**Comparing strings**

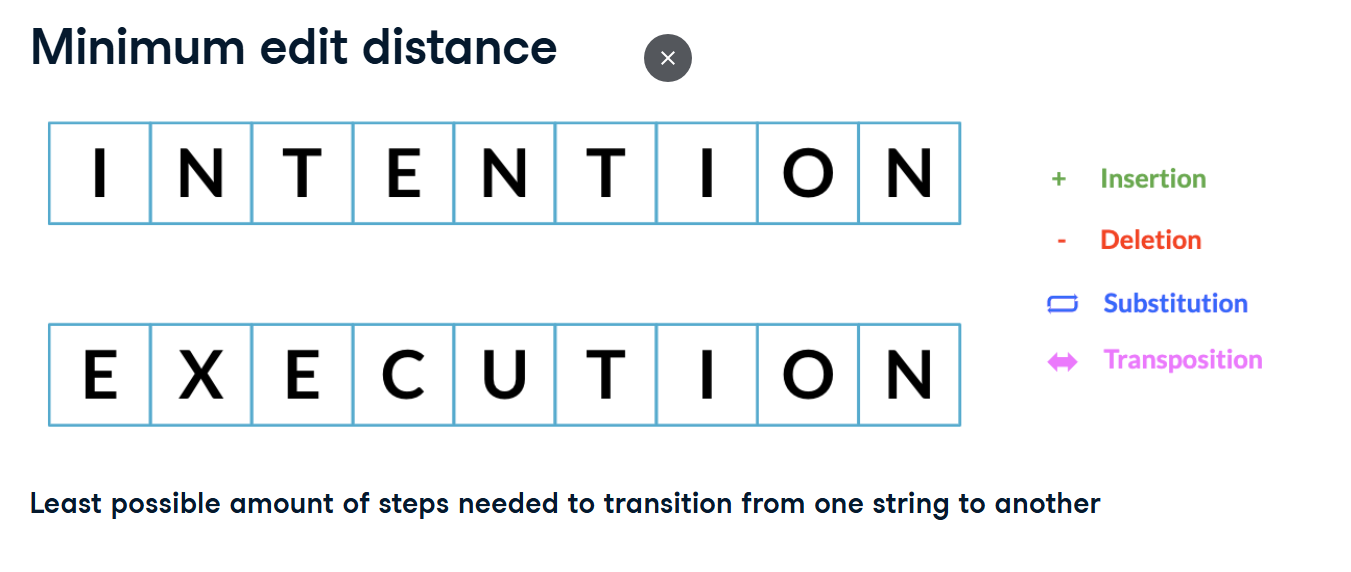
Awesome work on chapter 3! Welcome to the final chapter of this course,

**In this chapter**

where we'll discover the world of record linkage. But before we get deep dive into record linkage, let's sharpen our understanding of string similarity and minimum edit distance.

**Minimum edit distance**

Minimum edit distance is a systematic way to identify how close 2 strings are. For example, let's take a look at the following two words: intention, and execution. The minimum edit distance between them is the least possible amount of steps, that could get us from the word intention to execution, with the available operations being inserting new characters, deleting them, substituting them, and transposing consecutive characters.



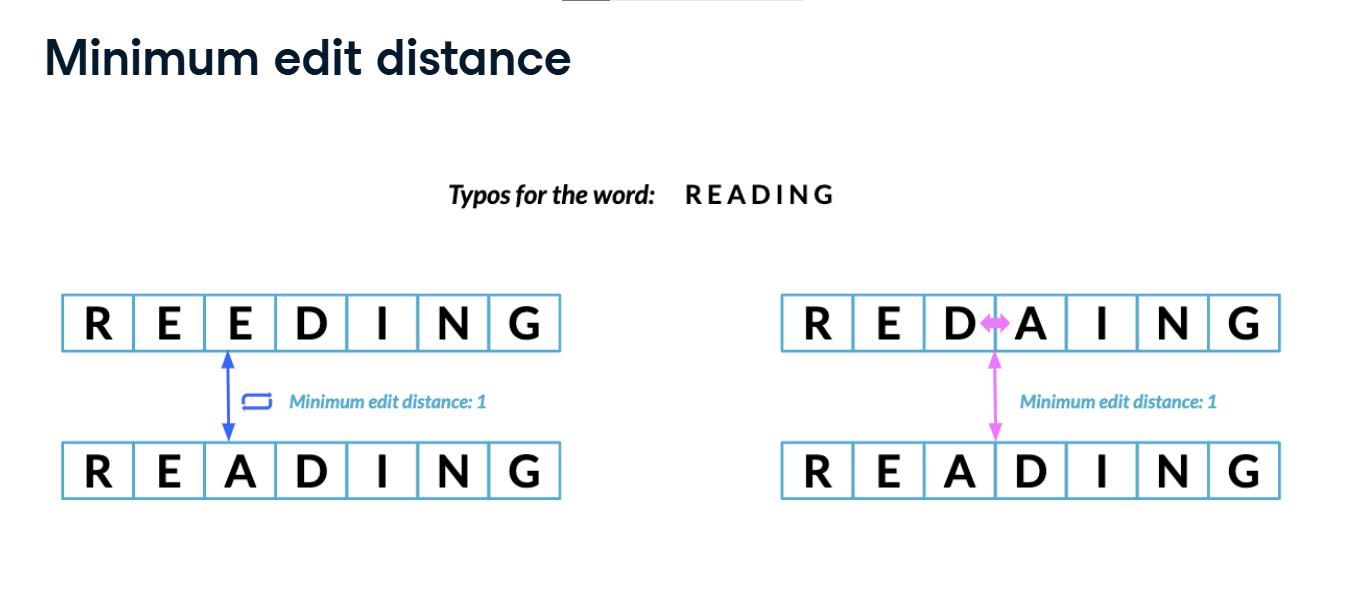
**Minimum edit distance**

To get from intention to execution,

We first start off by deleting I from intention, and adding C between E and N. Our minimum edit distance so far is 2, since these are two operations.

Then we substitute the first N with E, T with X, and N with U, leading us to execution! With the minimum edit distance being 5.

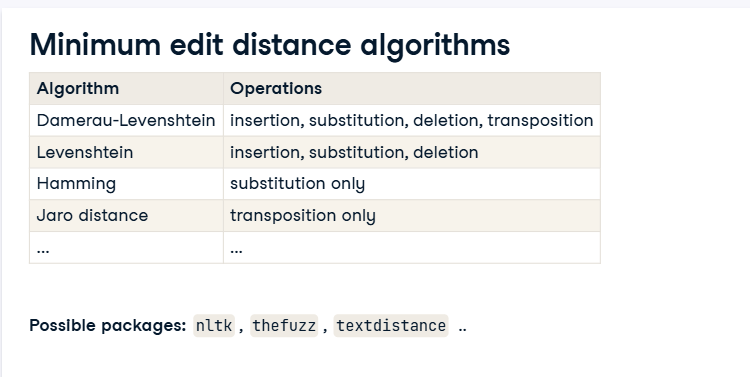
**Minimum edit distance**

The lower the edit distance, the closer two words are. For example, the two different typos of reading have a minimum edit distance of 1 between them and reading. 

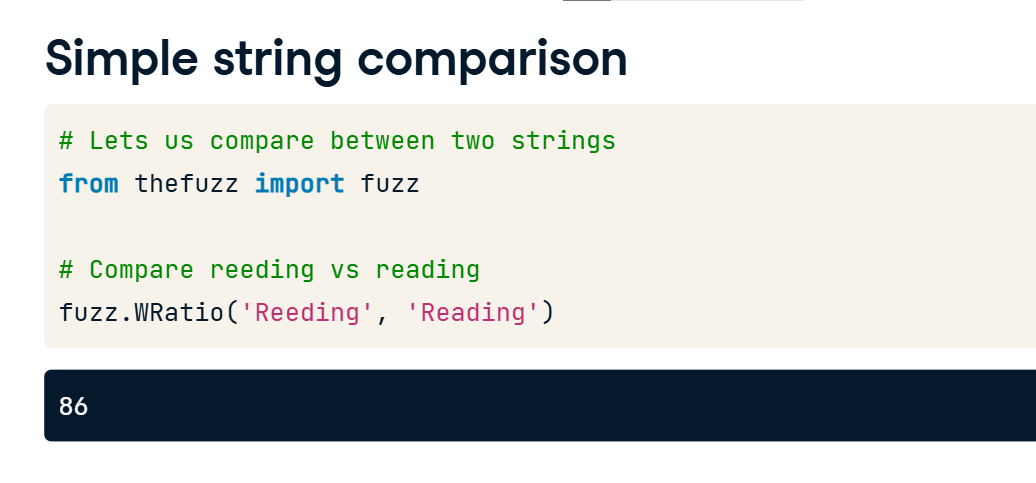
**Minimum edit distance algorithms**

There's a variety of algorithms based on edit distance that differ on which operations they use, how much weight attributed to each operation, which type of strings they're suited for and more, with a variety of packages to get each similarity.

For this lesson, we'll be comparing strings using Levenshtein distance since it's the most general form of string matching by using the thefuzz package.

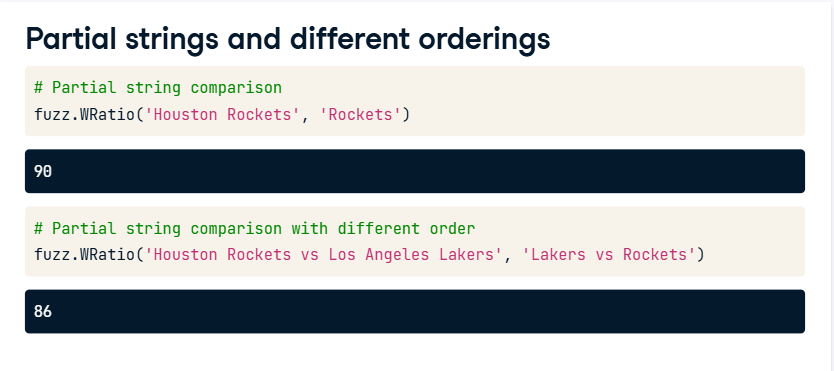


**Simple string comparison**

thefuzz is a package to perform string comparison. We first import fuzz from thefuzz, which allow us to compare between single strings. Here we use fuzz's WRatio function to compute the similarity between reading and its typo, inputting each string as an argument. For any comparison function using thefuzz, our output is a score from 0 to 100 with 0 being not similar at all, 100 being an exact match. Do not confuse this with the minimum edit distance score from earlier, where a lower minimum edit distance means a closer match. 

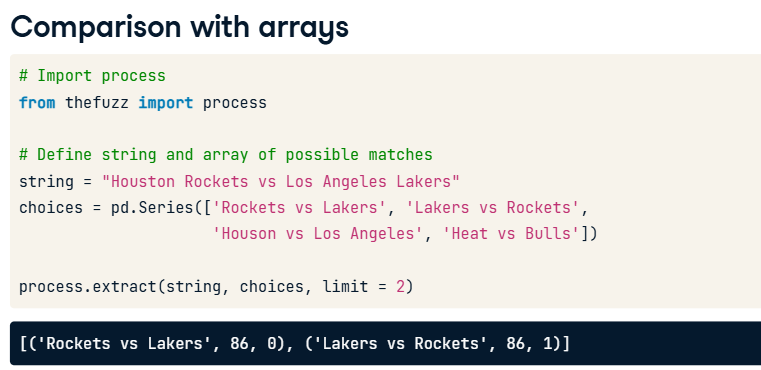
**Partial strings and different orderings**

The WRatio function is highly robust against partial string comparison with different orderings. For example here we compare the strings Houston Rockets and Rockets, and still receive a high similarity score. The same can be said for the strings Houston Rockets vs Los Angeles Lakers and Lakers vs Rockets, where the team names are only partial and they are differently ordered.



**Comparison with arrays**

We can also compare a string with an array of strings by using the extract function from the process module from fuzzy wuzzy. Extract takes in a string, an array of strings, and the number of possible matches to return ranked from highest to lowest. It returns a list of tuples with 3 elements, the first one being the matching string being returned, the second one being its similarity score, and the third one being its index in the array.

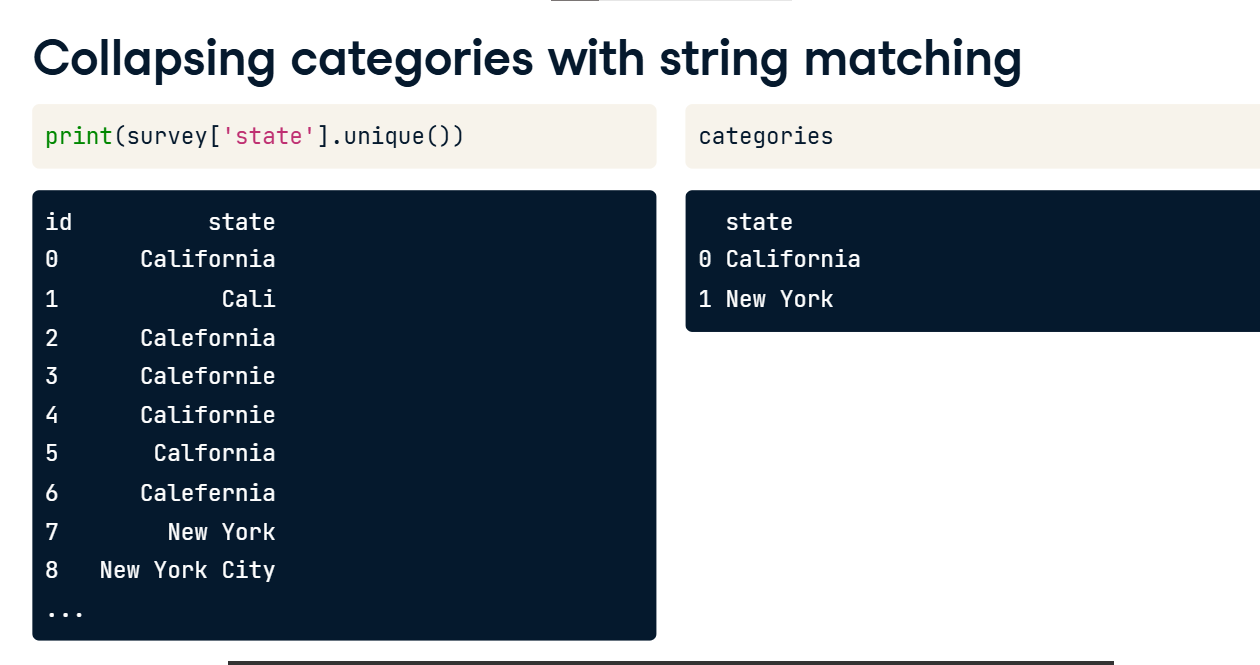


**Collapsing categories with string similarity**

In chapter 2, we learned that collapsing data into categories is an essential aspect of working with categorical and text data, and we saw how to manually replace categories in a column of a DataFrame. But what if we had so many inconsistent categories that a manual replacement is simply not feasible? We can easily do that with string similarity!

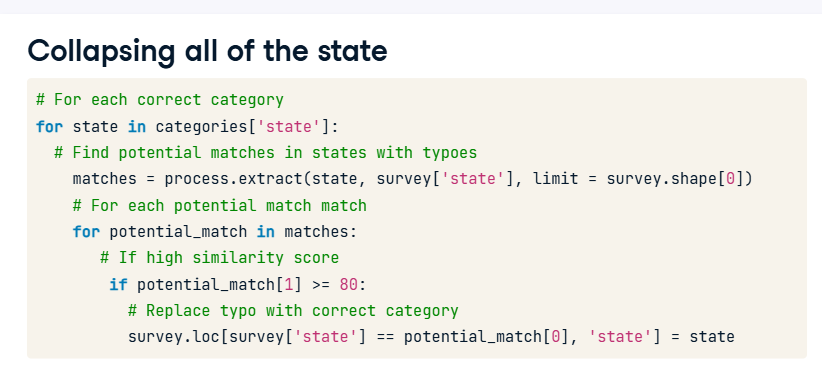
**Collapsing categories with string matching**

Say we have DataFrame named survey containing answers from respondents from the state of New York and California asking them how likely are you to move on a scale of 0 to 5. The state field was free text and contains hundreds of typos. Remapping them manually would take a huge amount of time. Instead, we'll use string similarity. We also have a category DataFrame containing the correct categories for each state. Let's collapse the incorrect categories with string matching!



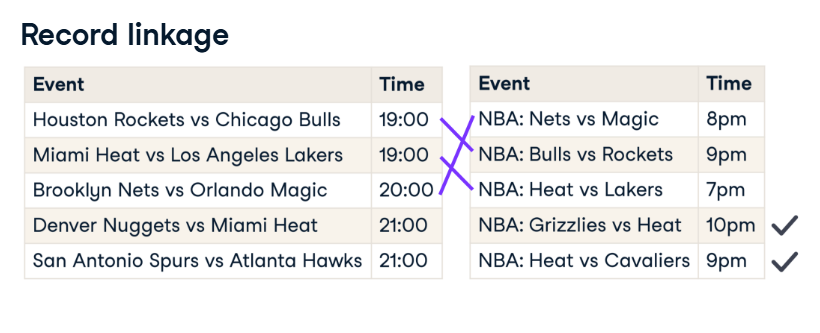
**Collapsing all of the state**

We first create a for loop iterating over each correctly typed state in the categories DataFrame. For each state, we find its matches in the state column of the survey DataFrame, returning all possible matches by setting the limit argument of extract to the length of the survey DataFrame. Then we iterate over each potential match, isolating the ones only with a similarity score higher or equal than 80 with an if statement. Then for each of those returned strings, we replace it with the correct state using the loc method.



**Record linkage**

Record linkage attempts to join data sources that have similarly fuzzy duplicate values, so that we end up with a final DataFrame with no duplicates by using string similarity. We'll cover record linkage in more detail in the next couple of lessons.



**Generating pairs**

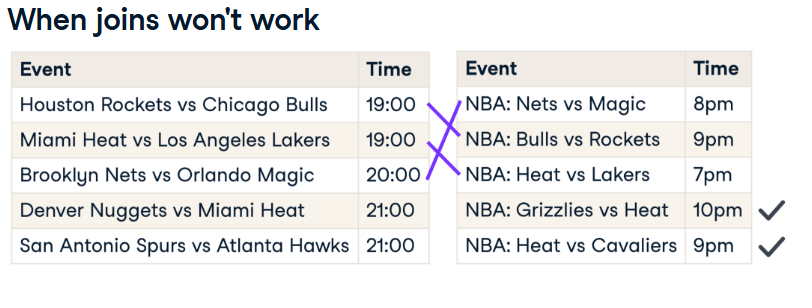
Great work with lesson 1 - you now have a solid understanding how to calculate string similarity.

**Motivation**

At the end of the last video exercise, we saw how record linkage attempts to join data sources with fuzzy duplicate values. For example here are two DataFrames containing NBA games and their schedules. They've both been scraped from different sites and we would want to merge them together and have one DataFrame containing all unique games.

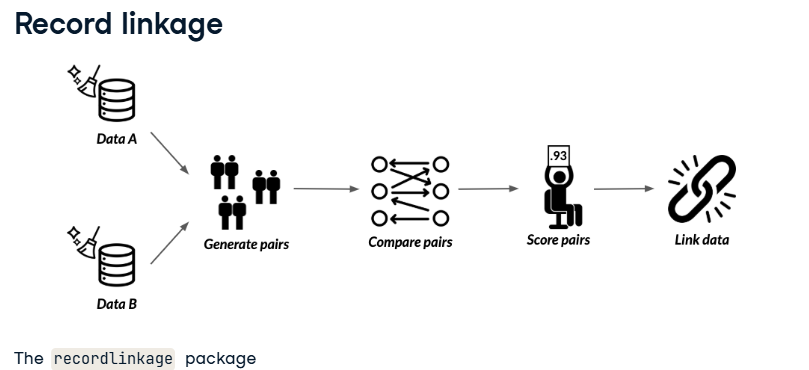
**When joins won't work**

We see that there are duplicates values in both DataFrames with different naming marked here in red, and non duplicate values, marked here in green. Since there are games happening at the same time, no common unique identifier between the DataFrames, and the events are differently named, a regular join or merge will not work. This is where record linkage comes in.



**Record linkage**

Record linkage is the act of linking data from different sources regarding the same entity. Generally, we clean two or more DataFrames, generate pairs of potentially matching records, score these pairs according to string similarity and other similarity metrics, and link them. All of these steps can be achieved with the recordlinkage package, let's find how!

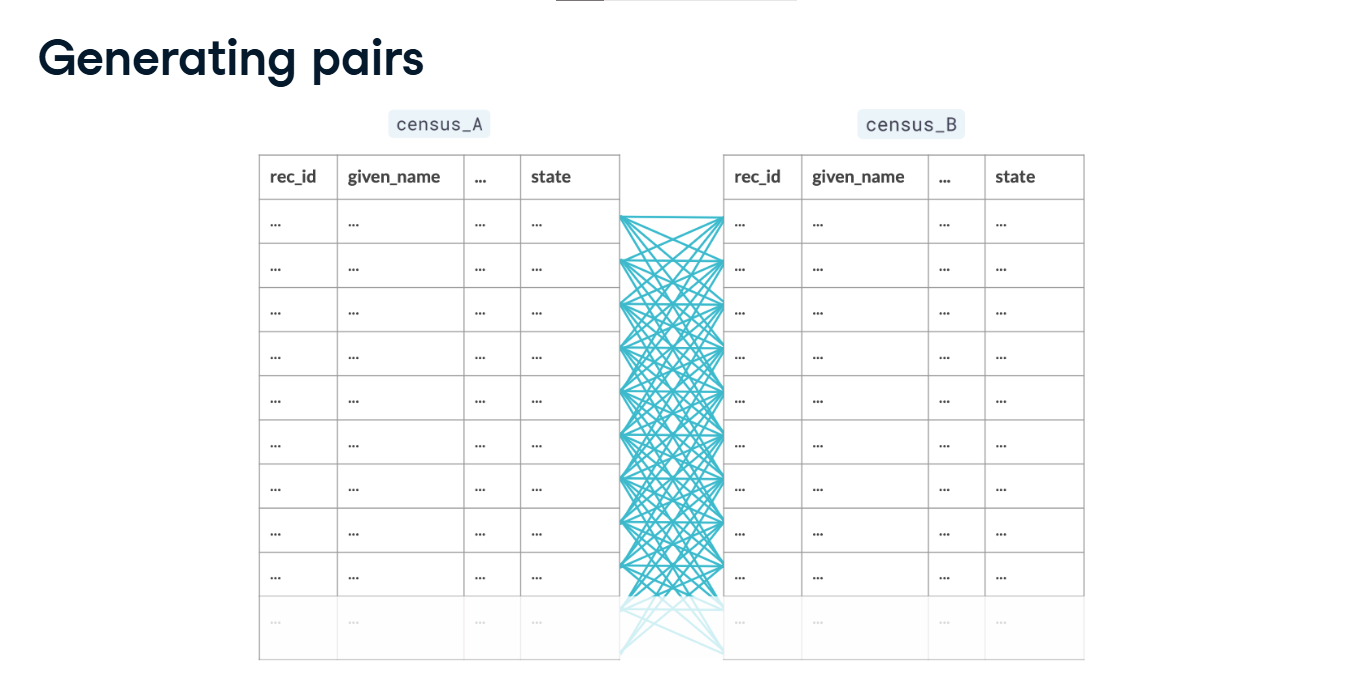


**Our DataFrames**

Here we have two DataFrames, census\_A, and census\_B, containing data on individuals throughout the states. We want to merge them while avoiding duplication using record linkage, since they are collected manually and are prone to typos, there are no consistent IDs between them.

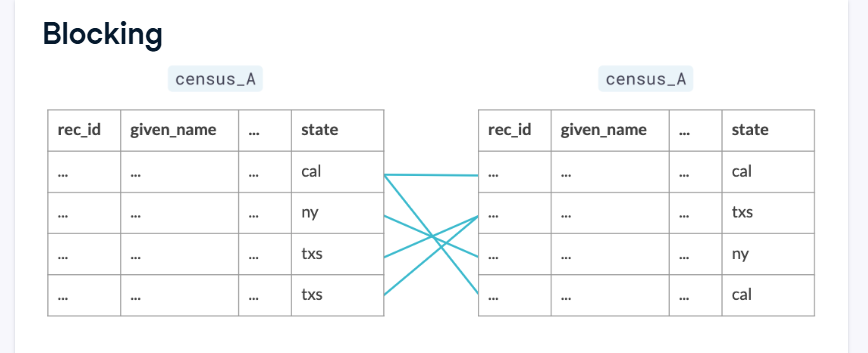
**Generating pairs**

We first want to generate pairs between both DataFrames. Ideally, we want to generate all possible pairs between our DataFrames. but what if we had big DataFrames and ended up having to generate millions if not billions of pairs? It wouldn't prove scalable and could seriously hamper development time.



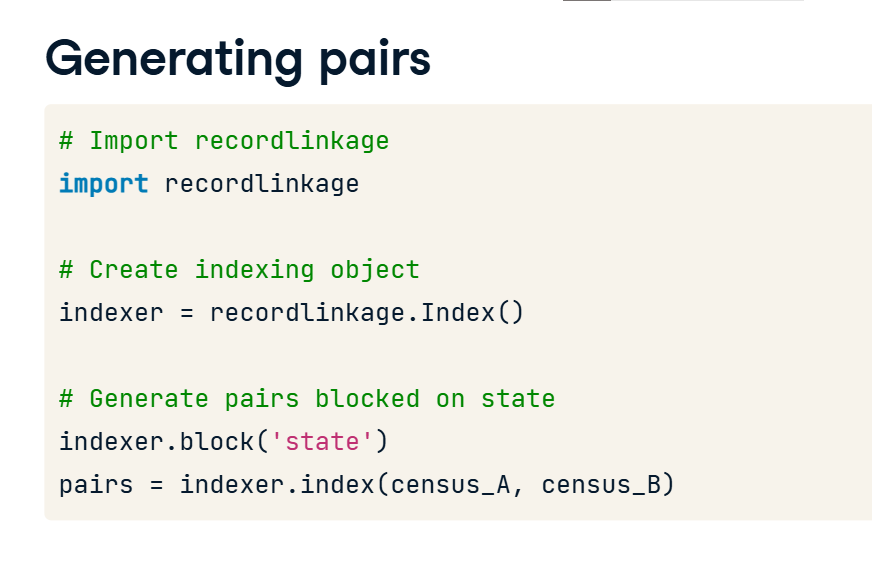
**Blocking**

This is where we apply what we call blocking, which creates pairs based on a matching column, which is in this case, the state column, reducing the number of possible pairs.



**Generating pairs**

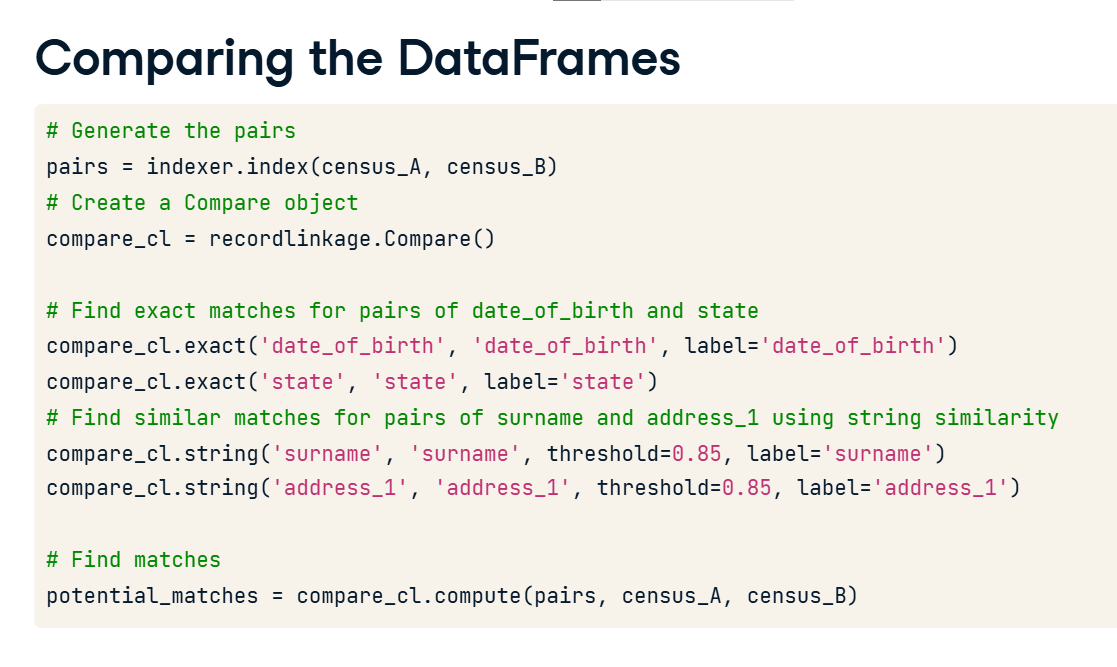
To do this, we first start off by importing recordlinkage. We then use the recordlinkage dot Index function, to create an indexing object. This essentially is an object we can use to generate pairs from our DataFrames. To generate pairs blocked on state, we use the block method, inputting the state column as input. Once the indexer object has been initialized, we generate our pairs using the dot index method, which takes in the two dataframes.



The resulting object, is a pandas multi index object containing pairs of row indices from both DataFrames, which is a fancy way to say it is an array containing possible pairs of indices that makes it much easier to subset DataFrames on.

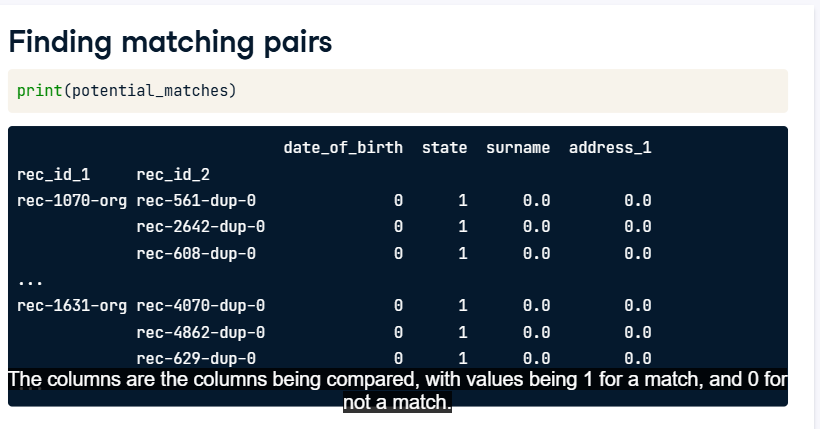
**Comparing the DataFrames**

Since we've already generated our pairs, it's time to find potential matches. We first start by creating a comparison object using the recordlinkage dot compare function. This is similar to the indexing object we created while generating pairs, but this one is responsible for assigning different comparison procedures for pairs. Let's say there are columns for which we want exact matches between the pairs. To do that, we use the exact method. It takes in the column name in question for each DataFrame, which is in this case date\_of\_birth and state, and a label argument which lets us set the column name in the resulting DataFrame. Now in order to compute string similarities between pairs of rows for columns that have fuzzy values, we use the dot string method, which also takes in the column names in question, the similarity cutoff point in the threshold argument, which takes in a value between 0 and 1, which we here set to 0.85. Finally to compute the matches, we use the compute function, which takes in the possible pairs, and the two DataFrames in question. Note that you need to always have the same order of DataFrames when inserting them as arguments when generating pairs, comparing between columns, and computing comparisons.



**Finding matching pairs**

The output is a multi index DataFrame, where the first index is the row index from the first DataFrame, or census A, and the second index is a list of all row indices in census B. The columns are the columns being compared, with values being 1 for a match, and 0 for not a match.



**Finding the only pairs we want**

To find potential matches, we just filter for rows where the sum of row values is higher than a certain threshold. Which in this case higher or equal to 2. But we'll dig deeper into these matches and see how to use them to link our census DataFrames in the next lesson.



**Linking DataFrames**

Awesome work on the first 2 lessons! You've made it to the last lesson of this course!

**Record linkage**

At this point, you've generated your pairs, compared them, and scored them.

Now it's time to link your data!

Remember our census DataFrames from the video of the previous lesson?

**What we've already done**

We've already generated pairs between them, compared four of their columns, two for exact matches and two for string similarity alongside a 0.85 threshold, and found potential matches.

**What we're doing now**

Now it's time to link both census DataFrames.

**Our potential matches**

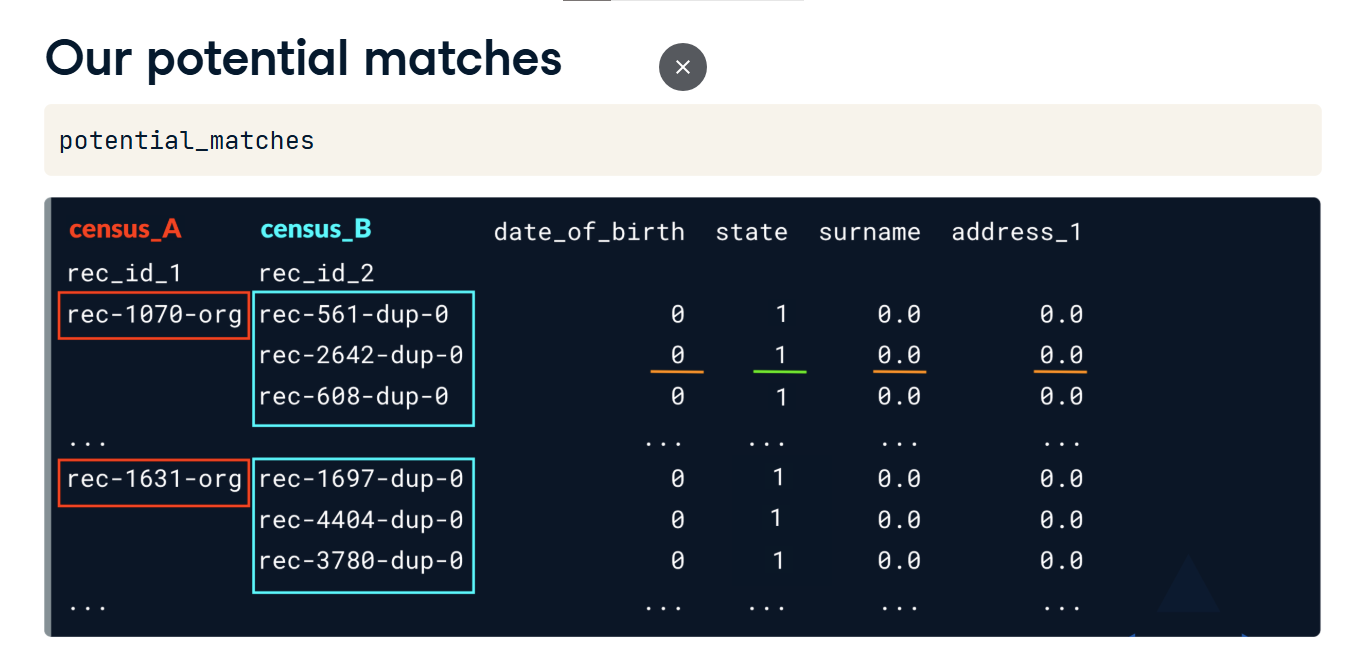
Let's look closely at our potential matches. It is a multi-index DataFrame, where we have two index columns, record id 1, and record id 2.

**Our potential matches**

The first index column, stores indices from census A.

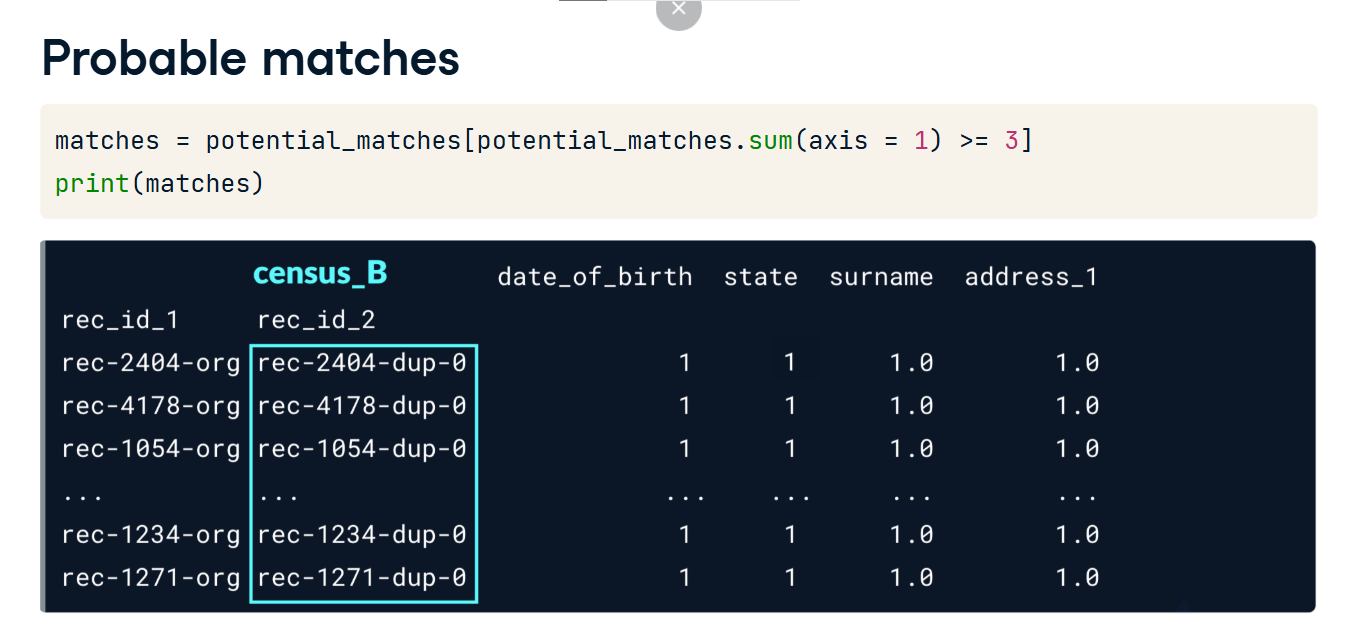
The second index column, stores all possible indices from census\_B, for each row index of census\_A.

The columns of our potential matches are the columns we chose to link both DataFrames on, where the value is 1 for a match, and 0 otherwise.



**Probable matches**

The first step in linking DataFrames, is to isolate the potentially matching pairs to the ones we're pretty sure of. We saw how to do this in the previous lesson, by subsetting the rows where the row sum is above a certain number of columns, in this case 3. The output is row indices between census A and census B that are most likely duplicates. Our next step is to extract the one of the index columns, and subsetting its associated DataFrame to filter for duplicates.



**Probable matches**

Here we choose the second index column, which represents row indices of census B. We want to extract those indices, and subset census\_B on them to remove duplicates with census\_A before appending them together.

**Get the indices**

We can access a DataFrame's index using the index attribute. Since this is a multi index DataFrame, it returns a multi index object containing pairs of row indices from census\_A and census\_B respectively. We want to extract all census\_B indices, so we chain it with the get\_level\_values method, which takes in which column index we want to extract its values. We can either input the index column's name, or its order, which is in this case 1.



**Linking DataFrames**

To find the duplicates in census B, we simply subset on all indices of census\_B, with the ones found through record linkage. You can choose to examine them further for similarity with their duplicates in census\_A, but if you're sure of your analysis, you can go ahead and find the non duplicates by repeating the exact same line of code, except by adding a tilde at the beginning of your subset. Now that you have your non duplicates, all you need is a simple append using the DataFrame append method of census A, and you have your linked Data!



**Linking DataFrames**

To recap, what we did was build on top of our previous work in generating pairs, comparing across columns and finding potential matches. We then isolated all possible matches, where there are matches across 3 columns or more, ensuring we tightened our search for duplicates across both DataFrames before we link them. Extracted the row indices of census\_B where there are duplicates. Found rows of census\_B where they are not duplicated with census\_A by using the tilde symbol. And linked both DataFrames for full census results!

