**Filtering joins**

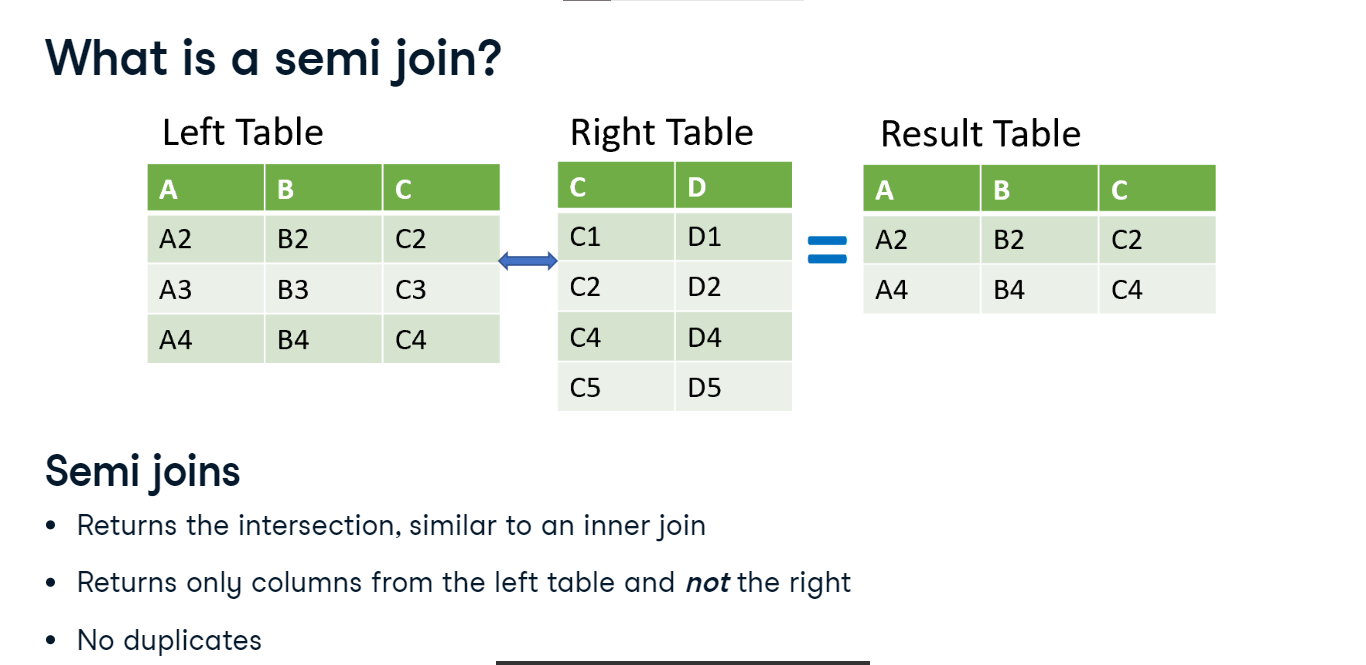
Welcome to the third chapter! In this lesson, we will discuss a type of join called a filtering join. pandas doesn't provide direct support for filtering joins, but we will learn how to replicate them.

**Mutating versus filtering joins**

So far, we have only worked with mutating joins, which combines data from two tables. However, filtering joins filter observations from one table based on whether or not they match an observation in another table.

**What is a semi join?**

Let's start with a semi join. A semi join filters the left table down to those observations that have a match in the right table. It is similar to an inner join where only the intersection between the tables is returned, but unlike an inner join, only the columns from the left table are shown. Finally, no duplicate rows from the left table are returned, even if there is a one-to-many relationship. Let's look at an example.



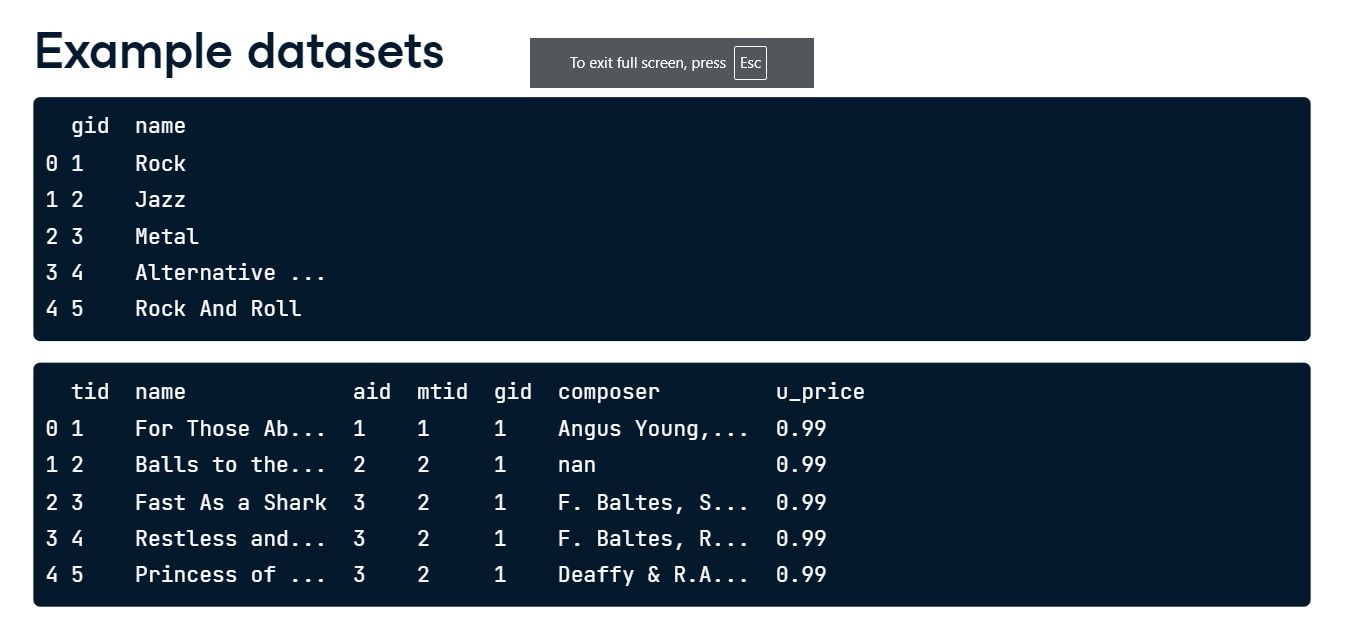
**Musical dataset**

For this chapter, the dataset we will use is from an online music streaming service.

1. 1 Photo by Vlad Bagacian from Pexels

**Example datasets**

In this new dataset, we have a table of song genres shown here. There's also a table of top-rated song tracks. The 'gid' column connects the two tables. Let's say we want to find what genres appear in our table of top songs. A semi join would return only the columns from the genre table and not the tracks.



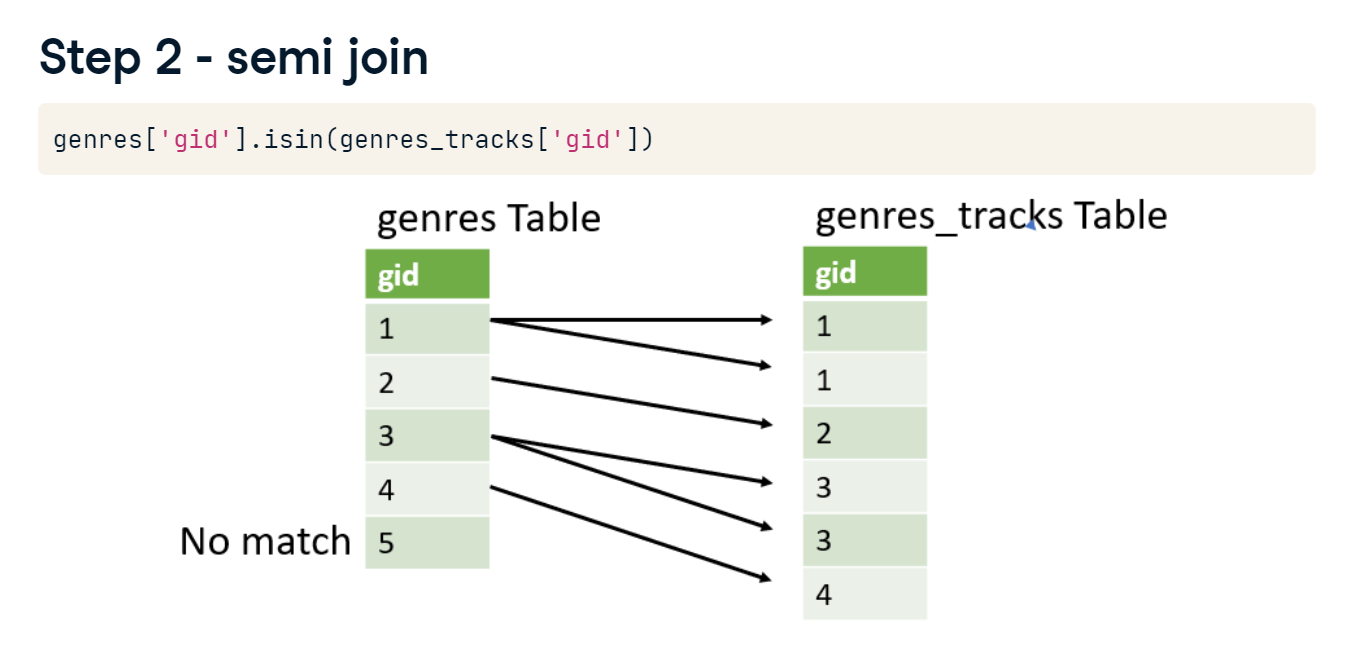
**Step 1 - semi join**

First, let's merge the two tables with an inner join. We also print the first few rows of the genres\_tracks variable. Since this is an inner join, the returned 'gid' column holds only values where both tables matched.



**Step 2 - semi join**

For the next step in the technique, let's focus on this line of code. It uses a method called isin(), which compares every 'gid' in the genres table to the 'gid' in the genres\_tracks table. This will tell us if our genre appears in our merged genres\_tracks table.



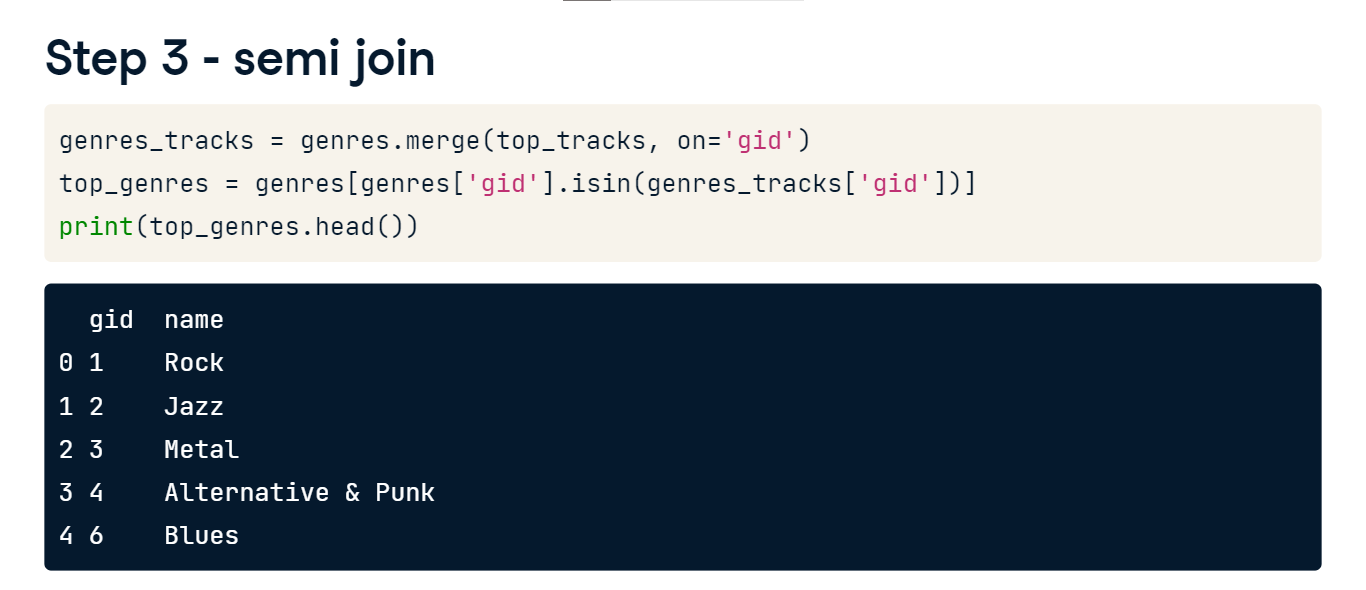
**Step 2 - semi join**

This line of code returns a Boolean Series of true or false values.



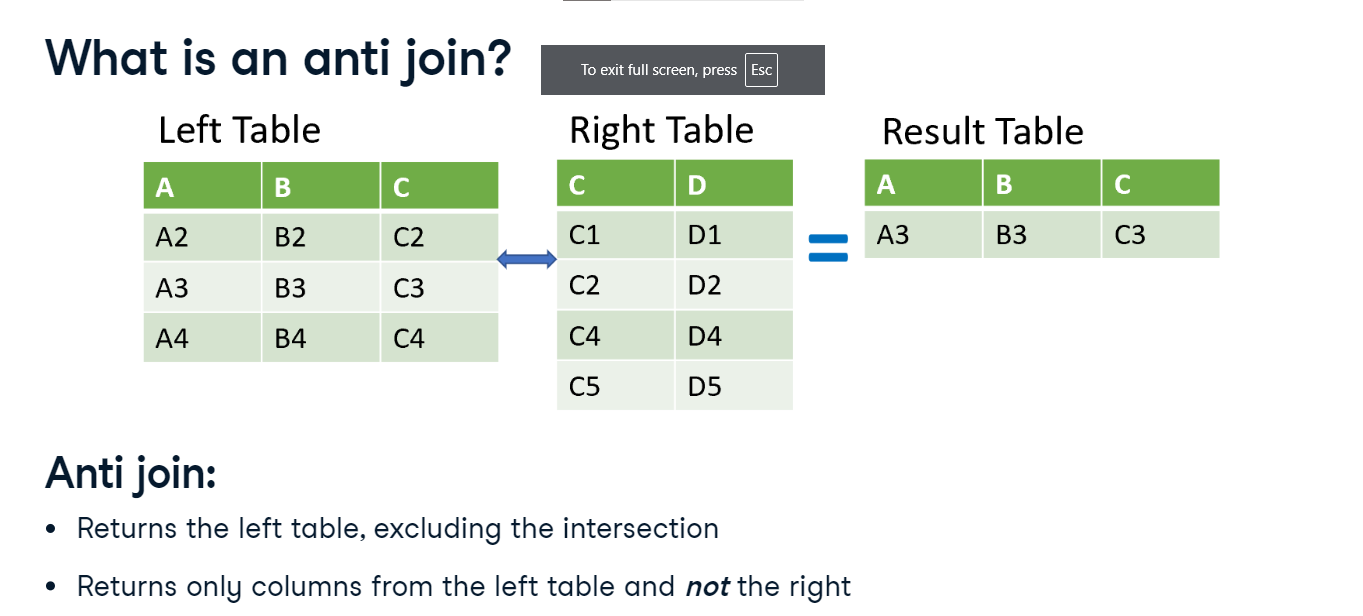
**Step 3 - semi join**

To combine everything, we use that line of code to subset the genres table. The results are saved to top\_genres and we print a few rows. We've completed a semi join. These are rows in the genre table that are also found in the top\_tracks table. This is called a filtering join because we've filtered the genres table by what's in the top\_tracks table.



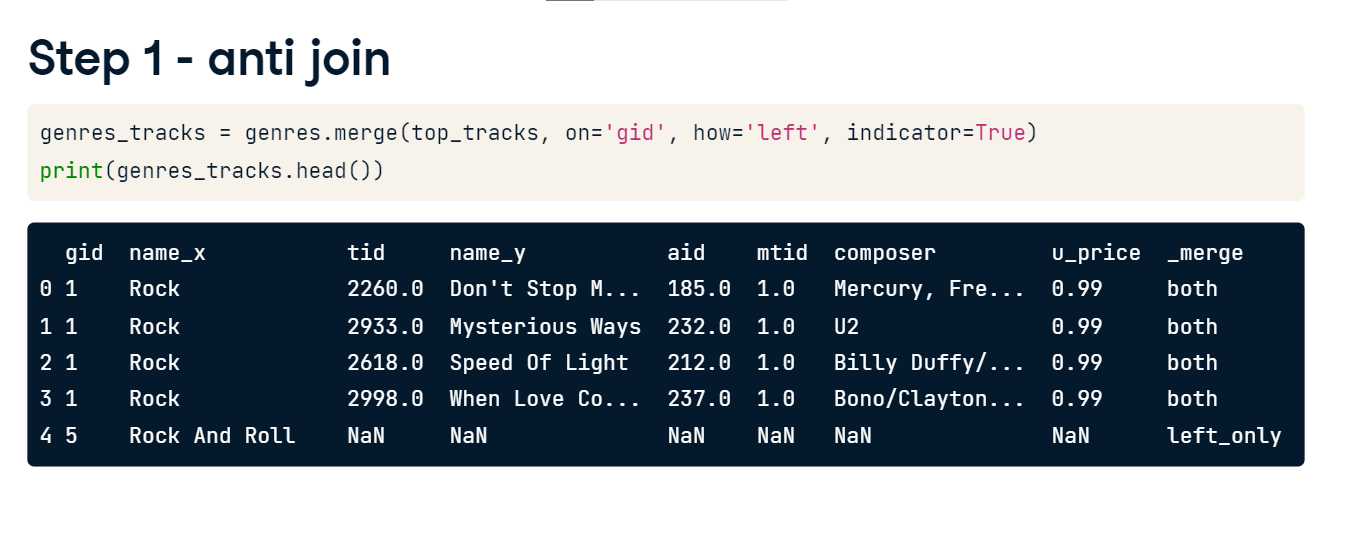
**What is an anti join?**

Now let's talk about anti joins. An anti join returns the observations in the left table that do not have a matching observation in the right table. It also only returns the columns from the left table. Now, let's go back to our example. Instead of finding which genres are in the table of top tracks, let's now find which genres are not with an anti join.



**Step 1 - anti join**

The first step is to use a left join returning all of the rows from the left table. Here we'll use the indicator argument and set it to True. With indicator set to True, the merge method adds a column called "\_merge" to the output. This column tells the source of each row. For example, the first four rows found a match in both tables, whereas the last can only be found in the left table.



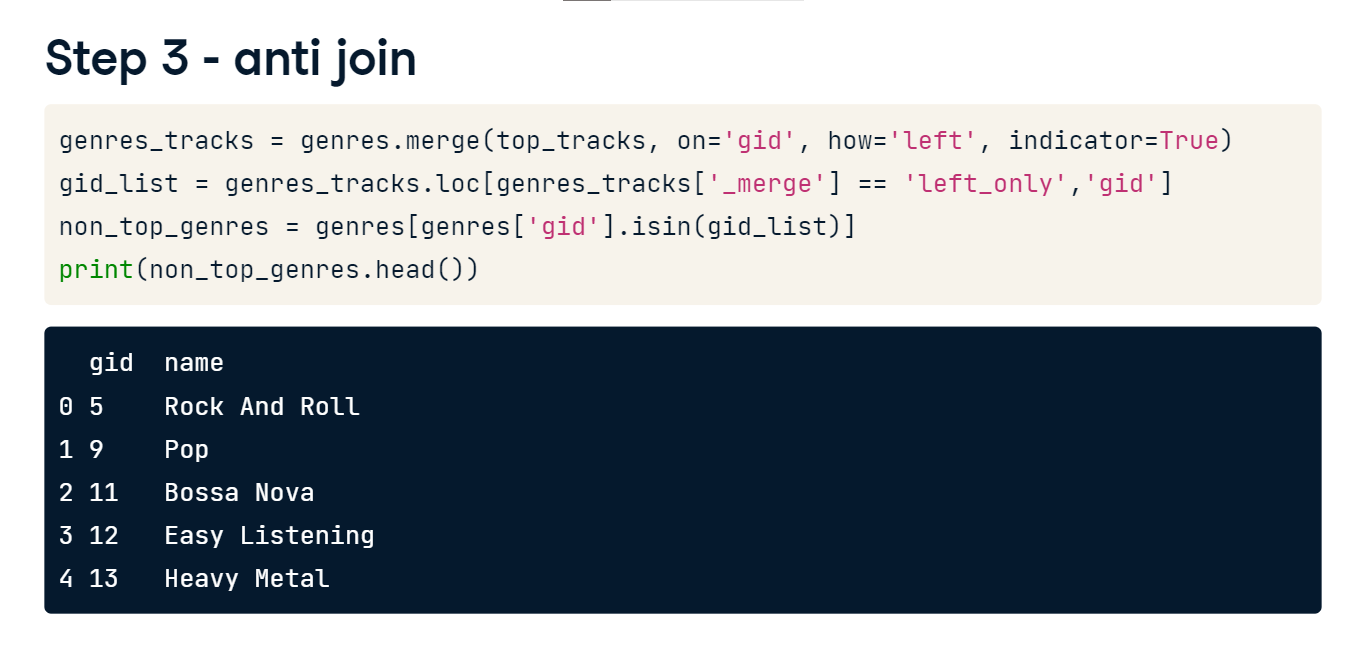
**Step 2 - anti join**

Next, we use the "loc" accessor and "\_merge" column to select the rows that only appeared in the left table and return only the "gid" column from the genres\_tracks table. We now have a list of gids not in the tracks table.



**Step 3 - anti join**

In our final step we use the isin() method to filter for the rows with gids in our gid\_list. Our output shows those genres not in the tracks table.

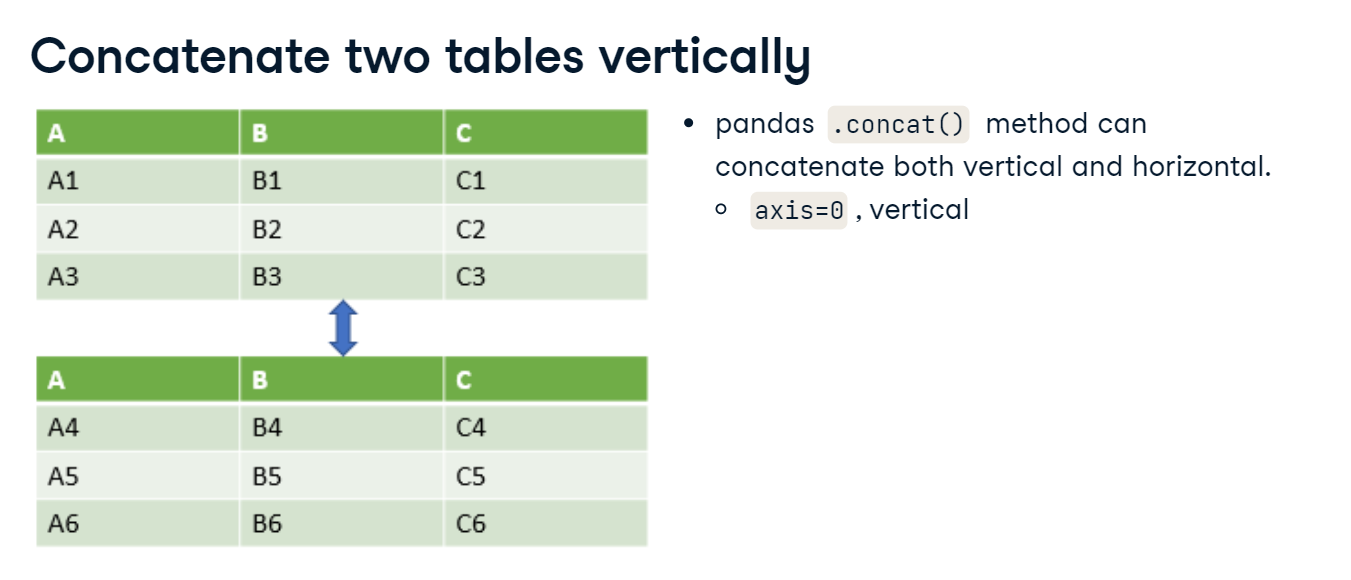


**Concatenate DataFrames together vertically**

Hello there! In this lesson, we'll talk about how to connect two tables vertically.

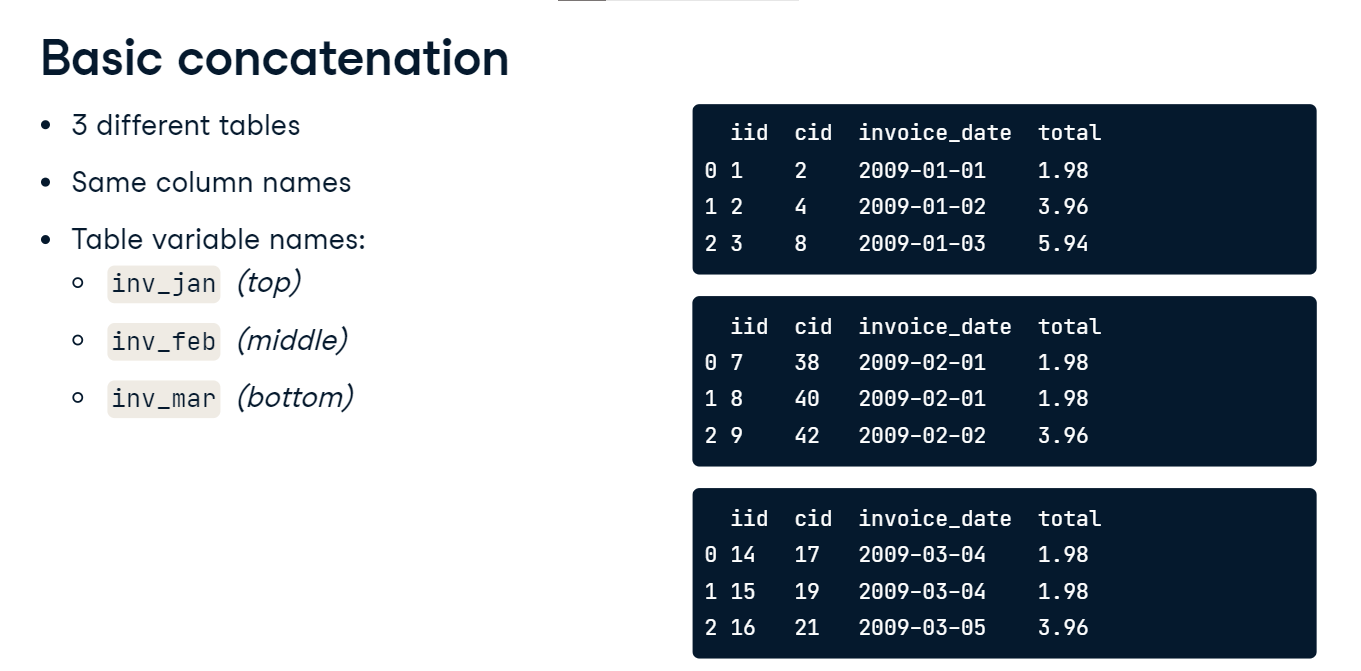
**Concatenate two tables vertically**

So far in this course, we have only discussed how to merge two tables, which mainly grows them horizontally. But what if we wanted to grow them vertically? We can use the concat method to concatenate, or stick tables together, vertically or horizontally, but in this lesson, we'll focus on vertical concatenation.



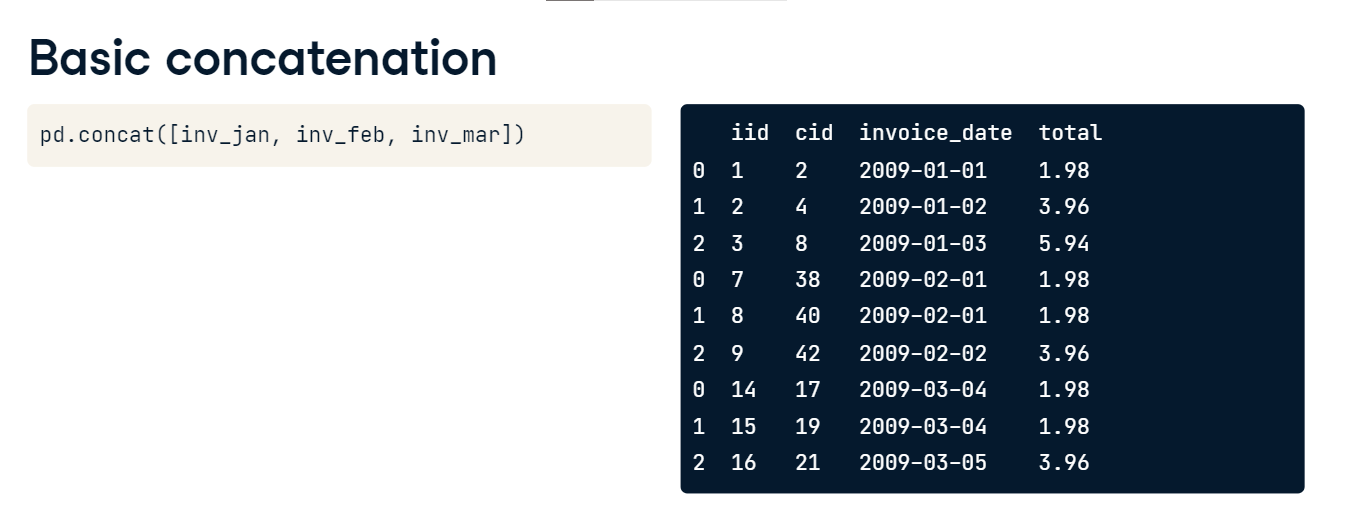
**Basic concatenation**

Often, data for different periods of time will come in multiple tables, but if we want to analyze it together, we'll need to combine them into one. Here are three separate tables of invoice data from our streaming service. Notice the column headers are the same. The separate tables are named "inv" underscore Jan through March.



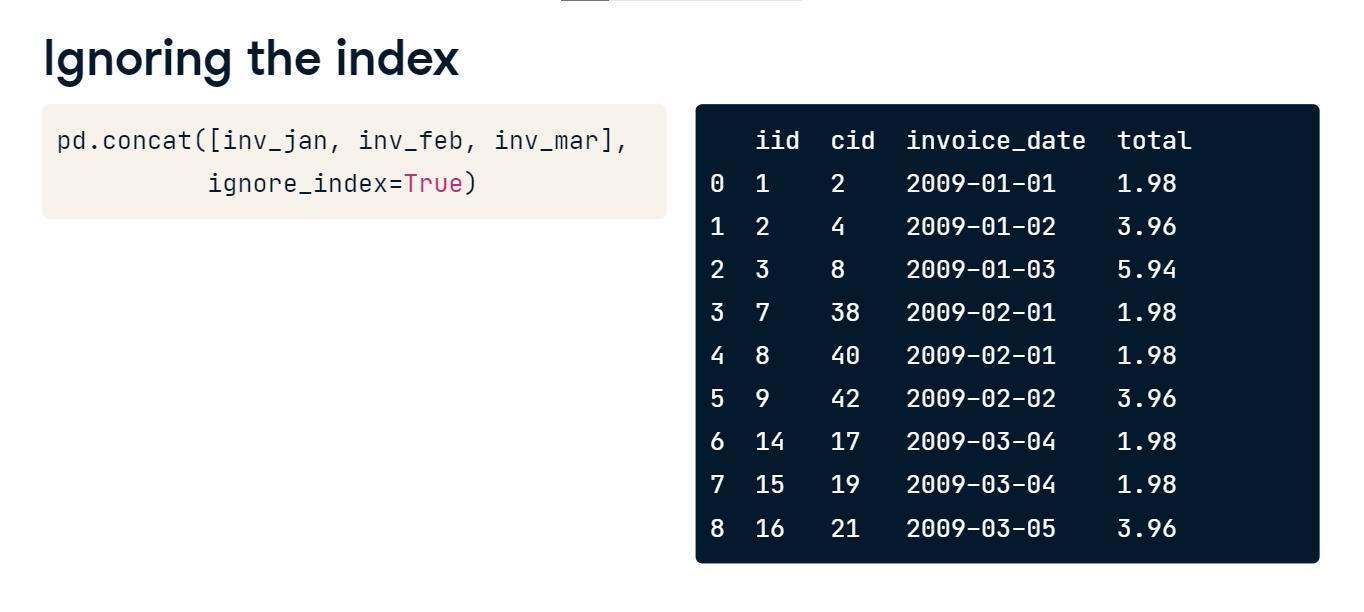
**Basic concatenation**

We can pass a list of table names into pandas dot concat to combine the tables in the order they're passed in. To concatenate vertically, the axis argument should be set to 0, but 0 is the default, so we don't need to explicitly write this. The result is a vertically combined table. Notice each table's index value was retained.



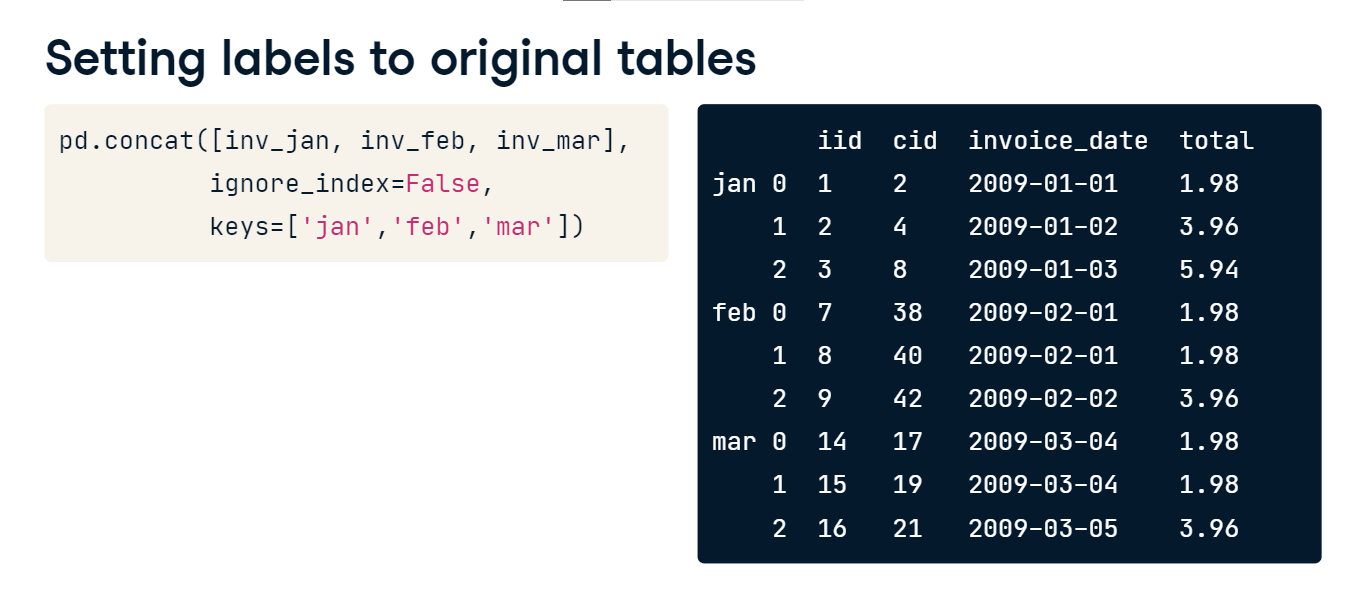
**Ignoring the index**

If the index contains no valuable information, then we can ignore it in the concat method by setting ignore\_index to True. The result is that the index will go from 0 to n-1.



**Setting labels to original tables**

Now, suppose we wanted to associate specific keys with each of the pieces of our three original tables. We can provide a list of labels to the keys argument. Make sure that ignore\_index argument is False, since you can't add a key and ignore the index at the same time. This results in a table with a multi-index, with the label on the first level.

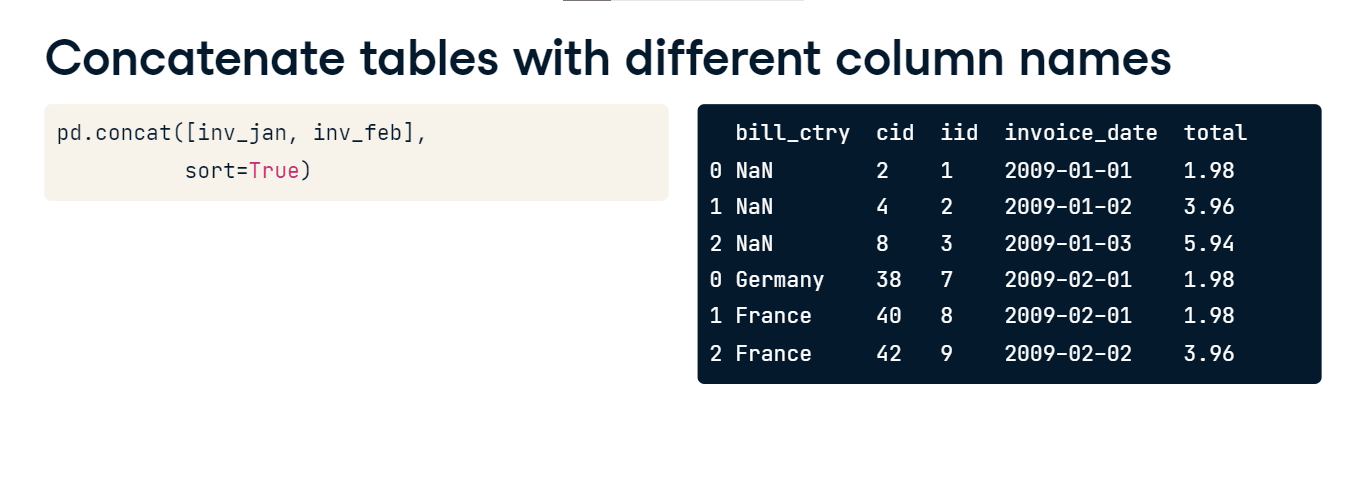


**Concatenate tables with different column names**

What if we need to combine tables that have different column names? The "inv\_feb" table now has a column added for billing country.

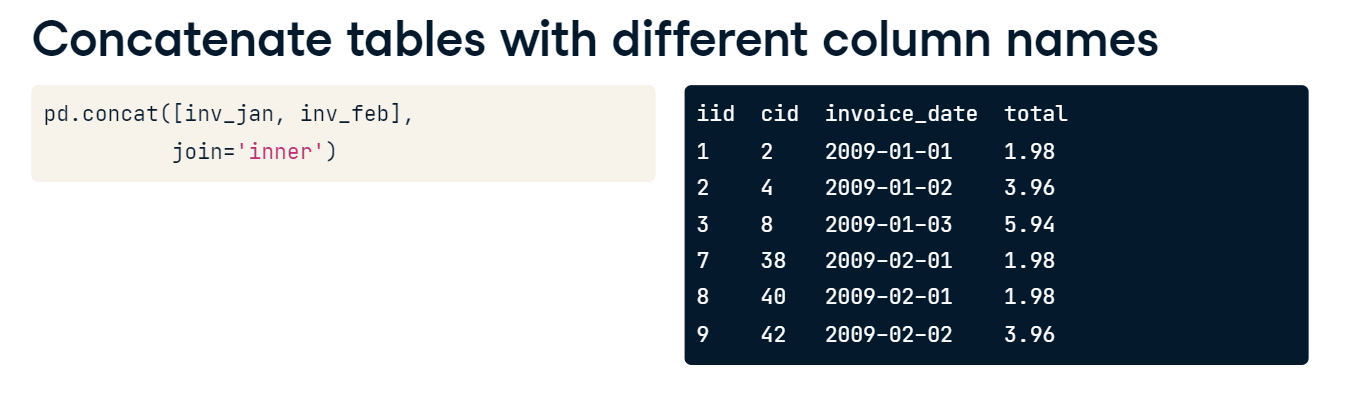
**Concatenate tables with different column names**

The concat method by default will include all of the columns in the different tables it's combining. The sort argument, if true, will alphabetically sort the different column names in the result. We can see in the results that the billing country for January invoices is NaN. However, there are values for the February invoices.



**Concatenate tables with different column names**

If we only want the matching columns between tables, we can set the join argument to "inner". Its default value is equal to "outer", which is why concat by default will include all of the columns. Additionally, the sort argument has no effect when join equals "inner". The order of the columns will be the same as the input tables. Now the bill country column is gone and we're only left with the columns the tables have in common.

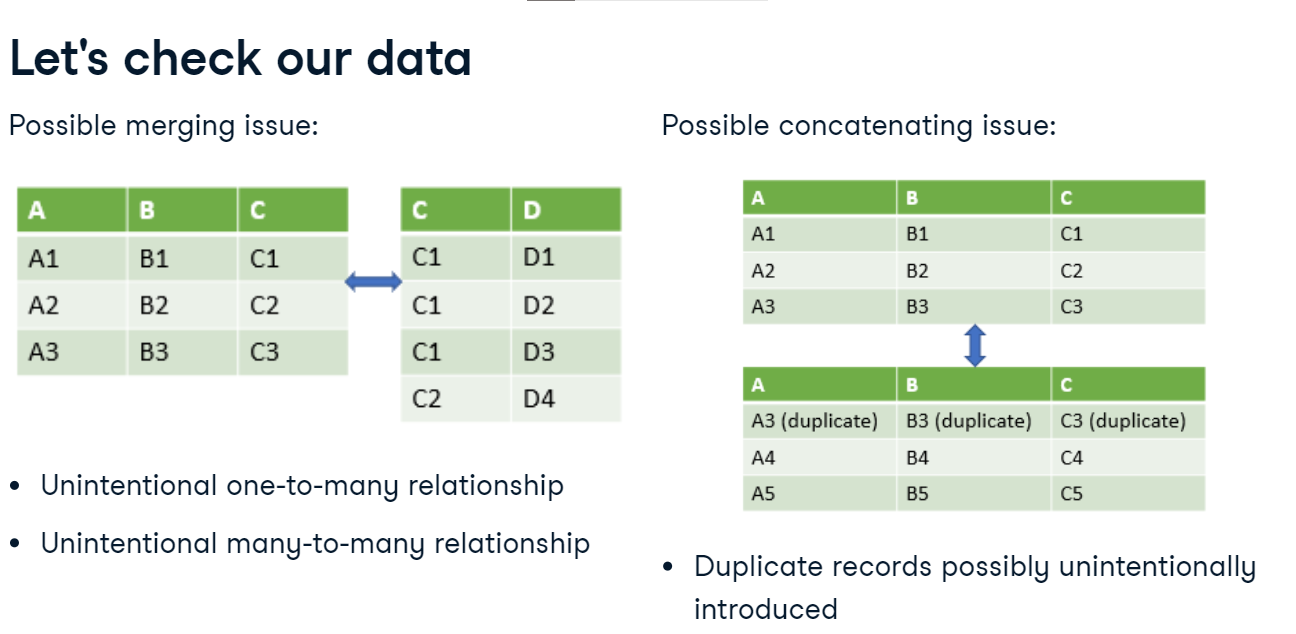


**Verifying integrity**

Welcome back. In this lesson, let's talk about verifying the integrity of our data.

**Let's check our data**

Both the merge and concat methods have special features that allow us to verify the structure of our data. When merging two tables, we might expect the tables to have a one-to-one relationship. However, one of the columns we are merging on may have a duplicated value, which will turn the relationship into a one-to-many. When concatenating tables vertically, we might unintentionally create duplicate records if a record exists in both tables. The validate and verify\_integrity arguments of the merge and concat methods respectively will allow us to verify the data.

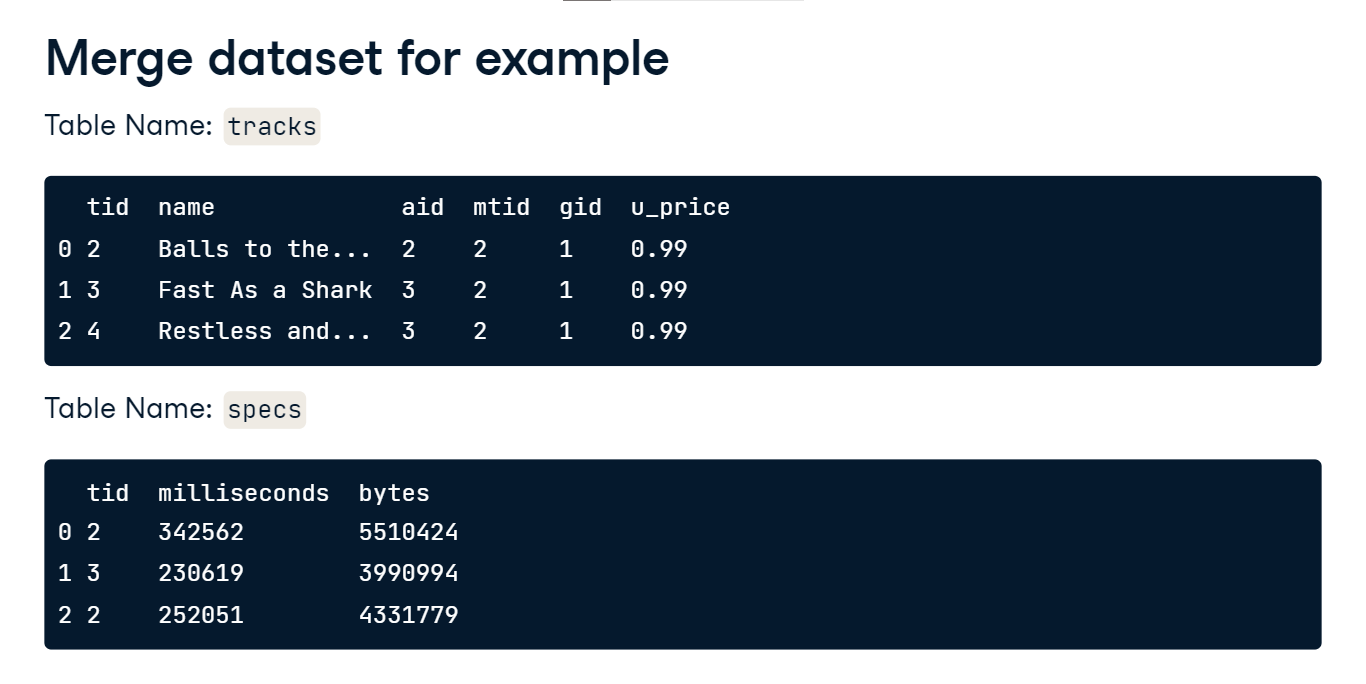


**Validating merges**

Let's start with the merge method. If we provide the validate argument one of these key strings, it will validate the relationship between the two tables. For example, if we specify we want a one-to-one relationship, but it turns out the relationship is not one-to-one, then an error is raised. Let's try it out.

**Merge dataset for example**

In this example, we want to merge these two tables on the column "tid". Again, our data is from our music service. The first table is named "tracks", and the second is called "specs" for the technical specifications of each track. Each track should have one set of specifications, so this should be a one-to-one merge. However, notice that the specs table has two rows with a "tid" value equal to two. Therefore, merging these tables now becomes, unintentionally, a one-to-many relationship.



**Merge validate: one\_to\_one**

Let's merge the two tables with the tracks table on the left and specs on the right. Additionally, let's set the validate argument equal to one\_to\_one. In the result, a MergeError is raised. Python then tells us that the right table has duplicates, so it is not a one-to-one merge. We know that we should handle those duplicates properly before merging.



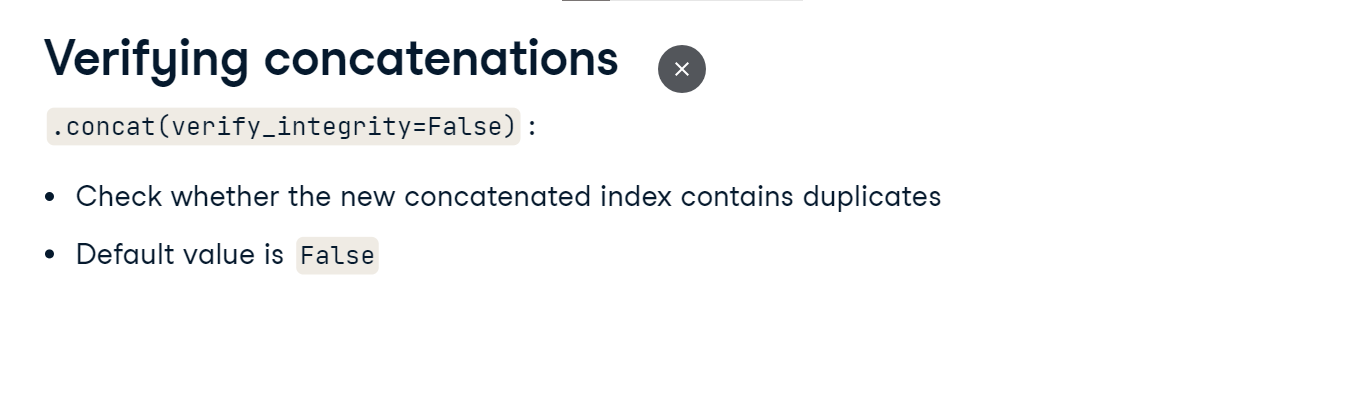
**Merge validate: one\_to\_many**

Now we'll merge album information with the tracks table. For every album there are multiple tracks, so this should be a one-to-many relationship. When we set the validate argument to "one\_to\_many" no error is raised.



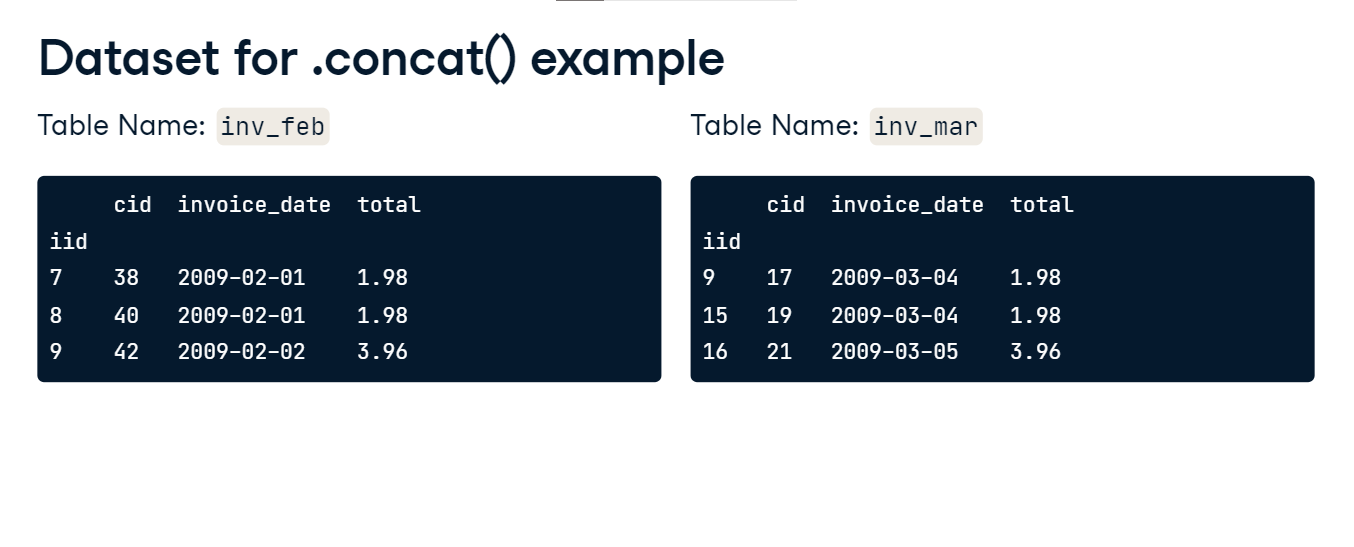
**Verifying concatenations**

Let's now talk about the concat method. It has the argument verify\_integrity, which by default is False. However, if set to True, it will check if there are duplicate values in the index and raise an error if there are. It will only check the index values and not the columns.



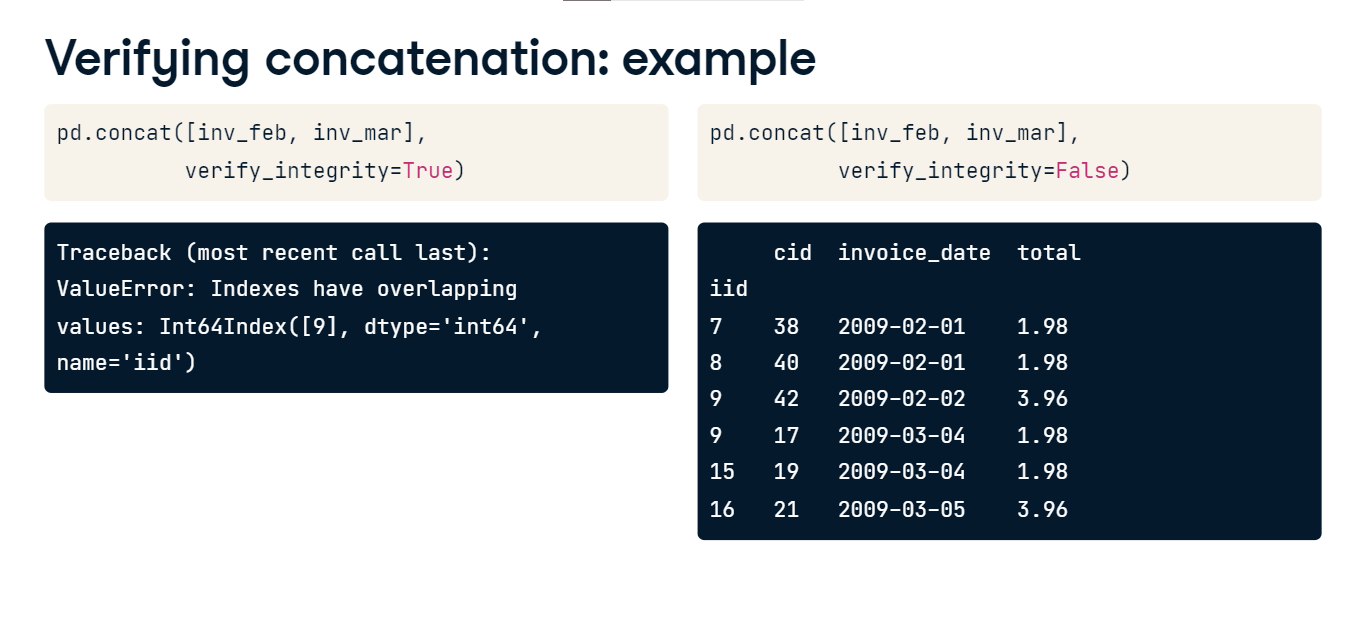
**Dataset for .concat() example**

To try out this feature, we will attempt to concatenate these two tables. They are the February and March invoice data shown in a previous video. However, both tables were modified so the index contains invoice IDs. Notice that invoice ID number 9 is in both tables.



**Verifying concatenation: example**

Let's try to concatenate the two tables together with the verify\_integrity argument set to True. The concat method raises a ValueError stating that the indexes have overlapping values. Now let's try to concatenate the two tables again with the verify\_integrity set back to the default value of False. The concat method now returns a combined table with the invoice ID of number 9 repeated twice.



**Why verify integrity and what to do**

Often our data is not clean, and it may not always be evident if data has the expected structure. Therefore, verifying this structure is useful, saving us from having a mean skewed by duplicate values, or from creating inaccurate plots. If you receive a MergeError or a ValueError, you can fix the incorrect data or drop duplicate rows. In general, you should look to correct the issue.