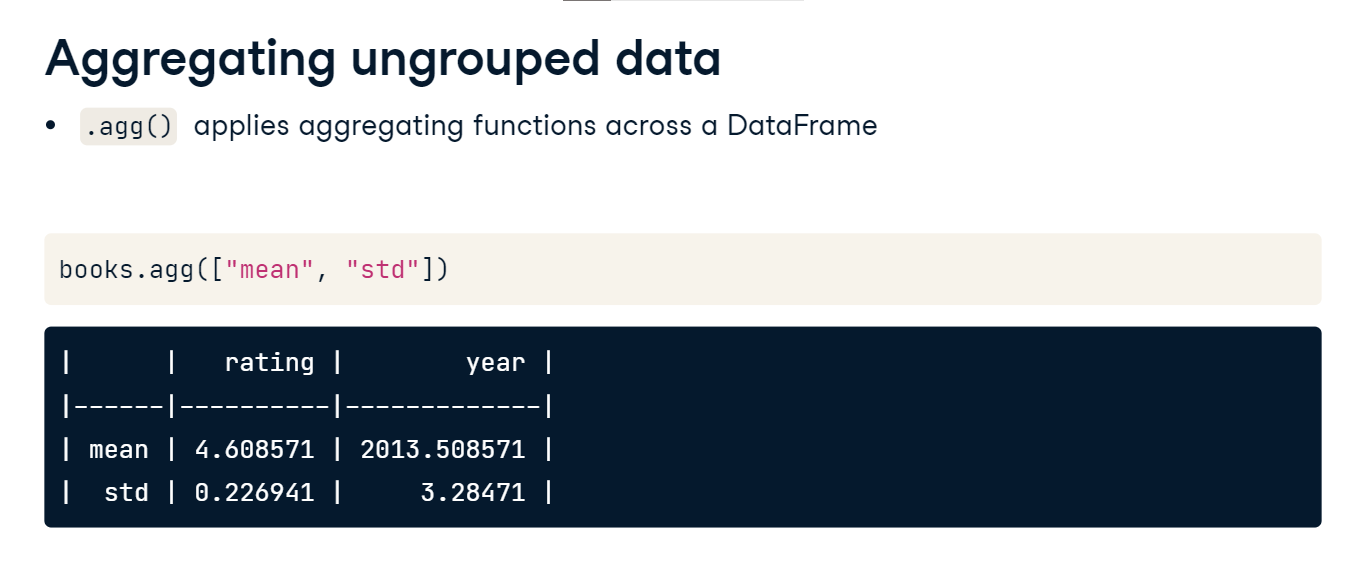
**Aggregating functions**

Other aggregating functions that are useful to chain with dot-groupby are dot-sum, dot-count, dot-min, dot-max, dot-var, which returns the variance, and dot-std, which returns the standard deviation.



**Aggregating ungrouped data**

The dot-agg function, short for aggregate, allows us to apply aggregating functions. By default, it aggregates data across all rows in a given column and is typically used when we want to apply more than one function. Here, we apply dot-agg to the books DataFrame and pass a list of aggregating functions to apply: dot-mean and dot-std. Our code returns a DataFrame of aggregated results, and dot-agg applies these functions only to numeric columns; the rating and year columns in the books DataFrame.



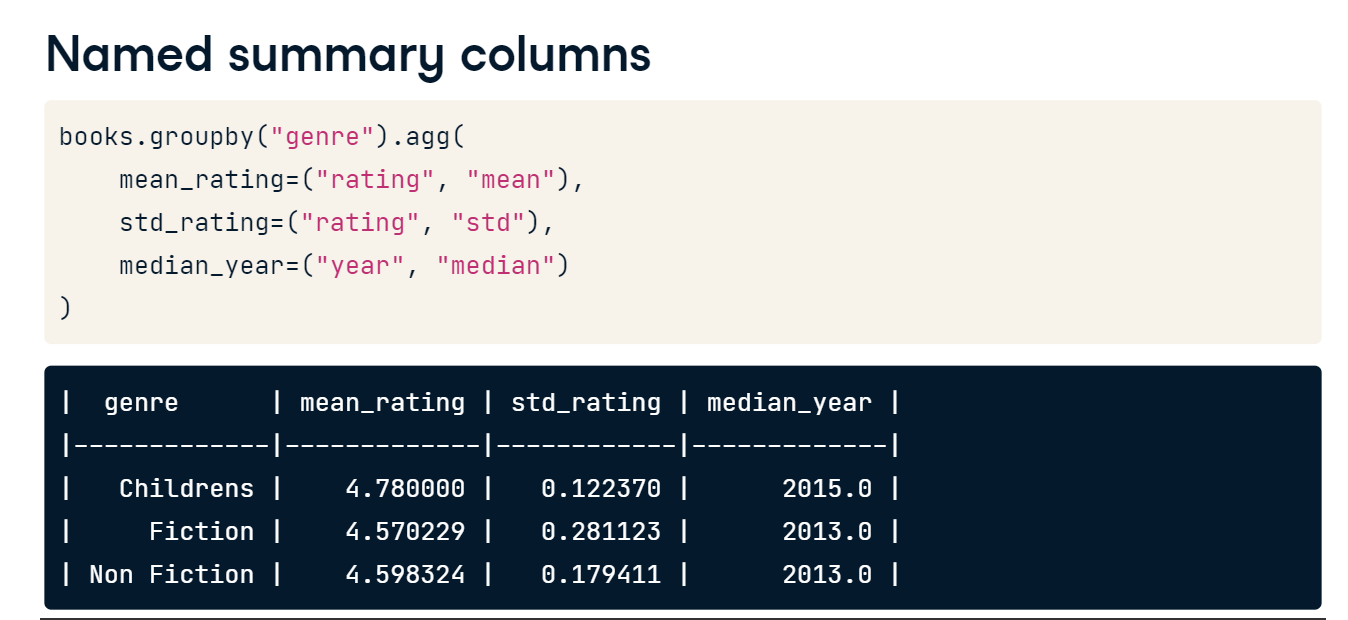
**Specifying aggregations for columns**

We can even use a dictionary to specify which aggregation functions to apply to which columns. The keys in the dictionary are the columns to apply the aggregation, and each value is a list of the specific aggregating functions to apply to that column.



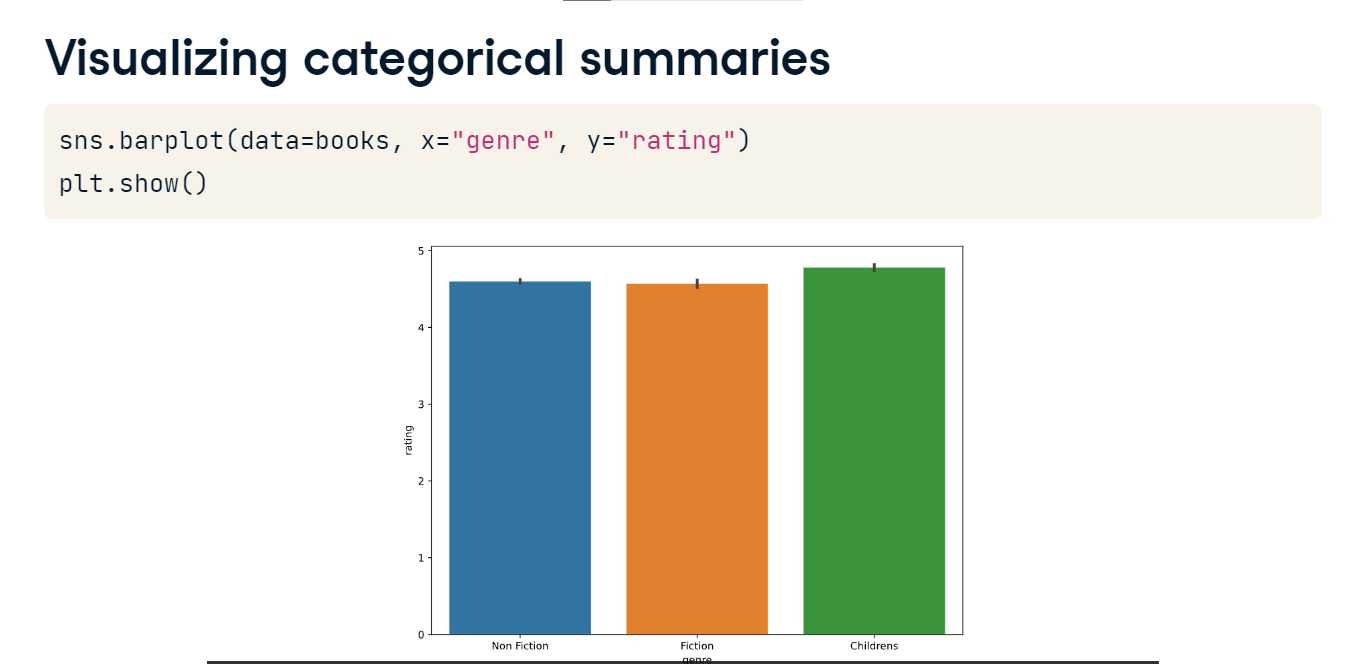
**Named summary columns**

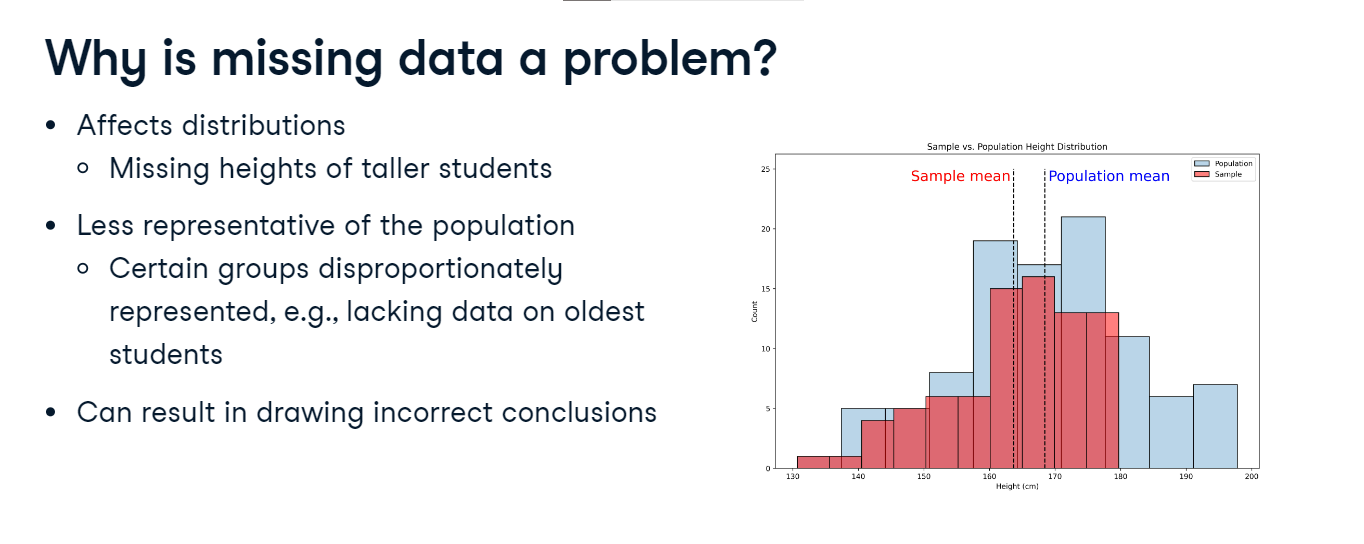
By combining dot-agg and dot-groupby, we can apply these new exploration skills to grouped data. Maybe we'd like to show the mean and standard deviation of rating for each book genre along with the median year. We can create named columns with our desired aggregations by using the dot-agg function and creating named tuples inside it. Each named tuple should include a column name followed by the aggregating function to apply to that column. The name of the tuple becomes the name of the resulting column. Now, we can get two summary values of interest about ratings and our year data looks cleaner! We can see that the Fiction genre has the lowest average rating as well as the largest variation in ratings.



**Visualizing categorical summaries**

We can display similar information visually using a barplot. In Seaborn, bar plots will automatically calculate the mean of a quantitative variable like rating across grouped categorical data, such as the genre category we've been looking at. In Seaborn, bar plots also show a 95% confidence interval for the mean as a vertical line on the top of each bar. Here, we pass the genre column as the x values and the rating column as the y values. The results reinforce what we saw in the last slide: while Fiction books have the lowest rating, their ratings also have a little more variation.



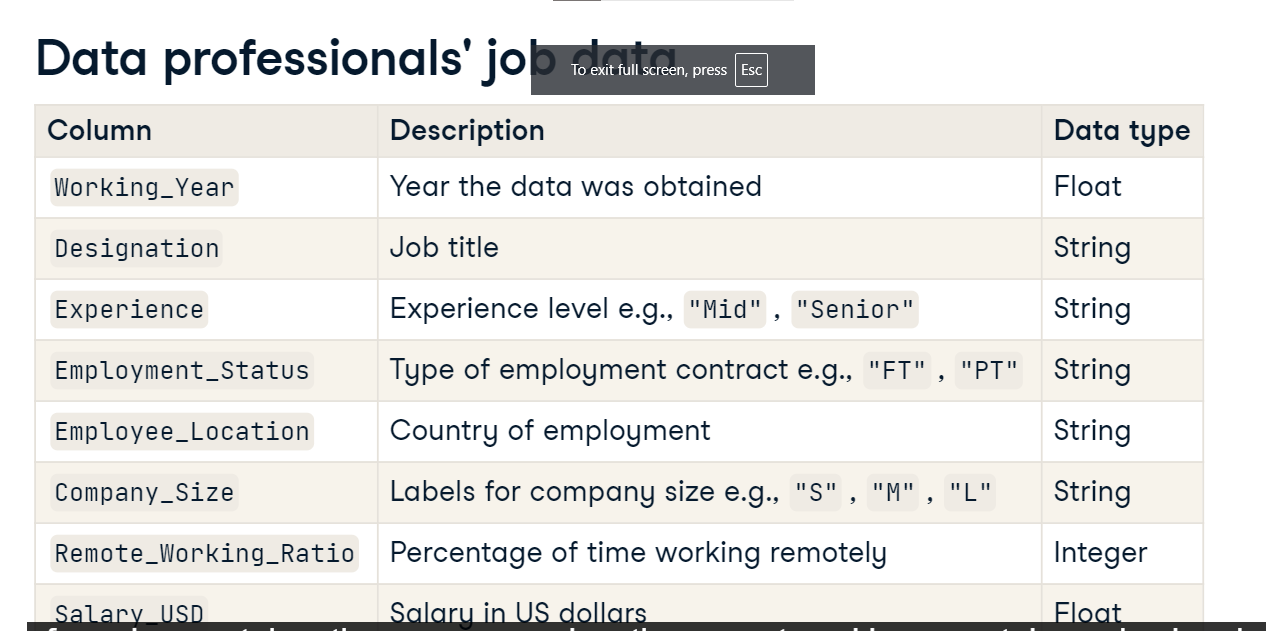


**Why is missing data a problem?**

So, why is it important to deal with missing data? Well, it can affect distributions. As an example, we collect the heights of students at a high school. If we fail to collect the heights of the oldest students, who were taller than most of our sample, then our sample mean will be lower than the population mean. Put another way, our data is less representative of the underlying population. In this case, parts of our population aren't proportionately represented. This misrepresentation can lead us to draw incorrect conclusions, like thinking that, on average, students are shorter than they really are.

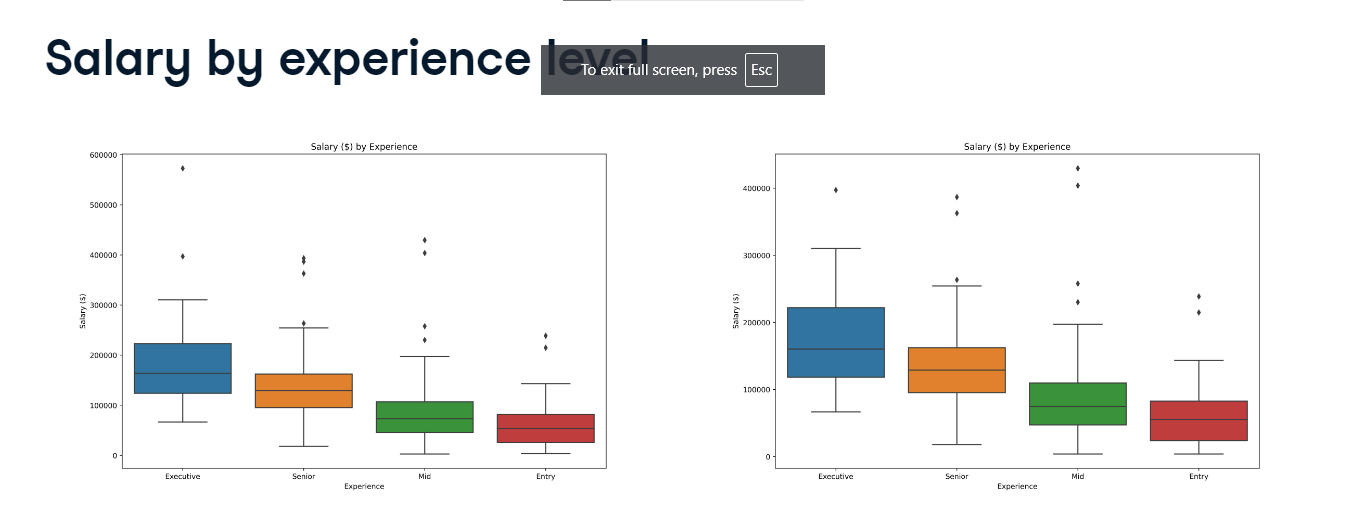
**Data professionals' job data**

Let's illustrate how missing data impacts exploratory analysis using a dataset about data professionals. This dataset includes the year the data was obtained, job title, experience level, type of employment, location, company size, time spent working remotely, and salary in US dollars.



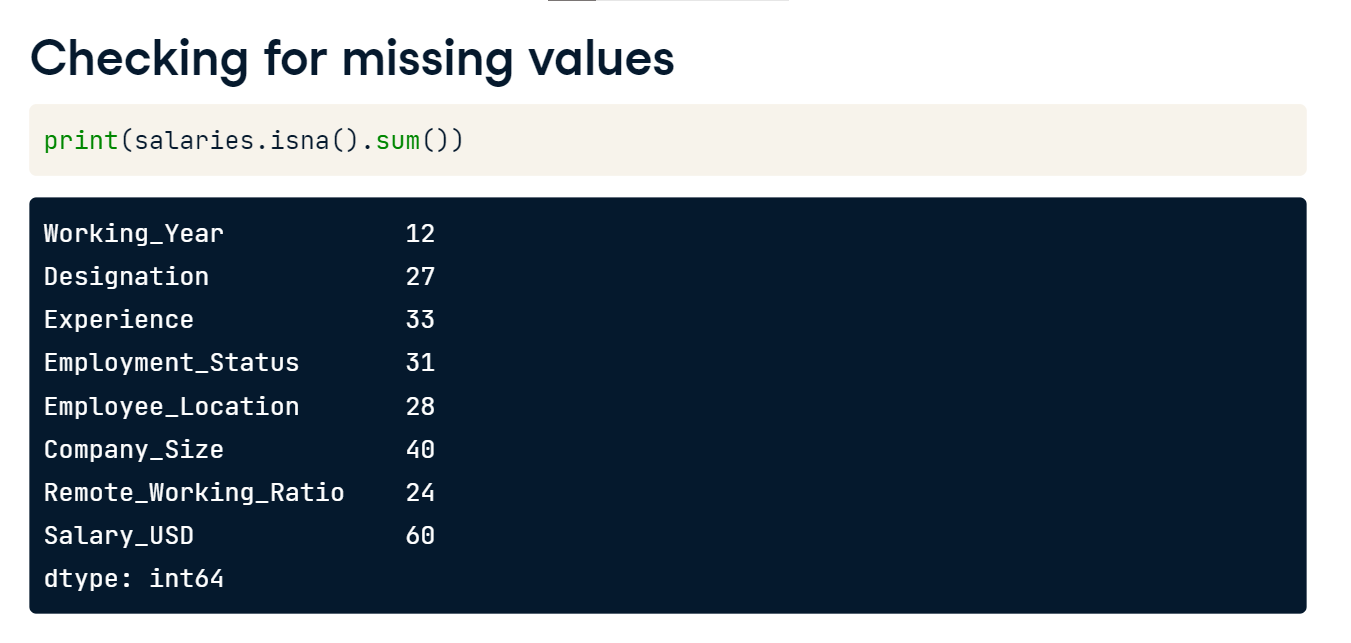
**Salary by experience level**

To highlight the impact of missing values, let's look at salaries by experience level using a full version of the dataset. Now, let's compare this to the same data with some missing values. The y-axis shows that the largest salary is around 150000 dollars less in the second plot!



**Checking for missing values**

With our dataset stored as a pandas DataFrame called salaries, we can count the number of missing values per column by chaining the dot-isna and dot-sum methods. isna refers to the fact that missing values are represented as na in DataFrames. The output shows all columns contain missing values, with Salary\_USD missing 60 values.



**Strategies for addressing missing data**

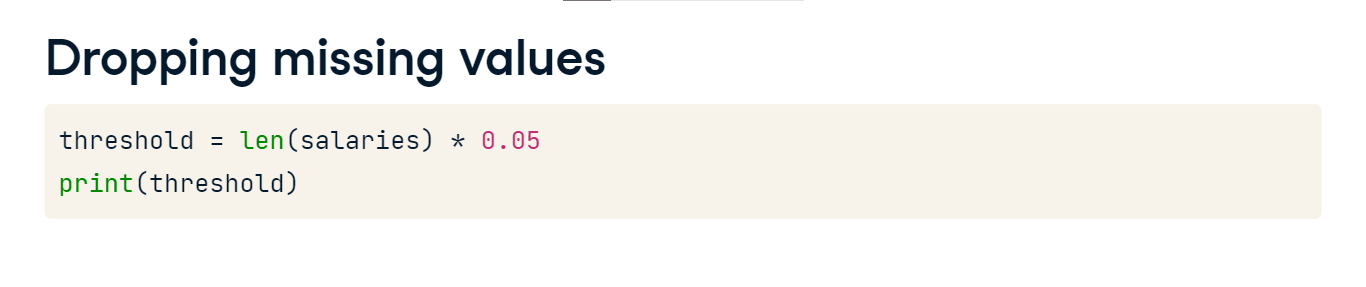
02:01 - 02:38

There are various approaches to handle missing data. One rule of thumb is to remove observations if they amount to five percent or less of all values. If we have more missing values, instead of dropping them, we can replace them with a summary statistic like the mean, median, or mode, depending on the context. This is known as imputation. Alternatively, we can impute by sub-groups. We saw that median salary varies by experience, so we could impute different salaries depending on experience.



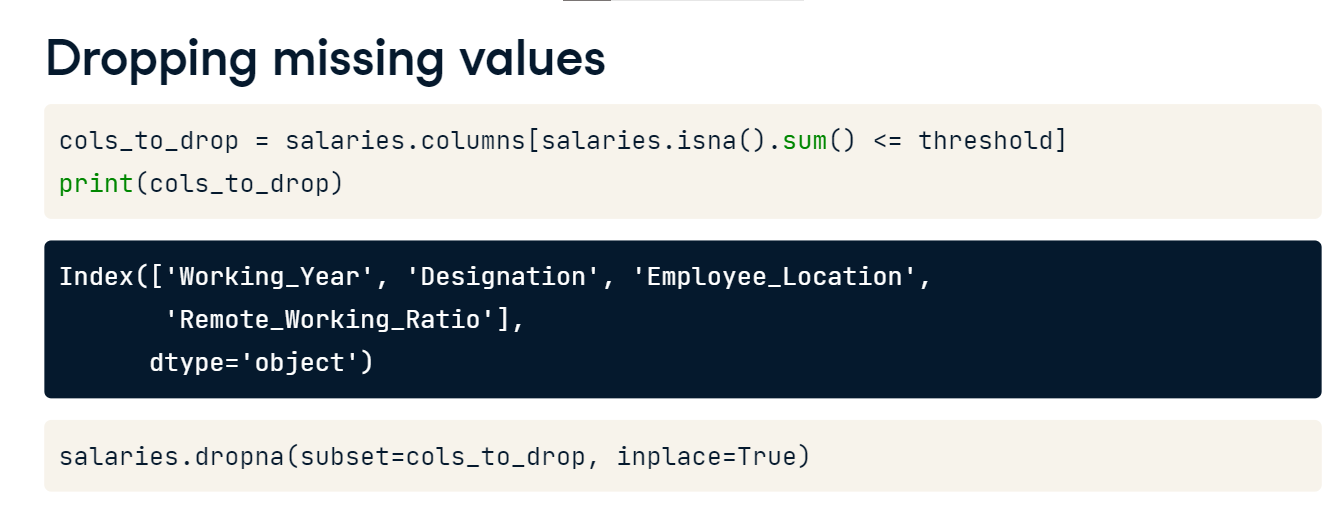
**Dropping missing values**

To calculate our missing values threshold we multiply the length of our DataFrame by five percent, giving us an upper limit of 30.



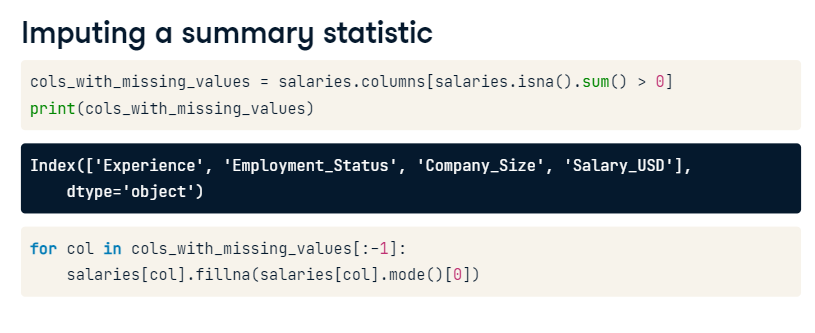
**Dropping missing values**

We can use Boolean indexing to filter for columns with missing values less than or equal to this threshold, storing them as a variable called cols\_to\_drop. Printing cols\_to\_drop shows four columns. We drop missing values by calling dot-dropna, passing cols\_to\_drop to the subset argument. We set inplace to True so the DataFrame is updated.



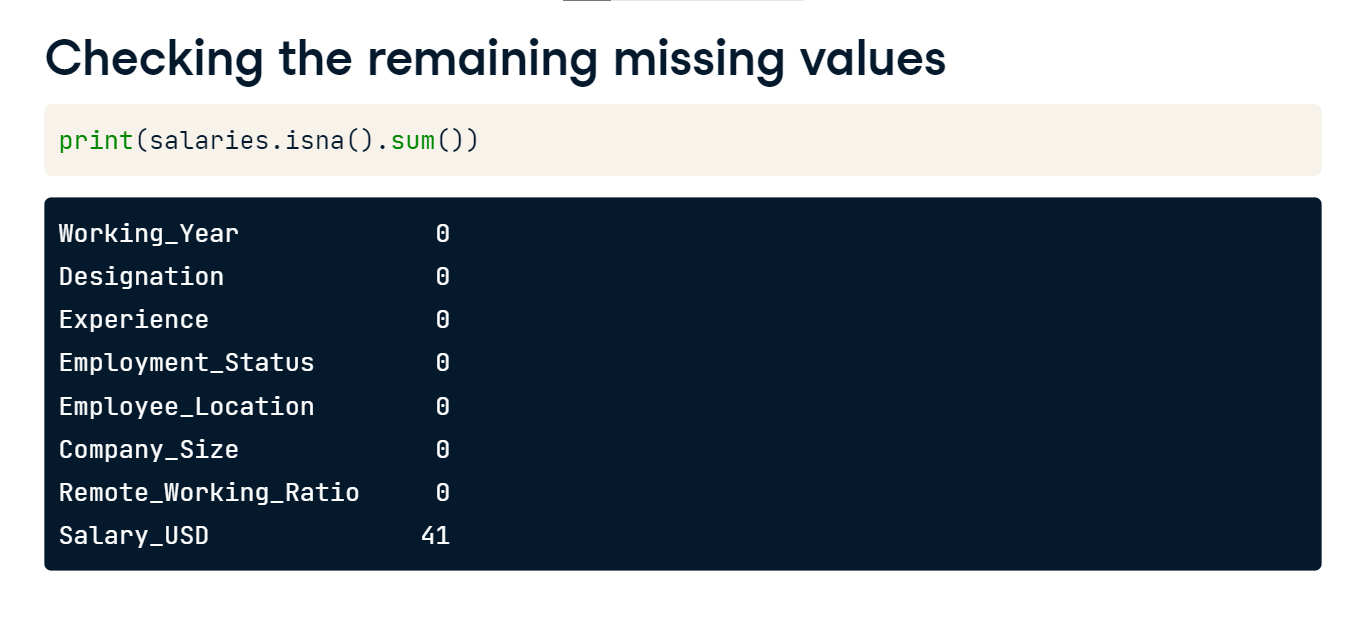
**Imputing a summary statistic**

We then filter for the remaining columns with missing values, giving us four columns. To impute the mode for the first three columns, we loop through them and call the dot-fillna method, passing the respective column's mode and indexing the first item, which contains the mode, in square brackets.



**Checking the remaining missing values**

Checking for missing values again, we see salary\_USD is now the only column with missing values and the volume has changed from 60 missing values to 41. This is because some rows may have contained missing values for our subset columns as well as salary, so they were dropped.



**Imputing by sub-group**

We'll impute median salary by experience level by grouping salaries by experience and calculating the median. We use the dot-to-dict method, storing the grouped data as a dictionary. Printing the dictionary returns the median salary for each experience level, with executives earning the big bucks!

We then impute using the dot-fillna method, providing the Experience column and calling the dot-map method, inside which we pass the salaries dictionary.





**Converting and analyzing categorical data**

Now let's explore how to create and analyze categorical data.

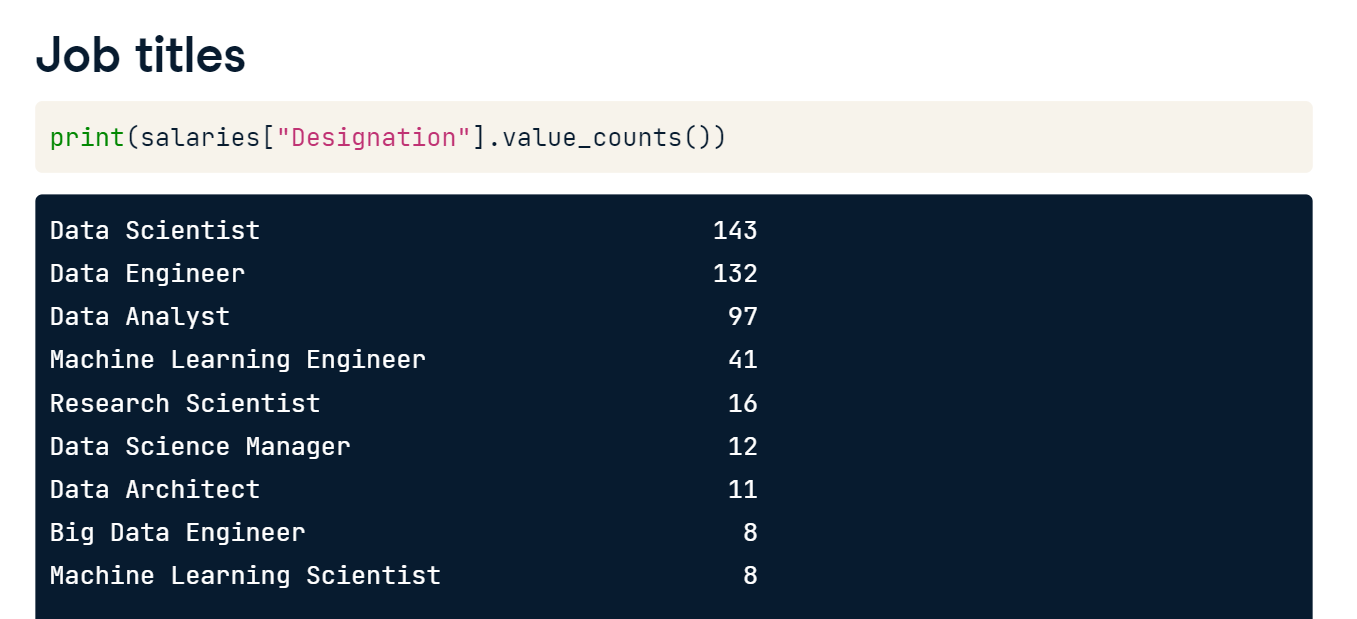
**Previewing the data**

Recall that we can use the select\_dtypes method to filter any non-numeric data. Chaining dot-head allows us to preview these columns in our salaries DataFrame, showing columns such as Designation, Experience, Employment\_Status, and Company\_Size.



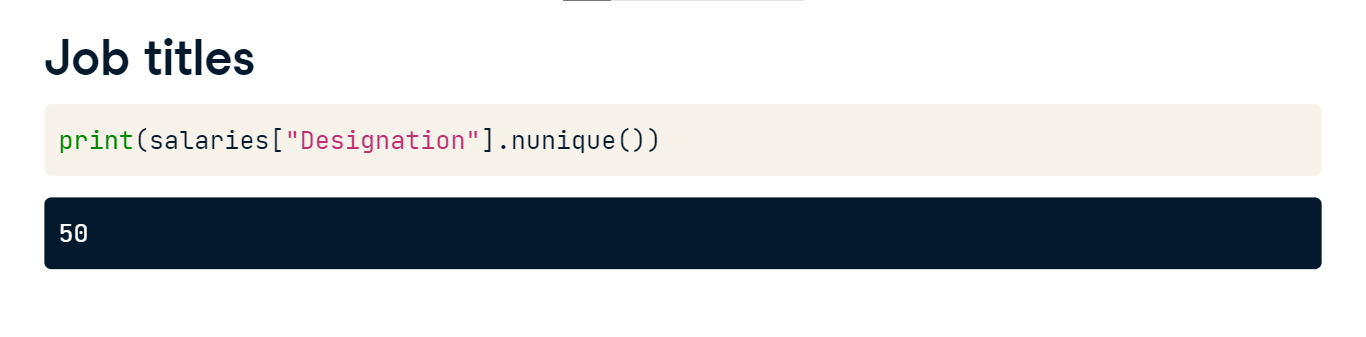
**Job titles**

Let's examine frequency of values in the Designation column. The output is truncated by pandas automatically since there are so many different job titles!



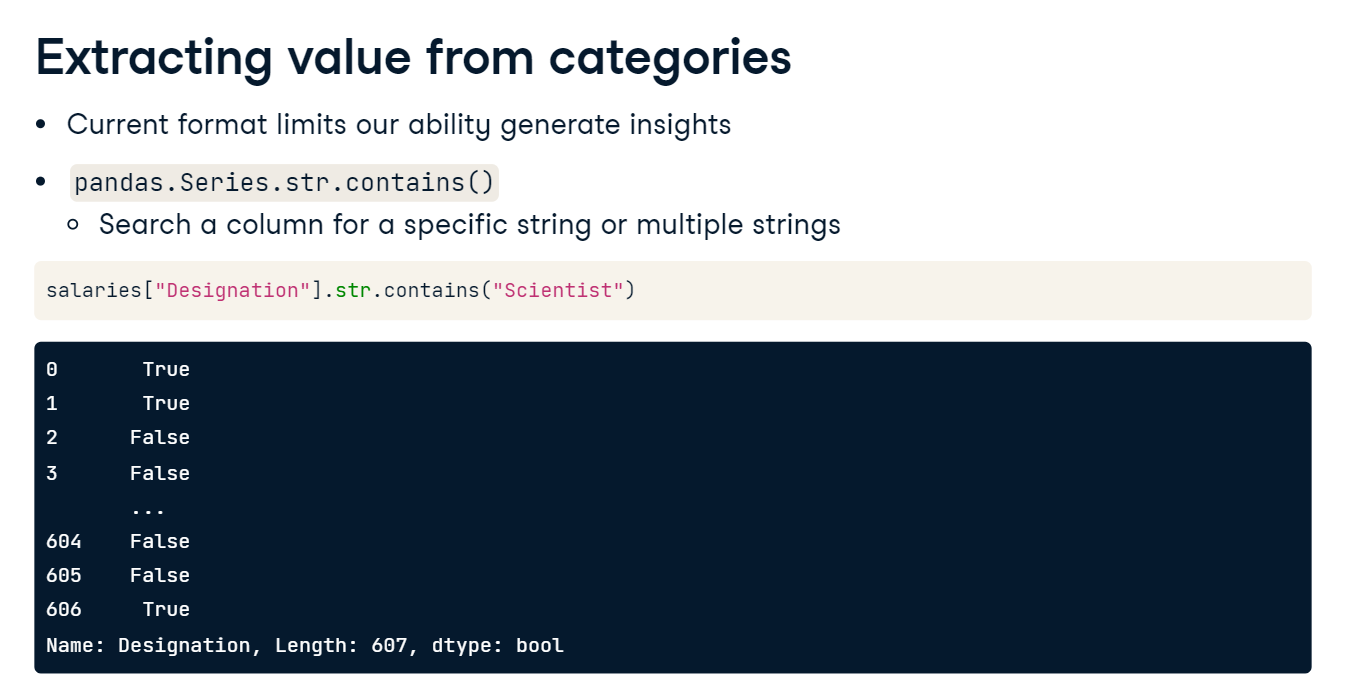
We can count how many unique job titles there are using pandas dot-nunique method. There are 50 in total!

However, the fifth most popular job title, Research Scientist, appears less than 20 times.



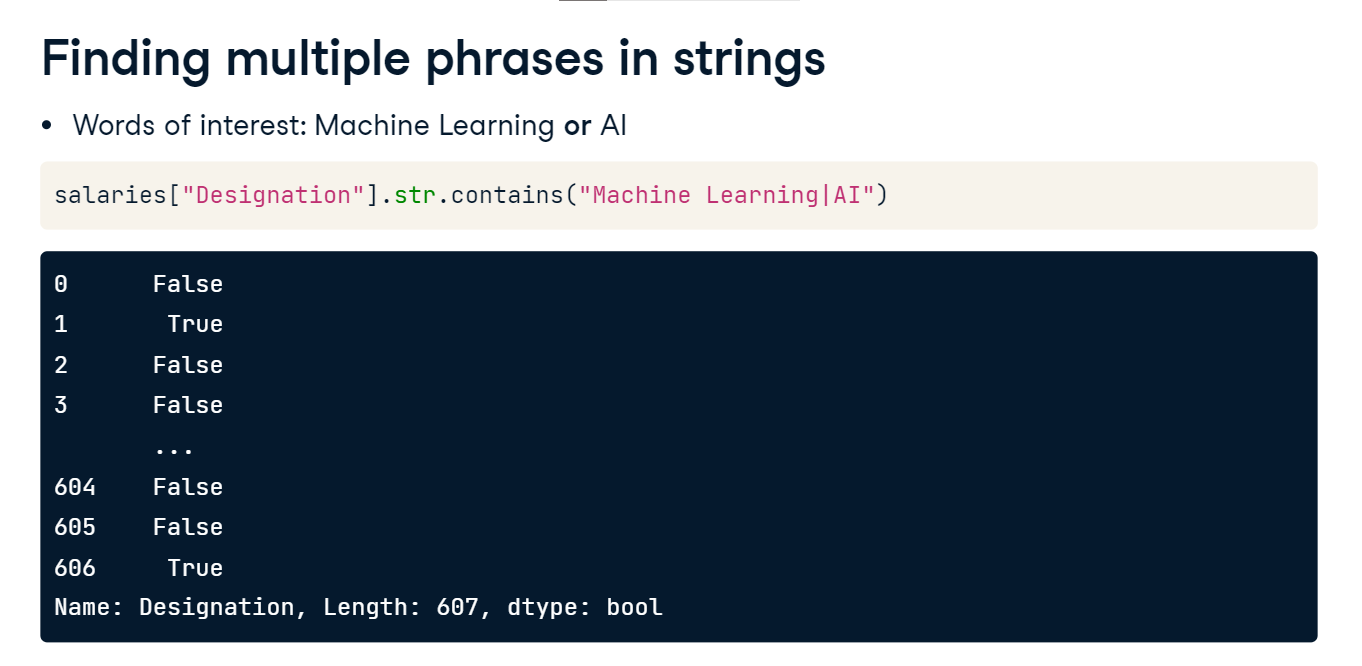
**Extracting value from categories**

The current format of the data limits our ability to generate insights. We can use the pandas series-dot-string-dot-contains method, which allows us to search a column for a specific string or multiple strings. Say we want to know which job titles have Scientist in them. We use the string-dot-contains method on the Designation column, passing the word Scientist. This returns True or False values depending on whether the row contains this word.



**Finding multiple phrases in strings**

What if we want to filter for rows containing one or more phrases? Say we want to find job titles containing either Machine Learning or AI. We use the string-dot-contains method again, but this time we include a pipe between our two phrases. This will return True if an observation in the Designation column contains Machine Learning or AI, or false if neither of these phrases are present! Notice that we avoid spaces before or after the pipe - if we included spaces then string-dot-contains will only capture values that have a space, which isn't necessary for us in this case. Again we are returned the Boolean results.



**Finding multiple phrases in strings**

What if we wanted to filter for job titles that start with a specific phrase such as "Data"? We use the same string-dot-contains method and include the caret symbol to indicate we are looking for this match at the start of the line. This will match titles such as "Data Scientist" but not "Big Data Engineer".



Now we have a sense of how this method works, let's define a list of job titles we want to find. We start by creating a list with the different categories of data roles, which will become the values of a new column in our DataFrame.

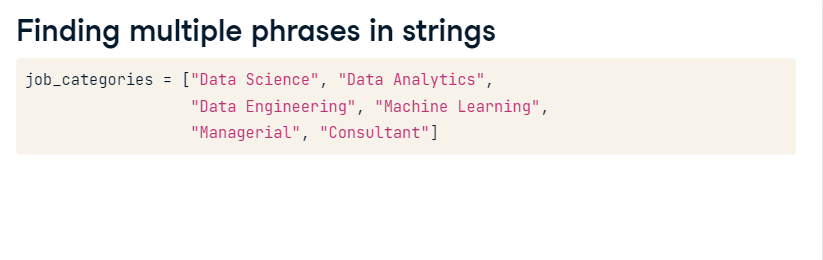
**Finding multiple phrases in strings**

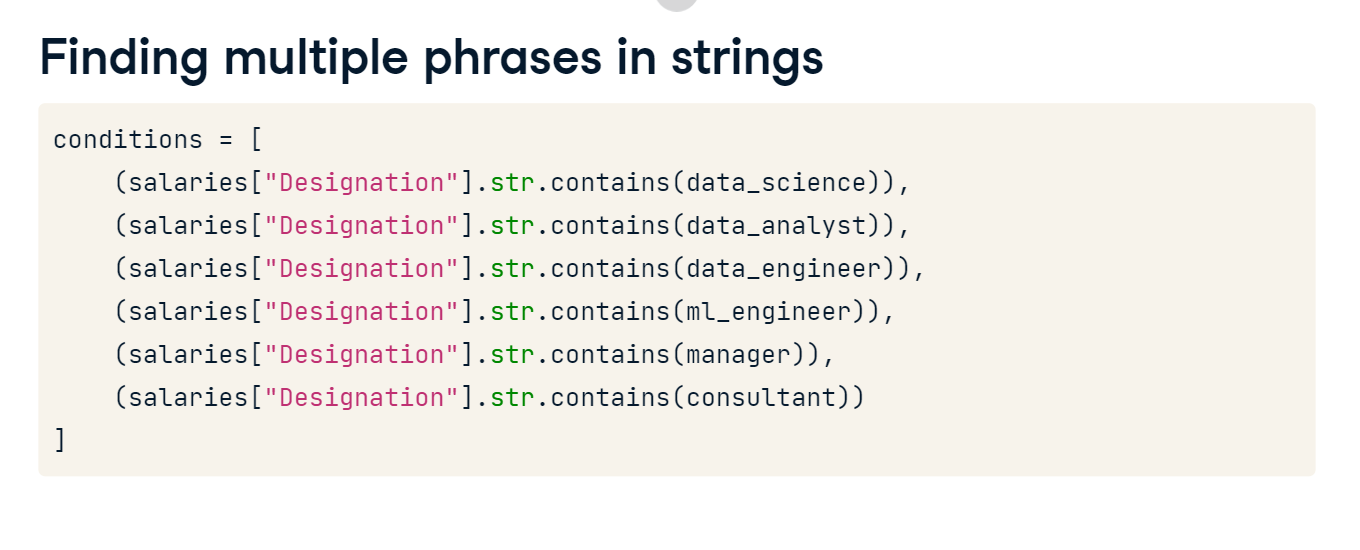
We then need to create variables containing our filters. We will look for Data Scientist or NLP for data science roles. We'll use Analyst or Analytics for data analyst roles. We repeat this for data engineer, machine learning engineer, managerial, and consultant roles.



**Finding multiple phrases in strings**

The next step is to create a list with our range of conditions for the string-dot-contains method. We add data science, data analyst, data engineer, and all remaining roles, remembering to close our list.





**Creating the categorical column**

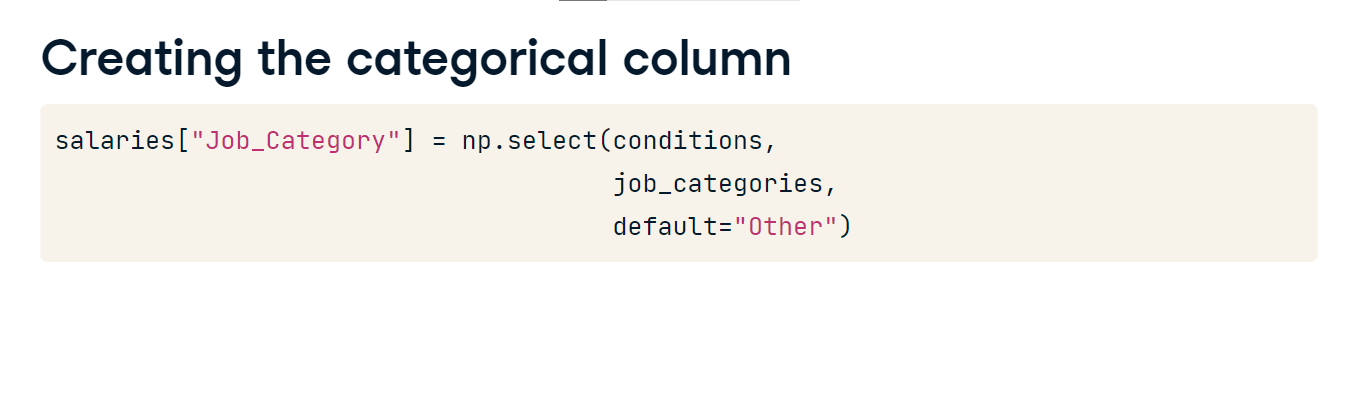
Finally, we can create our new Job\_Category column by using NumPy's dot-select function.

It takes a list of conditions as the first argument,

followed by a list of arrays to search for the conditions in.

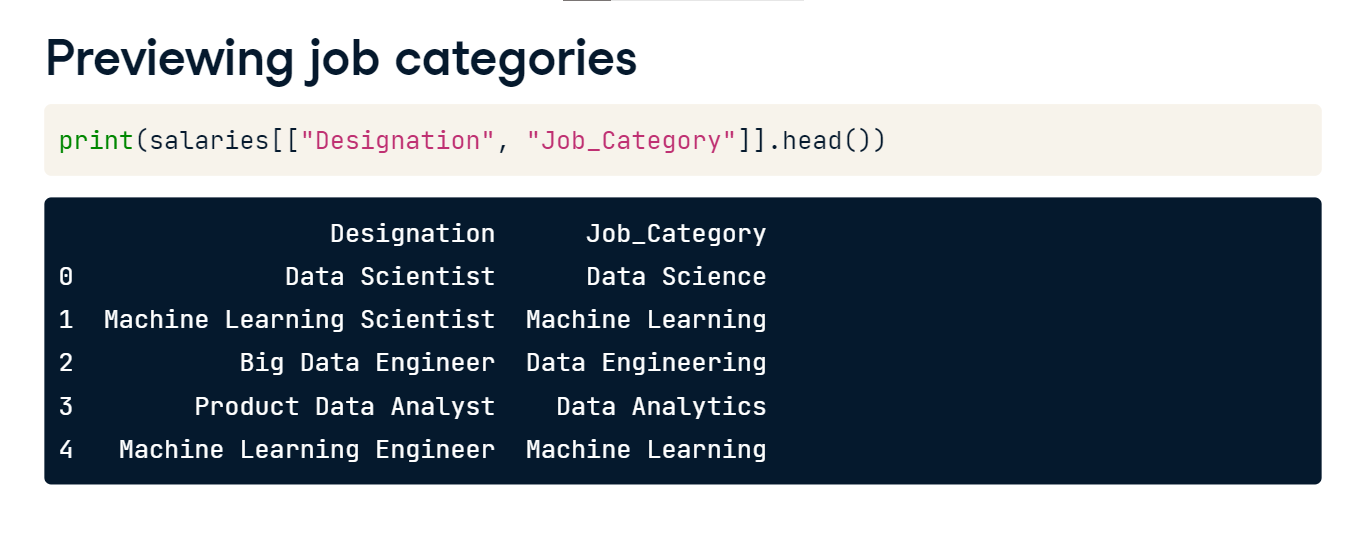
**Creating the categorical column**

By using an argument called default, we tell NumPy to assign "Other" when a value in our conditions list is not found.



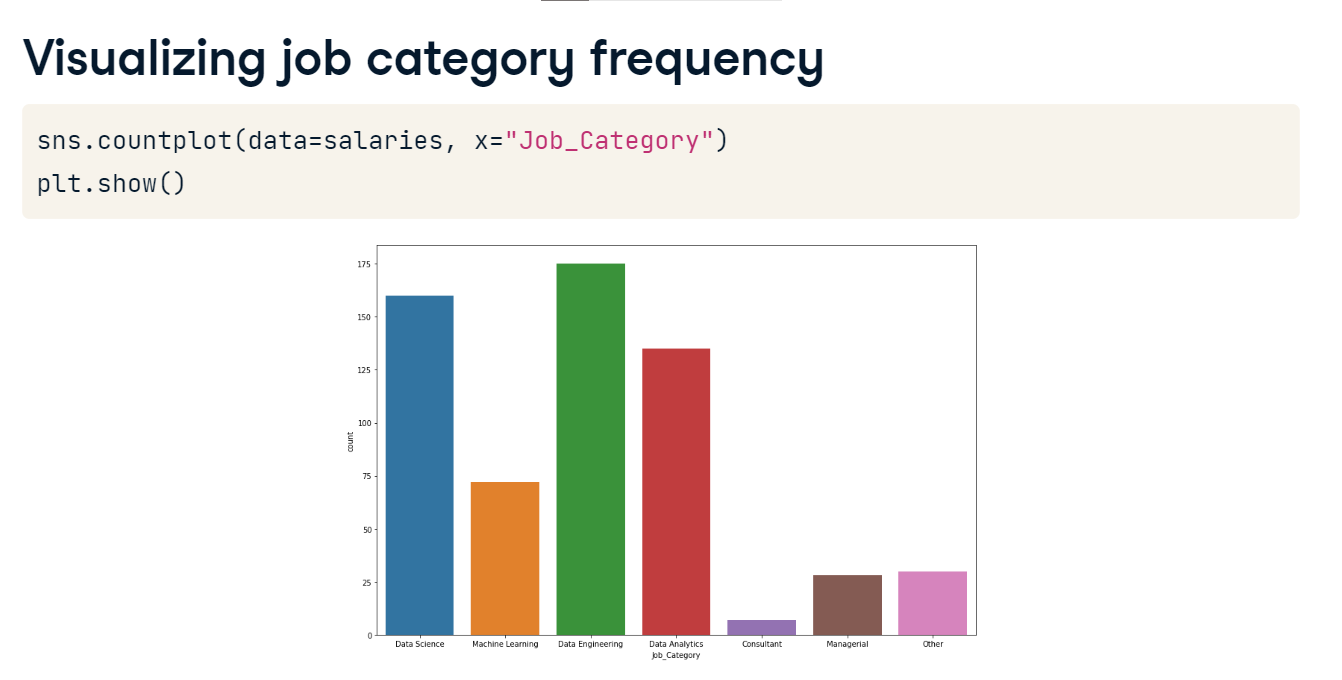
**Previewing job categories**

Previewing the Designation and our new Job\_Category columns, we can sense check the first five values. All looks good!



**Visualizing job category frequency**

With our new column, we can visualize how many jobs fall under each category. For this, we use Seaborn's countplot, passing our DataFrame to the data keyword argument and the Job\_Category column to x. We call p-l-t-dot-show to display the plot. We can see Data Science, Engineer, and Analyst roles are by far the most popular! There aren't many roles categorized as Other, suggesting we captured the majority of our data roles appropriately!



**Working with numeric data**

Time to switch our focus on to working with numeric data.

**The original salaries dataset**

So far, we've been looking at a modified version of the data professionals dataset. Let's print summary information about our original DataFrame.

The first thing that jumps out is that the Salary\_USD column we've been working with is not present, but there's a column called Salary\_In\_Rupees, referring to India's currency.

**Salary in rupees**

Previewing this column, we see that the values contain commas, and the data type is object.

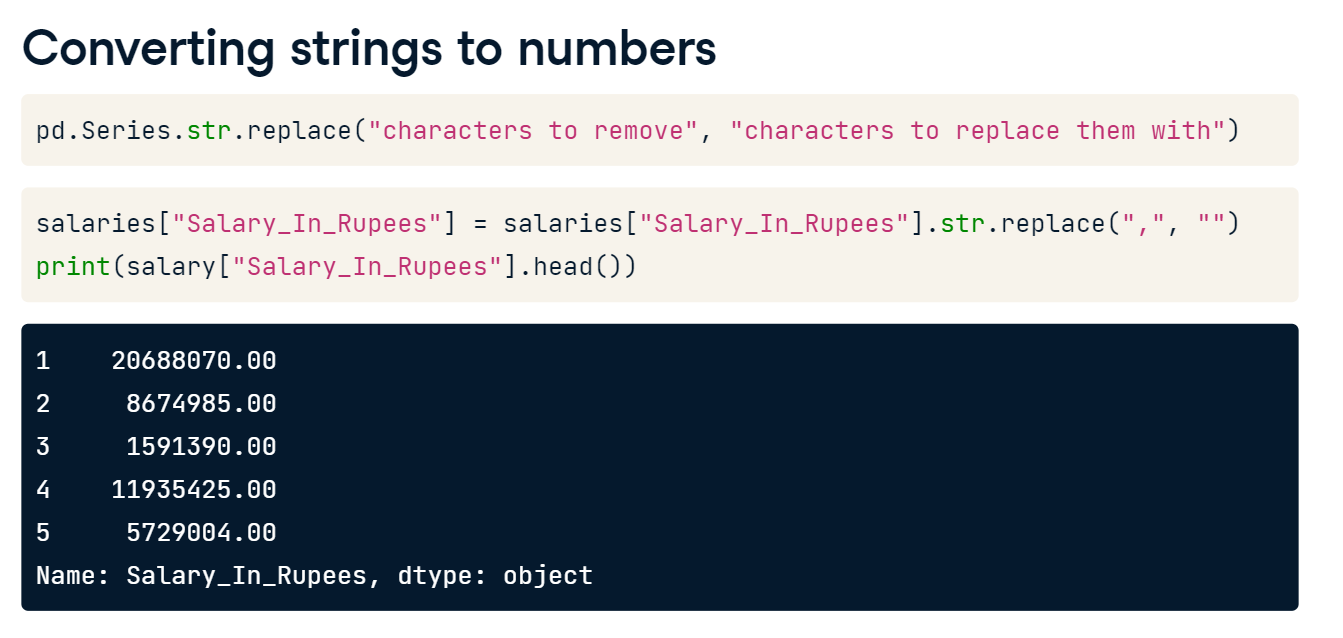


**Converting strings to numbers**

To obtain Salary in USD we'll need to perform a few tasks. First, we need to remove the commas from the values in the Salary\_In\_Rupees column. Next, we change the data type to float. Lastly, we'll make a new column by converting the currency.

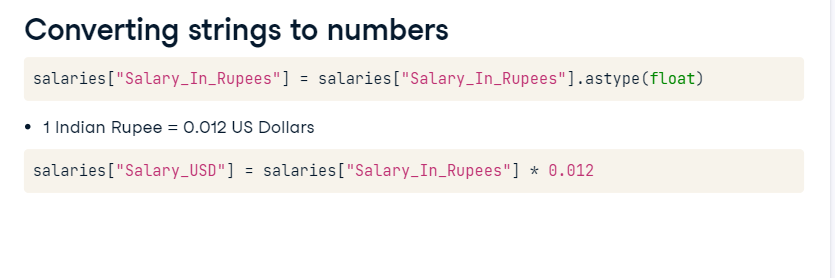
**Converting strings to numbers**

To remove commas, we can use the pandas Series-dot-string-dot-replace method. We first pass the characters we want to remove, followed by the characters to replace them with. As we don't want to add characters back in, when we update the column we provide an empty string in this part of the method. Printing the first five rows of this column, we see the commas have been removed. However, the column is still object data type.



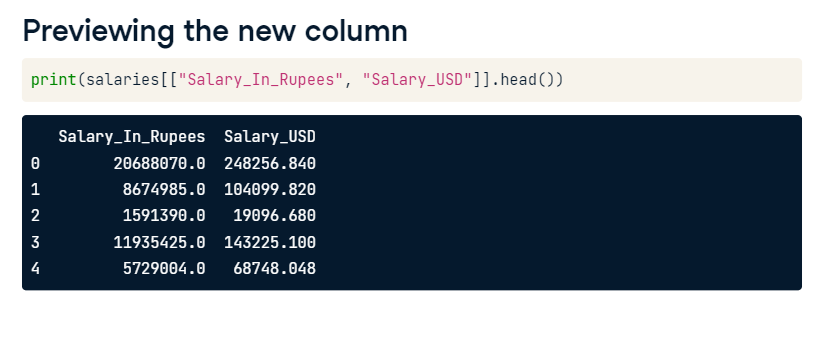
**Converting strings to numbers**

We update the data type to float. We've looked up the conversation rate from Indian rupees to US dollars, and currently one rupee is worth one-point-two cents. To create the Salary\_USD column we multiply the values in the rupees column by zero-point-zero-one-two.



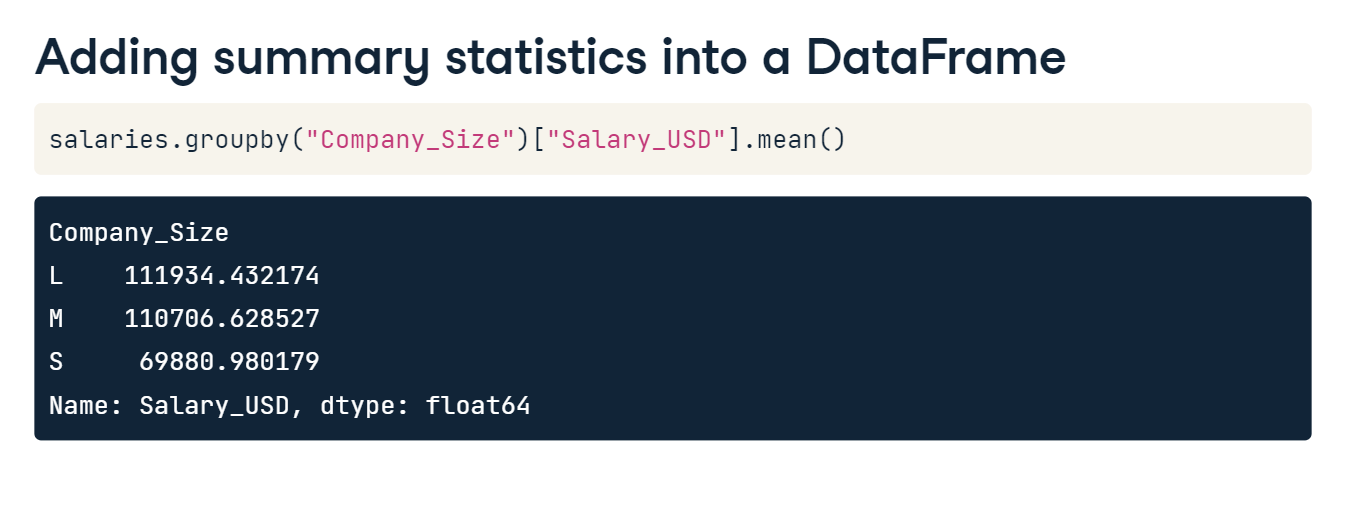
**Previewing the new column**

Printing the first five rows of the original and new column, we can see that values in Salary\_USD are equal to one-point-two percent of the Salary\_In\_Rupees column.



**Adding summary statistics into a DataFrame**

Recall that we've previously used pandas' groupby function to calculate summary statistics. Here, we find the mean salary in US dollars by company size. While this is useful, sometimes we might prefer to add summary statistics directly into our DataFrame, rather than creating a summary table.



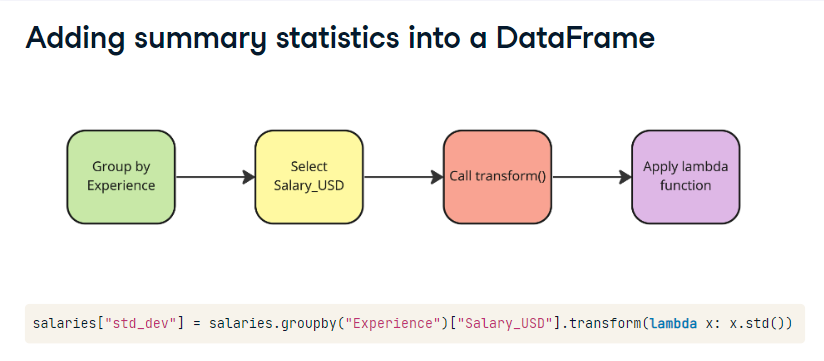
**Adding summary statistics into a DataFrame**

Let's say we would like to create a new column containing the standard deviation of Salary\_USD, where values are conditional based on the Experience column. The first step still involves a groupby, done here with the Experience column.

**Adding summary statistics into a DataFrame**

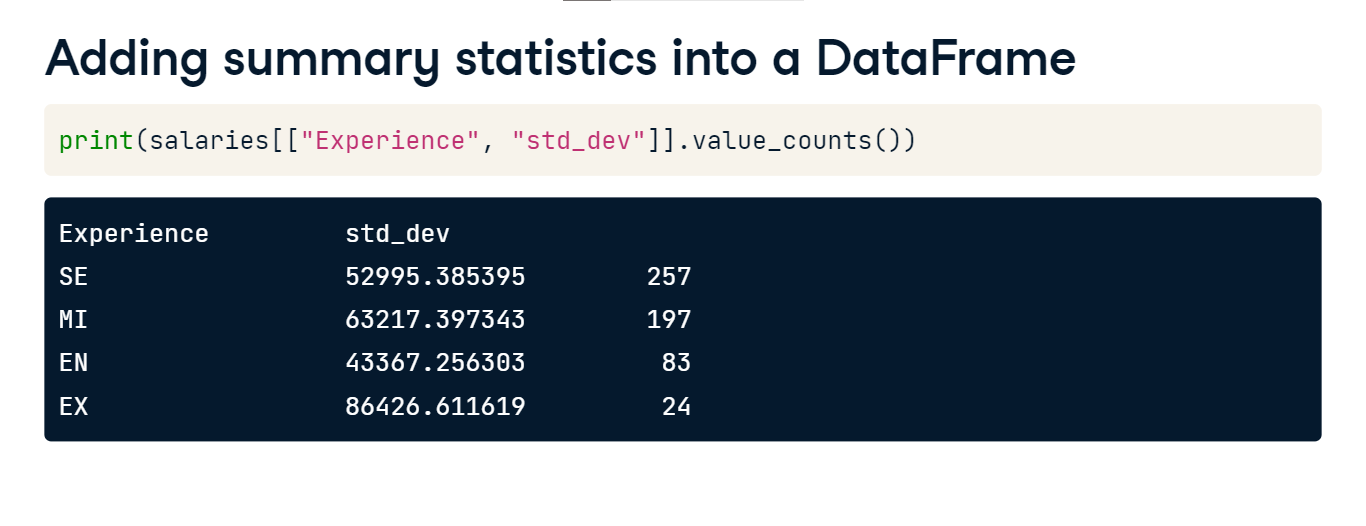
We then select the Salary\_USD column,and call pandas dot-transform.

Inside the transform call, we apply a lambda function using the syntax lambda x semi-colon, followed by a call of x-dot-std. This calculates the standard deviation of salaries based on experience.



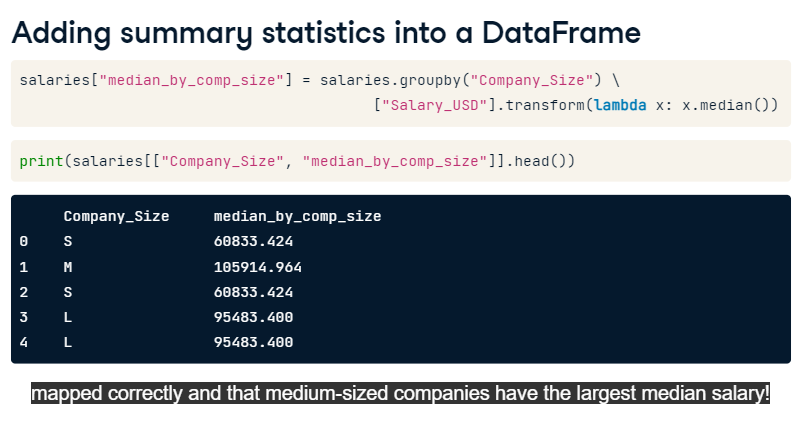
**Adding summary statistics into a DataFrame**

We can select more than one column and use the value\_counts method. This prints the combinations of values for the columns we have chosen, in this case Experience and newly created std\_dev columns. For example, there are 257 rows with SE, or Senior-level, experience, and the standard deviation in salary for this group is nearly 53000 dollars. Unsurprisingly, there appears to be a larger variation in salary associated with the most senior role, Executive.



**Adding summary statistics into a DataFrame**

We can repeat this process for other summary statistics! Here, we add a column for the median salary based on company size. We use a backslash to split our code over two lines, otherwise it is quite long and difficult to read. Previewing the two columns of interest we see the values have been mapped correctly and that medium-sized companies have the largest median salary!



**Handling outliers**

Let's look at how to handle outliers.

**What is an outlier?**

To recap, an outlier is an observation that is far away from other data points. If a house prices dataset has a median of 400,000 dollars, a house that costs five million dollars would likely be considered an outlier. However, we should consider factors that affect price such as location, number of bedrooms, and overall size.

1. 1 Image credit: https://unsplash.com/@ralphkayden

**Using descriptive statistics**

A starting place for identifying outliers is with the pandas dot-describe method. We can see that the maximum salary is more than four times the mean and median. Seems extreme right?

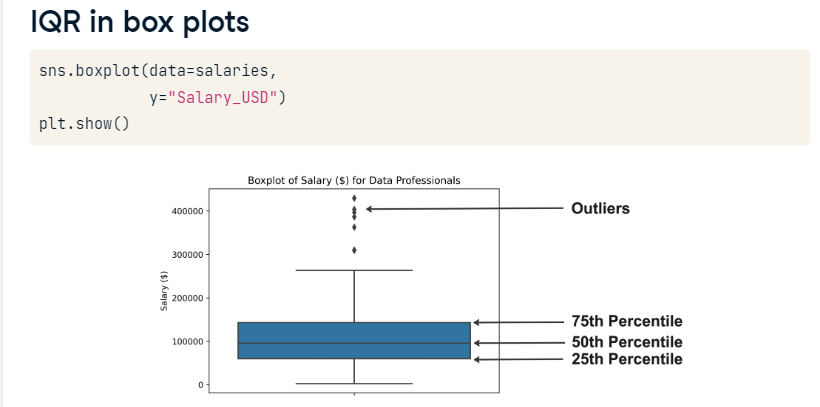


**Using the interquartile range**

We can define an outlier mathematically. First, we need to know the interquartile range, or IQR, which is the difference between the 75th and 25th percentiles.

**IQR in box plots**

Recall that these percentiles are included in box plots, like this one showing salaries of data professionals. The box contains percentiles, and observations considered to be outliers are represented as diamonds outside of the box.



**Using the interquartile range**

Once we have the IQR, we can find an upper outlier by looking for values above the sum of the 75th percentile plus one-point-five times the IQR. Lower outliers have values below the sum of the 25th percentile minus one-point-five times the IQR.



**Identifying thresholds**

We can calculate percentiles using the Series-dot-quantile method. We pass zero-point-seven-five to find the 75th percentile for salary, then pass zero-point-two-five to get the 25th percentile. We calculate the IQR by subtracting one from the other. Printing the result shows an IQR of around 76000 dollars.



**Identifying outliers**

We can plug these variables into our formulae to find the value thresholds, first for the upper limit and then for the lower limit. Printing the results, we can see that the lower limit is actually below zero, which isn't possible given we are working with salaries!



**Subsetting our data**

We can find values outside of these limits by subsetting our data. It will only return upper outliers, but for the purpose of demonstrating the syntax, we've also filtered for values below the lower threshold. We also subset to just show Experience, Employee\_Location, and Salary\_USD. There are nine individuals with a salary above the upper threshold. Notice how none of them are entry level and they are all based in the US?



**Why look for outliers?**

So why is the detection of outliers an important part of exploratory data analysis? These are extreme values and may not accurately represent the data. Additionally, they can skew the mean and standard deviation. If we plan to perform statistical tests or build machine learning models, these will often require data that is normally distributed and not skewed!

**What to do about outliers?**

Once we know we have outliers, we need to decide what to do. It's helpful to ask ourselves why these outliers exist. For example, salaries can be very high depending on level of experience and the country of employment, so could be representative of a subset of our data. If this is the case, we could just leave them alone. Alternatively, do we know the values are accurate? Could there have been an error in data collection? If there's an error, we could remove the values.

**Dropping outliers**

We can remove outliers by modifying the syntax we used to subset our data, filtering for values more than the lower limit and less than the upper limit. Reprinting our descriptive statistics shows nine fewer values, a mean that is 5000 dollars less than before, and a much lower maximum salary!



**Distribution of salaries**

To highlight the impact of removing outliers, let's plot a histogram of the original dataset containing the outliers. We see the distribution is right-skewed by the upper outliers. Plotting with the no\_outliers dataset, salaries is now less skewed and looks more like a normal distribution!

