# Filtering joins

JOINING DATA WITH PANDAS

1. 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.**

**Aaren Stubberfield**
Instructor

datacamp

# Mutating versus filtering joins

**Mutating joins:**

**Pandas doesn't provide direct support for filtering joins, but we will learn how to replicate them.**

- Combines data from two tables based on matching observations in both tables

**Filtering joins:**

- Filter observations from table based on whether or not they match an observation in another table

2. 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?

Left Table

| A | B | C |
|---|---|---|
| A2 | B2 | C2 |
| A3 | B3 | C3 |
| A4 | B4 | C4 |

Right Table

| C | D |
|---|---|
| C1 | D1 |
| C2 | D2 |
| C4 | D4 |
| C5 | D5 |

Result Table

| A | B | C |
|---|---|---|
| A2 | B2 | C2 |
| A4 | B4 | C4 |

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.

## Semi-joins

- Returns the intersection, similar to an inner join
- Returns only columns from the left table and ***not*** the right
- No duplicates

# Musical dataset


SQLITE TUTORIAL

[1] Photo by Vlad Bagacian from Pexels

# Example datasets

| | gid | name |
|---|---|---|
| 0 | 1 | Rock |
| 1 | 2 | Jazz |
| 2 | 3 | Metal |
| 3 | 4 | Alternative ... |
| 4 | 5 | Rock And Roll |

5. 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.

| | tid | name | aid | mtid | gid | composer | u_price |
|---|---|---|---|---|---|---|---|
| 0 | 1 | For Those Ab... | 1 | 1 | 1 | Angus Young,... | 0.99 |
| 1 | 2 | Balls to the... | 2 | 2 | 1 | nan | 0.99 |
| 2 | 3 | Fast As a Shark | 3 | 2 | 1 | F. Baltes, S... | 0.99 |
| 3 | 4 | Restless and... | 3 | 2 | 1 | F. Baltes, R... | 0.99 |
| 4 | 5 | Princess of ... | 3 | 2 | 1 | Deaffy & R.A... | 0.99 |

# Step 1 - semi-join

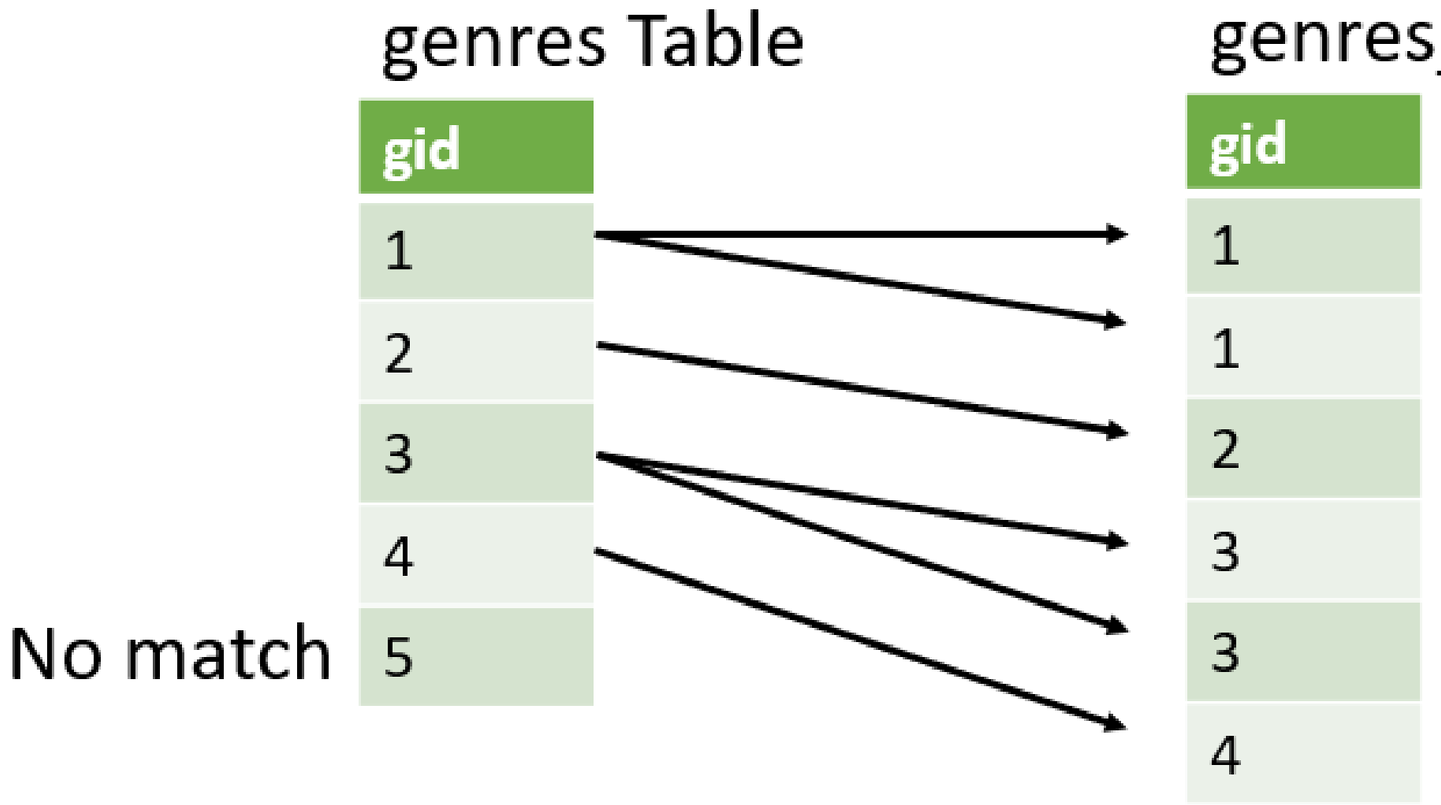
```
genres_tracks = genres.merge(top_tracks, on='gid')
print(genres_tracks.head())
```

```
  gid  name_x  tid   name_y           aid  mtid  composer          u_price
0 1    Rock    2260  Don't Stop M...  185  1     Mercury, Fre...   0.99
1 1    Rock    2933  Mysterious Ways  232  1     U2                0.99
2 1    Rock    2618  Speed Of Light   212  1     Billy Duffy/...   0.99
3 1    Rock    2998  When Love Co...  237  1     Bono/Clayton...   0.99
4 1    Rock    685   Who'll Stop ...  54   1     J. C. Fogerty     0.99
```

6. 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

```
genres['gid'].isin(genres_tracks['gid'])
```




7. 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

```
genres['gid'].isin(genres_tracks['gid'])
```

```
0     True
1     True
2     True
3     True
4    False
Name: gid, dtype: bool
```

8. Step 2 - semi-join
This line of code returns a Boolean Series of true or false values.

# Step 3 - semi-join

```
genres_tracks = genres.merge(top_tracks, on='gid')
top_genres = genres[genres['gid'].isin(genres_tracks['gid'])]
print(top_genres.head())
```

```
   gid  name
0  1    Rock
1  2    Jazz
2  3    Metal
3  4    Alternative & Punk
4  6    Blues
```

9. 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?

Left Table

| A | B | C |
|---|---|---|
| A2 | B2 | C2 |
| A3 | B3 | C3 |
| A4 | B4 | C4 |

Right Table

| C | D |
|---|---|
| C1 | D1 |
| C2 | D2 |
| C4 | D4 |
| C5 | D5 |

Result Table

| A | B | C |
|---|---|---|
| A3 | B3 | C3 |

10. 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.

## Anti-join:

- Returns the left table, excluding the intersection
- Returns only columns from the left table and ***not*** the right

# Step 1 - anti-join

```
genres_tracks = genres.merge(top_tracks, on='gid', how='left', indicator=True)
print(genres_tracks.head())
```

```
   gid  name_x           tid     name_y           aid    mtid  composer         u_price  _merge
0  1    Rock             2260.0  Don't Stop M...  185.0  1.0   Mercury, Fre...  0.99     both
1  1    Rock             2933.0  Mysterious Ways  232.0  1.0   U2               0.99     both
2  1    Rock             2618.0  Speed Of Light   212.0  1.0   Billy Duffy/...  0.99     both
3  1    Rock             2998.0  When Love Co...  237.0  1.0   Bono/Clayton...  0.99     both
4  5    Rock And Roll    NaN     NaN              NaN    NaN   NaN              NaN      left_only
```

11. 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

```
gid_list = genres_tracks.loc[genres_tracks['_merge'] == 'left_only', 'gid']
print(gid_list.head())
```

```
23      5
34      9
36     11
37     12
38     13
Name: gid, dtype: int64
```

12. 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

```
genres_tracks = genres.merge(top_tracks, on='gid', how='left', indicator=True)
gid_list = genres_tracks.loc[genres_tracks['_merge'] == 'left_only','gid']
non_top_genres = genres[genres['gid'].isin(gid_list)]
print(non_top_genres.head())
```

```
   gid  name
0  5    Rock And Roll
1  9    Pop
2  11   Bossa Nova
3  12   Easy Listening
4  13   Heavy Metal
```

13. Step 3 - anti-join
n 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.

# Let's practice!

JOINING DATA WITH PANDAS

# Concatenate DataFrames together vertically

JOINING DATA WITH PANDAS

**Aaren Stubberfield**
Instructor

2. 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.**

datacamp

# Concatenate two tables vertically

| A | B | C |
|---|---|---|
| A1 | B1 | C1 |
| A2 | B2 | C2 |
| A3 | B3 | C3 |

| A | B | C |
|---|---|---|
| A4 | B4 | C4 |
| A5 | B5 | C5 |
| A6 | B6 | C6 |

- Pandas `.concat()` method can concatenate both vertical and horizontal.
  - `axis=0`, vertical

3. 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

- 3 different tables
- Same column names
- Table variable names:
  - `inv_jan` *(top)*
  - `inv_feb` *(middle)*
  - `inv_mar` *(bottom)*

4. 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.

```
   iid  cid  invoice_date  total
0  1    2    2009-01-01    1.98
1  2    4    2009-01-02    3.96
2  3    8    2009-01-03    5.94
```

```
   iid  cid  invoice_date  total
0  7    38   2009-02-01    1.98
1  8    40   2009-02-01    1.98
2  9    42   2009-02-02    3.96
```

```
   iid  cid  invoice_date  total
0  14   17   2009-03-04    1.98
1  15   19   2009-03-04    1.98
2  16   21   2009-03-05    3.96
```

# Basic concatenation

```
pd.concat([inv_jan, inv_feb, inv_mar])
```

```
   iid  cid  invoice_date  total
0  1    2    2009-01-01    1.98
1  2    4    2009-01-02    3.96
2  3    8    2009-01-03    5.94
0  7    38   2009-02-01    1.98
1  8    40   2009-02-01    1.98
2  9    42   2009-02-02    3.96
0  14   17   2009-03-04    1.98
1  15   19   2009-03-04    1.98
2  16   21   2009-03-05    3.96
```

# Ignoring the index

```
pd.concat([inv_jan, inv_feb, inv_mar],
          ignore_index=True)
```

5. 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.

| | iid | cid | invoice_date | total |
|---|---|---|---|---|
| 0 | 1 | 2 | 2009-01-01 | 1.98 |
| 1 | 2 | 4 | 2009-01-02 | 3.96 |
| 2 | 3 | 8 | 2009-01-03 | 5.94 |
| 3 | 7 | 38 | 2009-02-01 | 1.98 |
| 4 | 8 | 40 | 2009-02-01 | 1.98 |
| 5 | 9 | 42 | 2009-02-02 | 3.96 |
| 6 | 14 | 17 | 2009-03-04 | 1.98 |
| 7 | 15 | 19 | 2009-03-04 | 1.98 |
| 8 | 16 | 21 | 2009-03-05 | 3.96 |

# Setting labels to original tables

```
pd.concat([inv_jan, inv_feb, inv_mar],
          ignore_index=False,
          keys=['jan','feb','mar'])
```

6. 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.

| | | iid | cid | invoice_date | total |
|---|---|---|---|---|---|
| jan | 0 | 1 | 2 | 2009-01-01 | 1.98 |
| | 1 | 2 | 4 | 2009-01-02 | 3.96 |
| | 2 | 3 | 8 | 2009-01-03 | 5.94 |
| feb | 0 | 7 | 38 | 2009-02-01 | 1.98 |
| | 1 | 8 | 40 | 2009-02-01 | 1.98 |
| | 2 | 9 | 42 | 2009-02-02 | 3.96 |
| mar | 0 | 14 | 17 | 2009-03-04 | 1.98 |
| | 1 | 15 | 19 | 2009-03-04 | 1.98 |
| | 2 | 16 | 21 | 2009-03-05 | 3.96 |

# Concatenate tables with different column names

Table: `inv_jan`

| | iid | cid | invoice_date | total |
|---|---|---|---|---|
| 0 | 1 | 2 | 2009-01-01 | 1.98 |
| 1 | 2 | 4 | 2009-01-02 | 3.96 |
| 2 | 3 | 8 | 2009-01-03 | 5.94 |

Table: `inv_feb`

| | iid | cid | invoice_date | total | bill_ctry |
|---|---|---|---|---|---|
| 0 | 7 | 38 | 2009-02-01 | 1.98 | Germany |
| 1 | 8 | 40 | 2009-02-01 | 1.98 | France |
| 2 | 9 | 42 | 2009-02-02 | 3.96 | France |

7. 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

```
pd.concat([inv_jan, inv_feb],
          sort=True)
```

8. Concatenate tables with different column names
he 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 result that the billing country for January invoices is NaN. However, there are values for the February invoices.

| | bill_ctry | cid | iid | invoice_date | total |
|---|---|---|---|---|---|
| 0 | NaN | 2 | 1 | 2009-01-01 | 1.98 |
| 1 | NaN | 4 | 2 | 2009-01-02 | 3.96 |
| 2 | NaN | 8 | 3 | 2009-01-03 | 5.94 |
| 0 | Germany | 38 | 7 | 2009-02-01 | 1.98 |
| 1 | France | 40 | 8 | 2009-02-01 | 1.98 |
| 2 | France | 42 | 9 | 2009-02-02 | 3.96 |

# Concatenate tables with different column names

```
pd.concat([inv_jan, inv_feb],
          join='inner')
```

9. Concatenate tables with different column names
If we only want the matching columns between tables, we 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 left with only the columns the tables have in common.

| iid | cid | invoice_date | total |
|---|---|---|---|
| 1 | 2 | 2009-01-01 | 1.98 |
| 2 | 4 | 2009-01-02 | 3.96 |
| 3 | 8 | 2009-01-03 | 5.94 |
| 7 | 38 | 2009-02-01 | 1.98 |
| 8 | 40 | 2009-02-01 | 1.98 |
| 9 | 42 | 2009-02-02 | 3.96 |

# Using append method

`.append()`

- Simplified version of the `.concat()` method
- Supports: `ignore_index` , and `sort`
- Does Not Support: `keys` and `join`
  - Always `join = outer`

10. Using append method
Now let's briefly talk about append. Append is a simplified concat method. It supports the ignore_index and sort arguments. However, it does not support keys or join. Join is always set to outer.

# Append these tables

```
   iid  cid  invoice_date  total
0  1    2    2009-01-01    1.98
1  2    4    2009-01-02    3.96
2  3    8    2009-01-03    5.94
```

```
   iid  cid  invoice_date  total  bill_ctry
0  7    38   2009-02-01    1.98   Germany
1  8    40   2009-02-01    1.98   France
2  9    42   2009-02-02    3.96   France
```

```
   iid  cid  invoice_date  total
0  14   17   2009-03-04    1.98
1  15   19   2009-03-04    1.98
2  16   21   2009-03-05    3.96
```

12. Append the tables
Append is a DataFrame method therefore, we list the "inv_jan" table first then call the method. We add the other tables as a list, and set the ignore_index and sort arguments similar to the concat method. In our output, we see null values for the billing country, except for February. Additionally, the index is adjusted as expected.

# Append the tables

```
inv_jan.append([inv_feb, inv_mar],
               ignore_index=True,
               sort=True)
```

12. Append the tables
Append is a DataFrame method therefore, we list the "inv_jan" table first then call the method. We add the other tables as a list, and set the ignore_index and sort arguments similar to the concat method. In our output, we see null values for the billing country, except for February. Additionally, the index is adjusted as expected.

Note: The .concat() method is excellent when you need a lot of control over how concatenation is performed. However, if you do not need as much control, then the .append() method is another option.

Even though .append() is less flexible, it's also simpler than .concat()

| | bill_ctry | cid | iid | invoice_date | total |
|---|---|---|---|---|---|
| 0 | NaN | 2 | 1 | 2009-01-01 | 1.98 |
| 1 | NaN | 4 | 2 | 2009-01-02 | 3.96 |
| 2 | NaN | 8 | 3 | 2009-01-03 | 5.94 |
| 3 | Germany | 38 | 7 | 2009-02-01 | 1.98 |
| 4 | France | 40 | 8 | 2009-02-01 | 1.98 |
| 5 | France | 42 | 9 | 2009-02-02 | 3.96 |
| 6 | NaN | 17 | 14 | 2009-03-04 | 1.98 |
| 7 | NaN | 19 | 15 | 2009-03-04 | 1.98 |
| 8 | NaN | 21 | 16 | 2009-03-05 | 3.96 |

# Let's practice!

JOINING DATA WITH PANDAS

# Verifying integrity

## JOINING DATA WITH PANDAS

**Aaren Stubberfield**
Instructor

datacamp

# Let's check our data

Possible merging issue:

| A | B | C |
|---|---|---|
| A1 | B1 | C1 |
| A2 | B2 | C2 |
| A3 | B3 | C3 |

| C | D |
|---|---|
| C1 | D1 |
| C1 | D2 |
| C1 | D3 |
| C2 | D4 |

- Unintentional one-to-many relationship
- Unintentional many-to-many relationship

Possible concatenating issue:

| A | B | C |
|---|---|---|
| A1 | B1 | C1 |
| A2 | B2 | C2 |
| A3 | B3 | C3 |

| A | B | C |
|---|---|---|
| A3 (duplicate) | B3 (duplicate) | C3 (duplicate) |
| A4 | B4 | C4 |
| A5 | B5 | C5 |

- Duplicate records possibly unintentionally introduced

# Validating merges

`.merge(validate=None)`:

- Checks if merge is of specified type
- `'one_to_one'`
- `'one_to_many'`
- `'many_to_one'`
- `'many_to_many'`

3. 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

Table Name: `tracks`

```
   tid  name              aid  mtid  gid  u_price
0  2    Balls to the...   2    2     1    0.99
1  3    Fast As a Shark   3    2     1    0.99
2  4    Restless and...   3    2     1    0.99
```

Table Name: `specs`

```
   tid  milliseconds  bytes
0  2    342562        5510424
1  3    230619        3990994
2  2    252051        4331779
```

4. 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

```
tracks.merge(specs, on='tid',
             validate='one_to_one')
```

```
Traceback (most recent call last):
MergeError: Merge keys are not unique in right dataset; not a one-to-one merge
```

5. 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

```
albums.merge(tracks, on='aid',
             validate='one_to_many')
```

| | aid | title | artid | tid | name | mtid | gid | u_price |
|---|---|---|---|---|---|---|---|---|
| 0 | 2 | Balls to the... | 2 | 2 | Balls to the... | 2 | 1 | 0.99 |
| 1 | 3 | Restless and... | 2 | 3 | Fast As a Shark | 2 | 1 | 0.99 |
| 2 | 3 | Restless and... | 2 | 4 | Restless and... | 2 | 1 | 0.99 |

6. 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

`.concat(verify_integrity=False)`:

- Check whether the new concatenated index contains duplicates
- Default value is `False`

7. 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

Table Name: `inv_feb`

| iid | cid | invoice_date | total |
|---|---|---|---|
| 7 | 38 | 2009-02-01 | 1.98 |
| 8 | 40 | 2009-02-01 | 1.98 |
| 9 | 42 | 2009-02-02 | 3.96 |

Table Name: `inv_mar`

| iid | cid | invoice_date | total |
|---|---|---|---|
| 9 | 17 | 2009-03-04 | 1.98 |
| 15 | 19 | 2009-03-04 | 1.98 |
| 16 | 21 | 2009-03-05 | 3.96 |

8. 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

```
pd.concat([inv_feb, inv_mar],
          verify_integrity=True)
```

```
Traceback (most recent call last):
ValueError: Indexes have overlapping
values: Int64Index([9], dtype='int64',
name='iid')
```

```
pd.concat([inv_feb, inv_mar],
          verify_integrity=False)
```

```
      cid  invoice_date  total
iid
7     38   2009-02-01    1.98
8     40   2009-02-01    1.98
9     42   2009-02-02    3.96
9     17   2009-03-04    1.98
15    19   2009-03-04    1.98
16    21   2009-03-05    3.96
```

9. 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

Why:

- Real world data is often ***NOT*** clean

What to do:

- Fix incorrect data
- Drop duplicate rows

10. 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.

# Let's practice!

JOINING DATA WITH PANDAS