Comparing strings

CLEANING DATA IN PYTHON

Adel Nehme
Content Developer @ DataCamp

datacamp

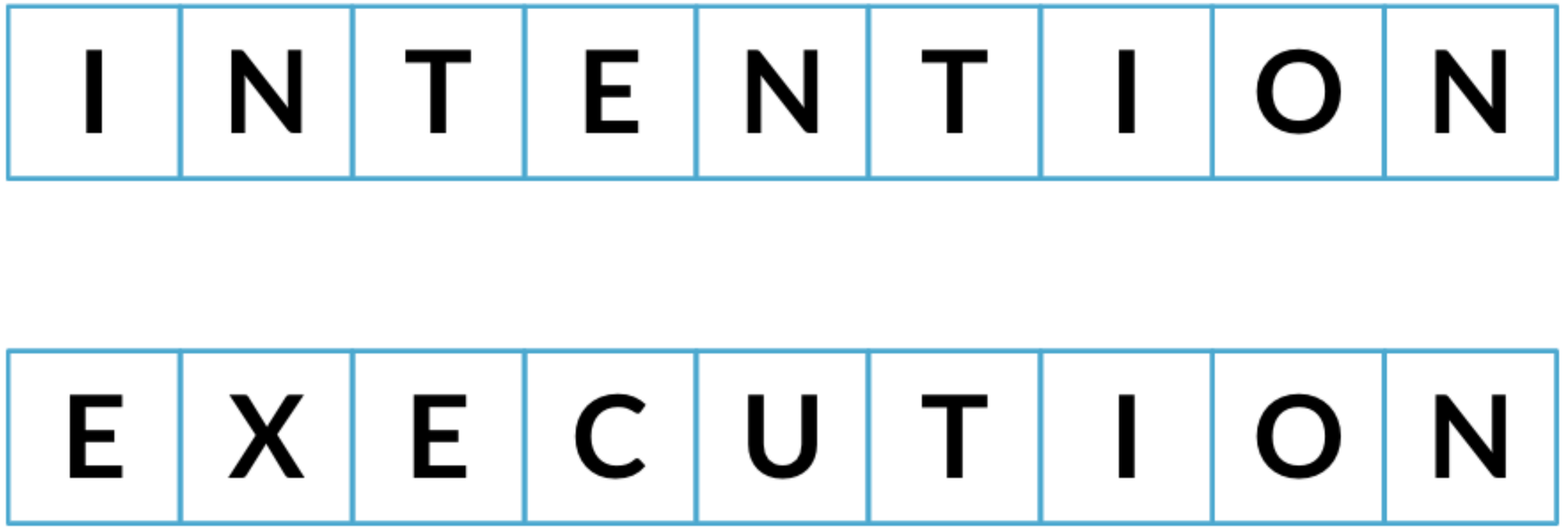
In this chapter

Chapter 4 - Record linkage

Minimum edit distance

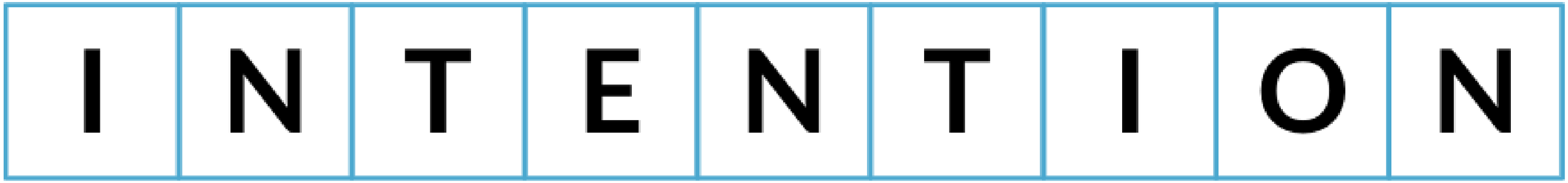
Least possible amount of steps needed to transition from one string to another

Minimum edit distance

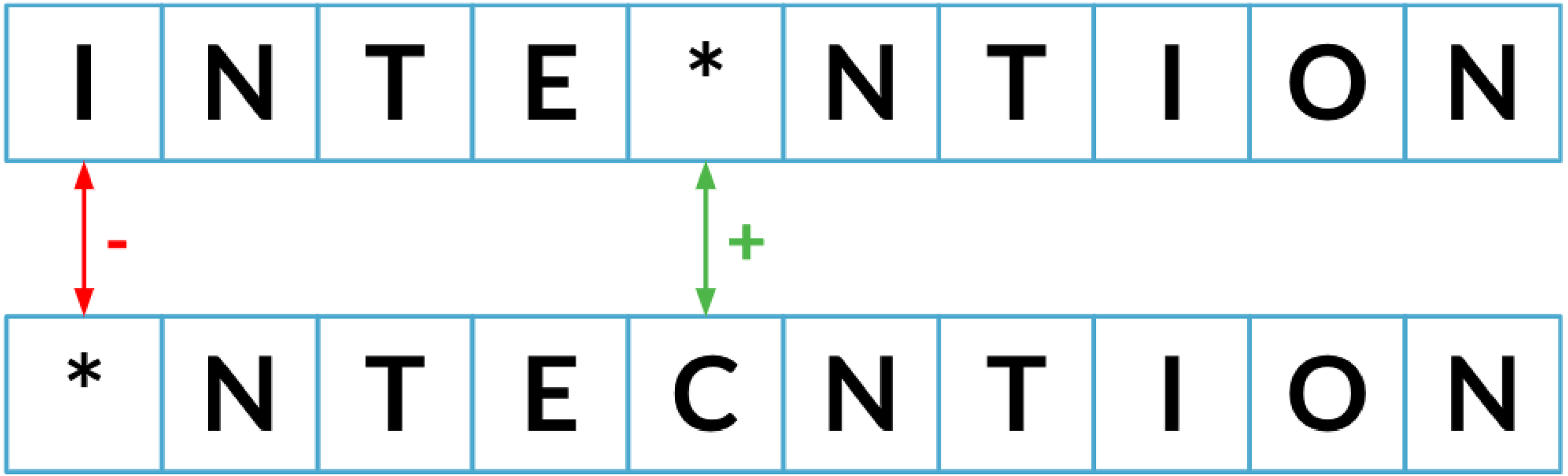


Least possible amount of steps needed to transition from one string to another

Minimum edit distance



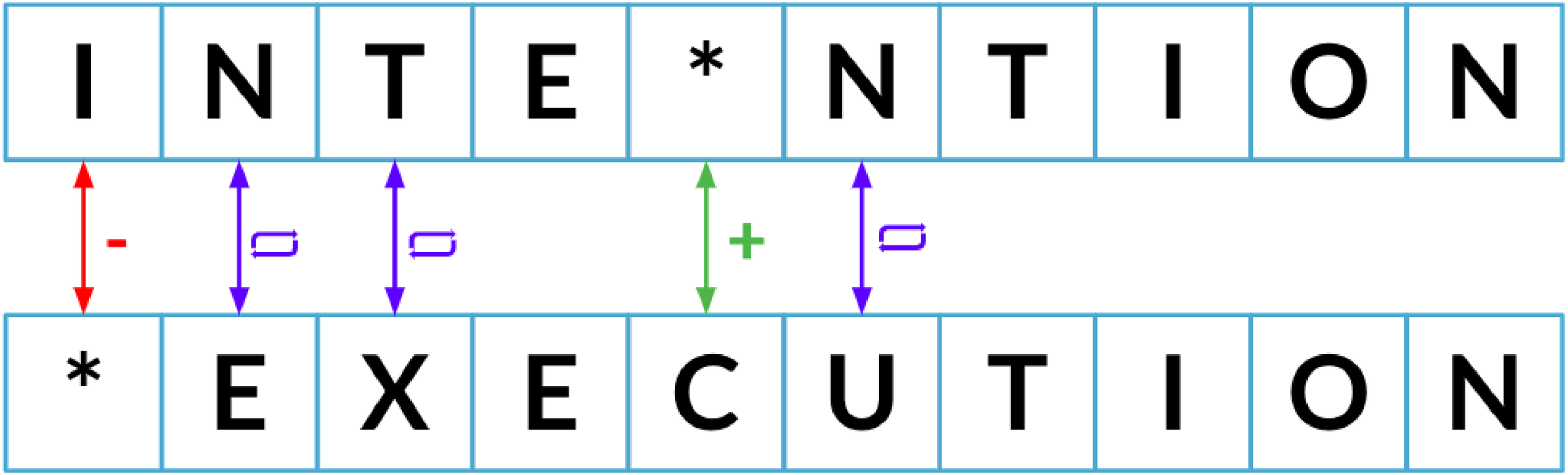
Minimum edit distance



Minimum edit distance so far: 2

5. Minimum edit distance
To get from intention to execution,
6. Minimum edit distance
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.

Minimum edit distance



Minimum edit distance: 5

7. Minimum edit distance
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

Typos for the word: READING

8. 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

Algorithm	Operations
Damerau-Levenshtein	insertion, substitution, deletion, transposition
Levenshtein	insertion, substitution, deletion
Hamming	substitution only
Jaro distance	transposition only
...	...

Possible packages: `nltk` , `fuzzywuzzy` , `textdistance` ..

9. Minimum edit distance algorithms
There's a variety of algorithms based on edit distance that differ on which operations they use, how

Minimum edit distance algorithms

Algorithm	Operations
Damerau-Levenshtein	insertion, substitution, deletion, transposition
Levenshtein	***insertion*, *substitution*, *deletion***
Hamming	substitution only
Jaro distance	transposition only
...	...

Possible packages: `fuzzywuzzy`

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

Simple string comparison

```
# Lets us compare between two strings
from fuzzywuzzy import fuzz


# Compare reeding vs reading
fuzz.WRatio('Reeding', 'Reading')
```

```
86
```

11. Simple string comparison
Fuzzywuzzy is a simple to use package to perform string comparison. We first import fuzz from fuzzywuzzy, 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 fuzzywuzzy, 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 earlier, where a lower minimum edit distance means a closer match.

Partial strings and different orderings

```
# Partial string comparison
fuzz.WRatio('Houston Rockets', 'Rockets')
```

```
90
```

```
# Partial string comparison with different order
fuzz.WRatio('Houston Rockets vs Los Angeles Lakers', 'Lakers vs Rockets')
```

```
86
```

12. 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

13. 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.

```
# Import process
from fuzzywuzzy import process

# Define string and array of possible matches
string = "Houston Rockets vs Los Angeles Lakers"
choices = pd.Series(['Rockets vs Lakers', 'Lakers vs Rockets',
                     'Houson vs Los Angeles', 'Heat vs Bulls'])

process.extract(string, choices, limit = 2)
```

```
[('Rockets vs Lakers', 86, 0), ('Lakers vs Rockets', 86, 1)]
```

Collapsing categories with string similarity

Chapter 2

Use `.replace()` to collapse `"eur"` into `"Europe"`

What if there are too many variations?

`"EU"`, `"eur"`, `"Europ"`, `"Europa"`, `"Erope"`, `"Evropa"` ...

14. 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!

String similarity!

Collapsing categories with string matching

```
print(survey['state'].unique())
```

```
id           state
0       California
1             Cali
2       Calefornia
3       Calefornie
4       Californie
5        Calfornia
6       Calefernia
7         New York
8   New York City
...
```

```
categories
```

```
  state
0 California
1 New York
```

15. 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

```
# For each correct category
for state in categories['state']:
  # Find potential matches in states with typoes
    matches = process.extract(state, survey['state'], limit = survey.shape[0])
    # For each potential match match
    for potential_match in matches:
       # If high similarity score
        if potential_match[1] >= 80:
          # Replace typo with correct category
          survey.loc[survey['state'] == potential_match[0], 'state'] = state
```

16. 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

Event	Time
Houston Rockets vs Chicago Bulls	19:00
Miami Heat vs Los Angeles Lakers	19:00
Brooklyn Nets vs Orlando Magic	20:00
Denver Nuggets vs Miami Heat	21:00
San Antonio Spurs vs Atlanta Hawks	21:00

Event	Time	
NBA: Nets vs Magic	8pm	
NBA: Bulls vs Rockets	9pm	
NBA: Heat vs Lakers	7pm	
NBA: Grizzlies vs Heat	10pm	✓
NBA: Heat vs Cavaliers	9pm	✓

17. 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**...

Let's practice!

CLEANING DATA IN PYTHON

Generating pairs

CLEANING DATA IN PYTHON

Adel Nehme
Content Developer @ DataCamp

datacamp

Motivation

Event	Time
Houston Rockets vs Chicago Bulls	19:00
Miami Heat vs Los Angeles Lakers	19:00
Brooklyn Nets vs Orlando Magic	20:00
Denver Nuggets vs Miami Heat	21:00
San Antonio Spurs vs Atlanta Hawks	21:00

Event	Time
NBA: Nets vs Magic	8pm
NBA: Bulls vs Rockets	9pm
NBA: Heat vs Lakers	7pm
NBA: Grizzlies vs Heat	10pm
NBA: Heat vs Cavaliers	9pm

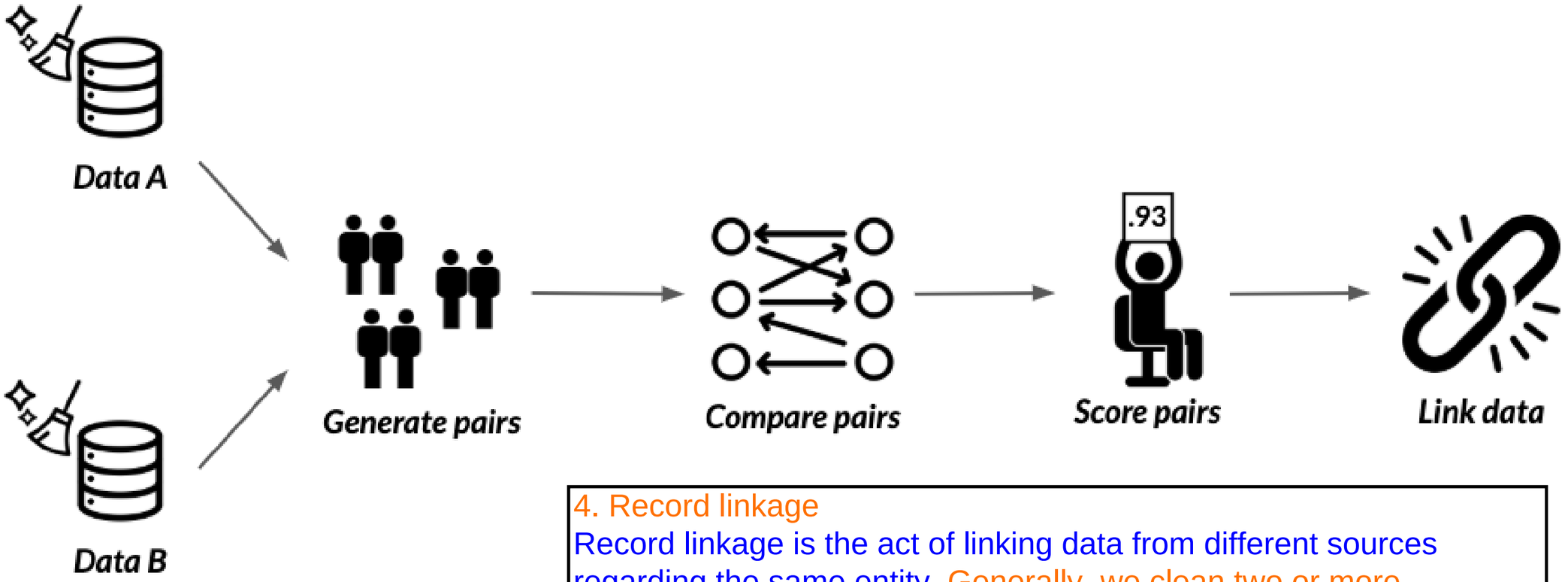
When joins won't work

Event	Time
Houston Rockets vs Chicago Bulls	19:00
Miami Heat vs Los Angeles Lakers	19:00
Brooklyn Nets vs Orlando Magic	20:00
Denver Nuggets vs Miami Heat	21:00
San Antonio Spurs vs Atlanta Hawks	21:00

Event	Time
NBA: Nets vs Magic	8pm
NBA: Bulls vs Rockets	9pm
NBA: Heat vs Lakers	7pm
NBA: Grizzlies vs Heat	10pm ✓
NBA: Heat vs Cavaliers	9pm ✓

3. 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



The `recordlinkage` package

4. 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

5. 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.

`census_A`

rec_id	given_name	surname	date_of_birth	suburb	state	address_1
rec-1070-org	michaela	neumann	19151111	winston hills	cal	stanley street
rec-1016-org	courtney	painter	19161214	richlands	txs	pinkerton circuit
...						

`census_B`

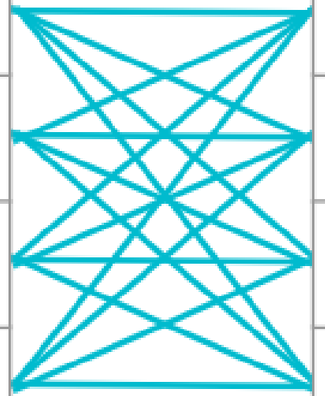
rec_id	given_name	surname	date_of_birth	suburb	state	address_1
rec-561-dup-0	elton	NaN	19651013	windermere	ny	light setreet
rec-2642-dup-0	mitchell	maxon	19390212	north ryde	cal	edkins street
...						

Generating pairs

6. Generating pairs
We first want to generate pairs between both DataFrames. Ideally, we want to generate all possible pairs between our DataFrames.

census_A

rec_id	given_name	...	state
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...



census_B

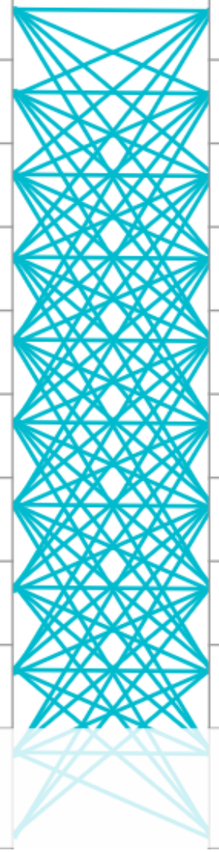
rec_id	given_name	...	state
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...

Generating pairs

7. Generating pairs
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.

census_A

rec_id	given_name	...	state
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...



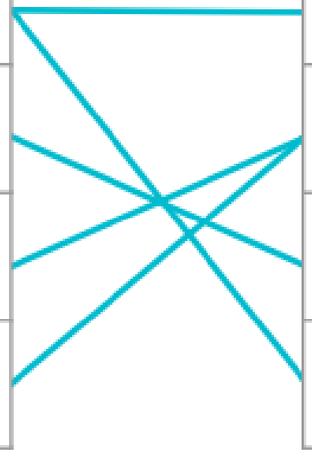
census_B

rec_id	given_name	...	state
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...

Blocking

census_A

rec_id	given_name	...	state
...	...	...	cal
...	...	...	ny
...	...	...	txs
...	...	...	txs



census_A

rec_id	given_name	...	state
...	...	...	cal
...	...	...	txs
...	...	...	ny
...	...	...	cal

8. 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

```
# Import recordlinkage
import recordlinkage


# Create indexing object
indexer = recordlinkage.Index()


# Generate pairs blocked on state
indexer.block('state')
pairs = indexer.index(census_A, census_B)
```

9. 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.

Generating pairs

```
print(pairs)
```

```
MultiIndex(levels=[['rec-1007-org', 'rec-1016-org', 'rec-1054-org', 'rec-1066-org',
'rec-1070-org', 'rec-1075-org', 'rec-1080-org', 'rec-110-org', 'rec-1146-org',
'rec-1157-org', 'rec-1165-org', 'rec-1185-org', 'rec-1234-org', 'rec-1271-org',
'rec-1280-org',.....
66, 14, 13, 18, 34, 39, 0, 16, 80, 50, 20, 69, 28, 25, 49, 77, 51, 85, 52, 63, 74, 61,
83, 91, 22, 26, 55, 84, 11, 81, 97, 56, 27, 48, 2, 64, 5, 17, 29, 60, 72, 47, 92, 12,
95, 15, 19, 57, 37, 70, 94]], names=['rec_id_1', 'rec_id_2'])
```

10. Generating pairs
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

```
# Generate the pairs
pairs = indexer.index(census_A, census_B)
# Create a Compare object
compare_cl = recordlinkage.Compare()

# Find exact matches for pairs of date_of_birth and state
compare_cl.exact('date_of_birth', 'date_of_birth', label='date_of_birth')
compare_cl.exact('state', 'state', label='state')
# Find similar matches for pairs of surname and address_1 using string similarity
compare_cl.string('surname', 'surname', threshold=0.85, label='surname')
compare_cl.string('address_1', 'address_1', threshold=0.85, label='address_1')

# Find matches
potential_matches = compare_cl.compute(pairs, census_A, census_B)
```

11. Comparing the DataFrames

Since we've already generated our pairs, it's time to find potential matches. . ... the similarity cutoff point in the threshold argument, which takes in a value between 0 and 1, which we here set to 0.85... 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

```
print(potential_matches)
```

```
                              date_of_birth  state  surname  address_1
rec_id_1     rec_id_2
rec-1070-org rec-561-dup-0                0      1      0.0        0.0
             rec-2642-dup-0               0      1      0.0        0.0
             rec-608-dup-0                0      1      0.0        0.0
...
rec-1631-org rec-4070-dup-0               0      1      0.0        0.0
             rec-4862-dup-0               0      1      0.0        0.0
             rec-629-dup-0                0      1      0.0        0.0
...
```

Finding the only pairs we want

```
potential_matches[potential_matches.sum(axis = 1) => 2]
```

		date_of_birth	state	surname	address_1
rec_id_1	rec_id_2				
rec-4878-org	rec-4878-dup-0	1	1	1.0	0.0
rec-417-org	rec-2867-dup-0	0	1	0.0	1.0
rec-3964-org	rec-394-dup-0	0	1	1.0	0.0
rec-1373-org	rec-4051-dup-0	0	1	1.0	0.0
	rec-802-dup-0	0	1	1.0	0.0
rec-3540-org	rec-470-dup-0	0	1	1.0	0.0

12. 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.

13. 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.

Let's practice!

CLEANING DATA IN PYTHON

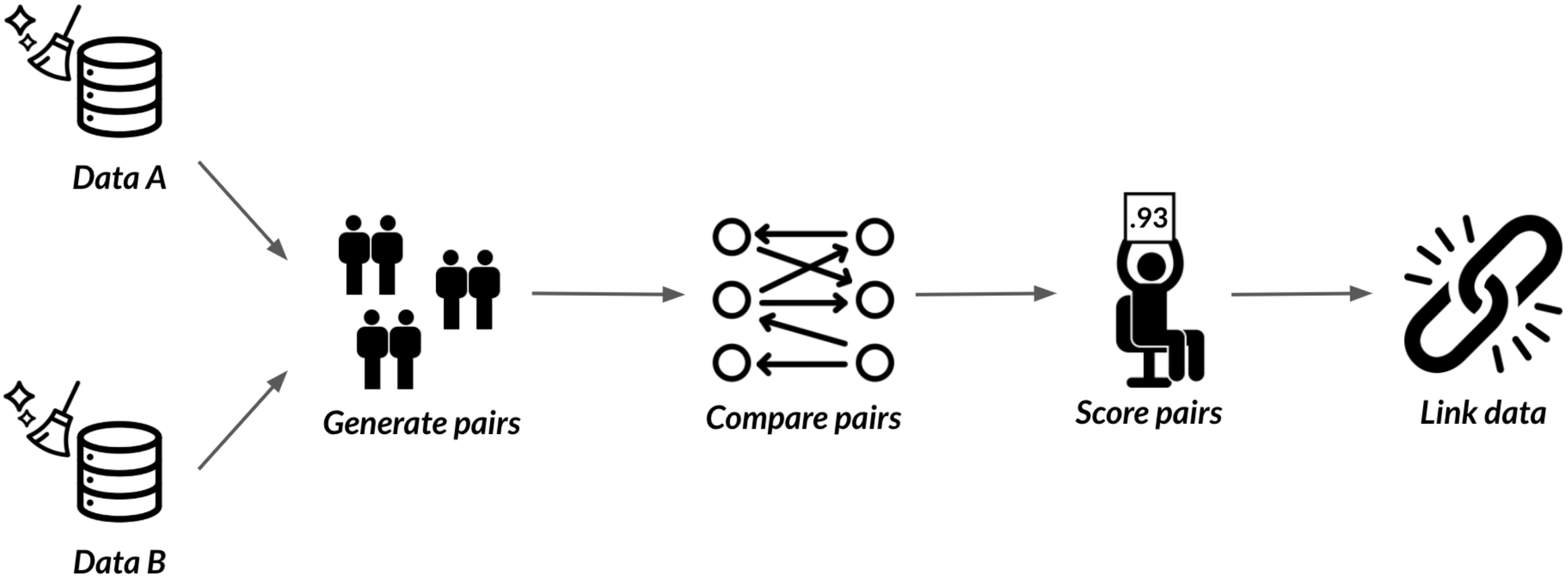
Linking DataFrames

CLEANING DATA IN PYTHON

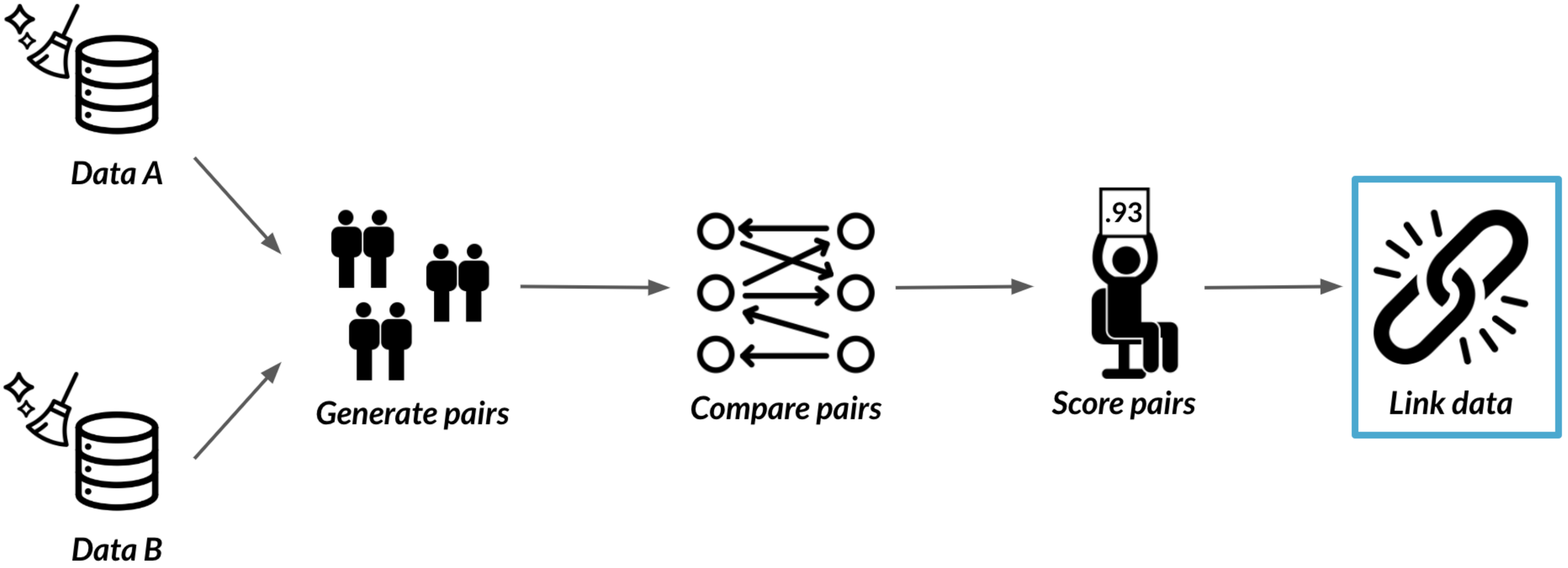
Adel Nehme
Content Developer @ DataCamp

datacamp

Record linkage



Record linkage



Our DataFrames

`census_A`

```
            given_name  surname date_of_birth         suburb state  address_1
rec_id
rec-1070-org   michaela  neumann      19151111  winston hills   nsw  stanley street
rec-1016-org   courtney  painter      19161214      richlands   vic  pinkerton circuit
...
```

`census_B`

```
              given_name  surname date_of_birth            suburb state  address_1
rec_id
rec-561-dup-0       elton      NaN      19651013        windermere   vic  light setreet
rec-2642-dup-0   mitchell    maxon      19390212        north ryde   nsw  edkins street
...
```

What we've already done

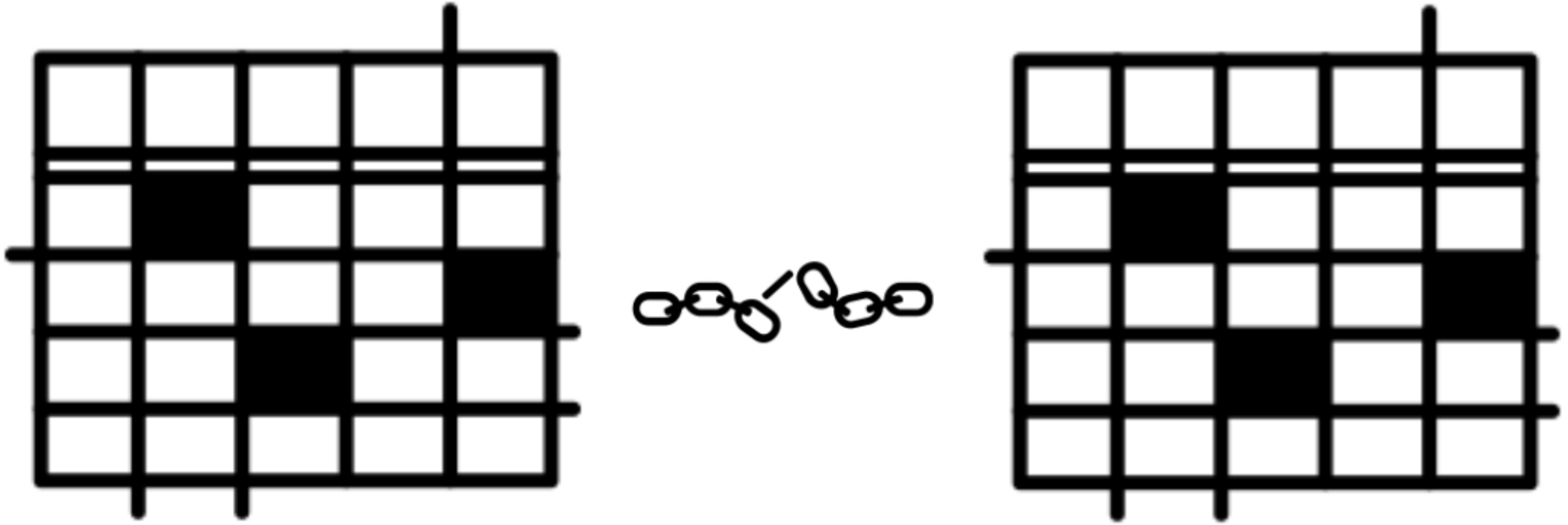
```
# Import recordlinkage and generate full pairs
import recordlinkage
indexer = recordlinkage.Index()
indexer.block('state')
full_pairs = indexer.index(census_A, census_B)

# Comparison step
compare_cl = recordlinkage.Compare()
compare_cl.exact('date_of_birth', 'date_of_birth', label='date_of_birth')
compare_cl.exact('state', 'state', label='state')
compare_cl.string('surname', 'surname', threshold=0.85, label='surname')
compare_cl.string('address_1', 'address_1', threshold=0.85, label='address_1')

potential_matches = compare_cl.compute(full_pairs, census_A, census_B)
```

5. 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



census_A

census_B

Our potential matches

```
potential_matches
```

		date_of_birth	state	surname	address_1
rec_id_1	rec_id_2				
rec-1070-org	rec-561-dup-0	0	1	0.0	0.0
	rec-2642-dup-0	0	1	0.0	0.0
	rec-608-dup-0	0	1	0.0	0.0
...		...	...	...	...
rec-1631-org	rec-1697-dup-0	0	1	0.0	0.0
	rec-4404-dup-0	0	1	0.0	0.0
	rec-3780-dup-0	0	1	0.0	0.0
...		...	...	...	...

Our potential matches

```
potential_matches
```

census_A		date_of_birth	state	surname	address_1
rec_id_1	rec_id_2				
rec-1070-org	rec-561-dup-0	0	1	0.0	0.0
	rec-2642-dup-0	0	1	0.0	0.0
	rec-608-dup-0	0	1	0.0	0.0
...		...	...	...	...
rec-1631-org	rec-1697-dup-0	0	1	0.0	0.0
	rec-4404-dup-0	0	1	0.0	0.0
	rec-3780-dup-0	0	1	0.0	0.0
...		...	...	...	...

Our potential matches

```
potential_matches
```

census_A	census_B	date_of_birth	state	surname	address_1
rec_id_1	rec_id_2				
rec-1070-org	rec-561-dup-0	0	1	0.0	0.0
	rec-2642-dup-0	0	1	0.0	0.0
	rec-608-dup-0	0	1	0.0	0.0
...		...	...	...	...
rec-1631-org	rec-1697-dup-0	0	1	0.0	0.0
	rec-4404-dup-0	0	1	0.0	0.0
	rec-3780-dup-0	0	1	0.0	0.0
...		...	...	...	...

Our potential matches

```
potential_matches
```

census_A	census_B	date_of_birth	state	surname	address_1
rec_id_1	rec_id_2				
rec-1070-org	rec-561-dup-0	0	1	0.0	0.0
	rec-2642-dup-0	0	1	0.0	0.0
	rec-608-dup-0	0	1	0.0	0.0
...		...	...	...	...
rec-1631-org	rec-1697-dup-0	0	1	0.0	0.0
	rec-4404-dup-0	0	1	0.0	0.0
	rec-3780-dup-0	0	1	0.0	0.0
...		...	...	...	...

Probable matches

```
matches = potential_matches[potential_matches.sum(axis = 1) >= 3]
print(matches)
```

```
                                date_of_birth  state  surname  address_1
rec_id_1      rec_id_2
rec-2404-org  rec-2404-dup-0               1      1      1.0        1.0
rec-4178-org  rec-4178-dup-0               1      1      1.0        1.0
rec-1054-org  rec-1054-dup-0               1      1      1.0        1.0
...           ...                        ...    ...      ...        ...
rec-1234-org  rec-1234-dup-0               1      1      1.0        1.0
rec-1271-org  rec-1271-dup-0               1      1      1.0        1.0
```

11. 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

```
matches = potential_matches[potential_matches.sum(axis = 1) >= 3]
print(matches)
```

	census_B	date_of_birth	state	surname	address_1
rec_id_1	rec_id_2				
rec-2404-org	rec-2404-dup-0	1	1	1.0	1.0
rec-4178-org	rec-4178-dup-0	1	1	1.0	1.0
rec-1054-org	rec-1054-dup-0	1	1	1.0	1.0
...	...	...	...	...	...
rec-1234-org	rec-1234-dup-0	1	1	1.0	1.0
rec-1271-org	rec-1271-dup-0	1	1	1.0	1.0

12. 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

13. 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.

```
matches.index
```

```
MultiIndex(levels=[['rec-1007-org', 'rec-1016-org', 'rec-1054-org', 'rec-1066-org',
'rec-1070-org', 'rec-1075-org', 'rec-1080-org', 'rec-110-org', ...
```

```
# Get indices from census_B only
duplicate_rows = matches.index.get_level_values(1)
print(census_B_index)
```

```
Index(['rec-2404-dup-0', 'rec-4178-dup-0', 'rec-1054-dup-0', 'rec-4663-dup-0',
       'rec-485-dup-0', 'rec-2950-dup-0', 'rec-1234-dup-0', ... , 'rec-299-dup-0'])
```

Linking DataFrames

```
# Finding duplicates in census_B
census_B_duplicates = census_B[census_B.index.isin(duplicate_rows)]


# Finding new rows in census_B
census_B_new = census_B[~census_B.index.isin(duplicate_rows)]
```

```
# Link the DataFrames!
full_census = census_A.append(census_B_new)
```

14. 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!

```
# Import recordlinkage and generate pairs and compare across columns
...
# Generate potential matches
potential_matches = compare_cl.compute(full_pairs, census_A, census_B)

# Isolate matches with matching values for 3 or more columns
matches = potential_matches[potential_matches.sum(axis = 1) >= 3]

# Get index for matching census_B rows only
duplicate_rows = matches.index.get_level_values(1)

# Finding new rows in census_B
census_B_new = census_B[~census_B.index.isin(duplicate_rows)]

# Link the DataFrames!
full_census = census_A.append(census_B_new)
```

15. Linking DataFrames
To recap, ... 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

Let's practice!

CLEANING DATA IN PYTHON

Congratulations!

CLEANING DATA IN PYTHON

Adel Nehme
Content Developer @ DataCamp

What we've learned

Diagnose dirty data

Side effects of dirty data

Clean data

What we've learned

Data Type Constraints

Strings
Numeric data
...

Data Range Constraints

Out of range data
Out of range dates
...

Uniqueness Constraints

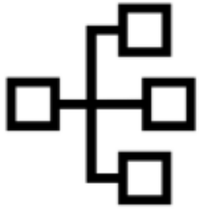
Finding duplicates
Treating them
...

Chapter 1 - Common data problems

What we've learned

Membership Constraints

Finding inconsistent categories
Treating them with joins
...



Categorical Variables

Finding inconsistent categories
Collapsing them into less
...

Cleaning Text Data

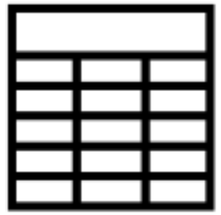
Unifying formats
Finding lengths
...

Chapter 2 - Text and categorical data problems

What we've learned

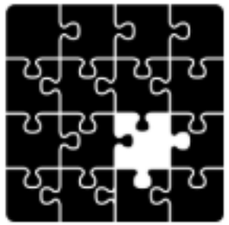
Uniformity

Unifying currency formats
Unifying date formats
...



Cross field validation

Summing across rows
Building assert functions
...

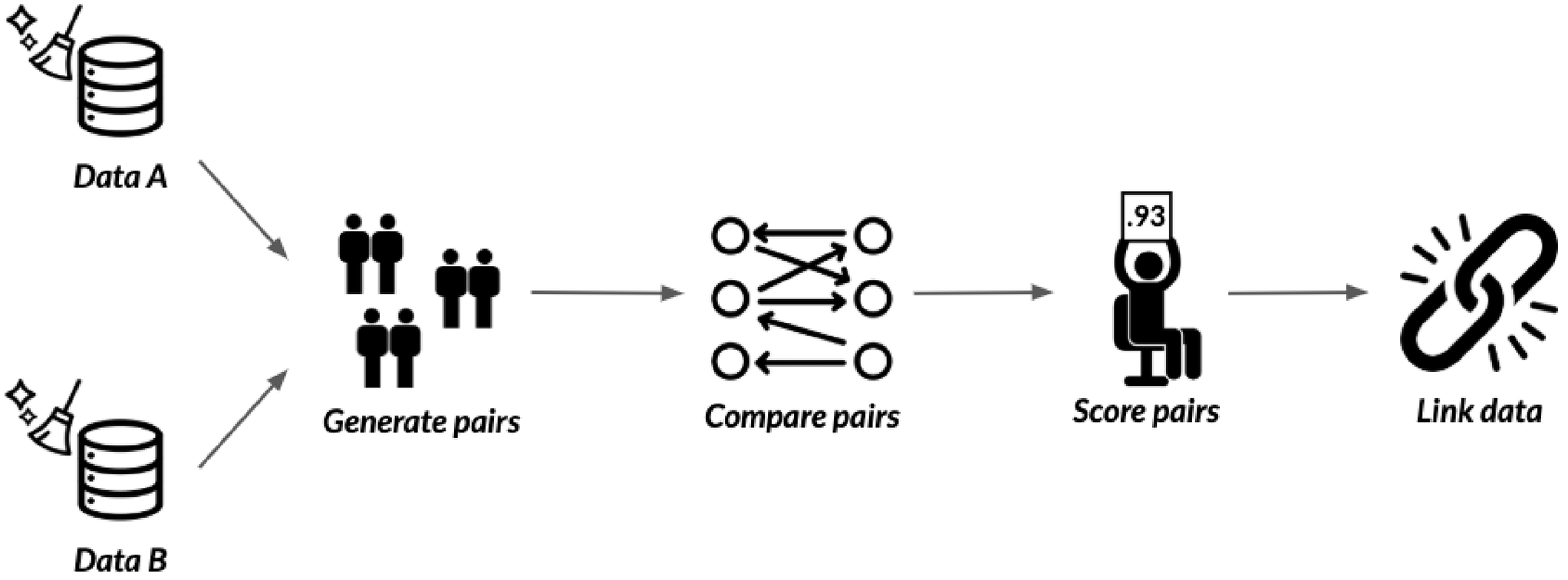


Completeness

Finding missing data
Treating them
...

Chapter 3 - Advanced data problems

What we've learned

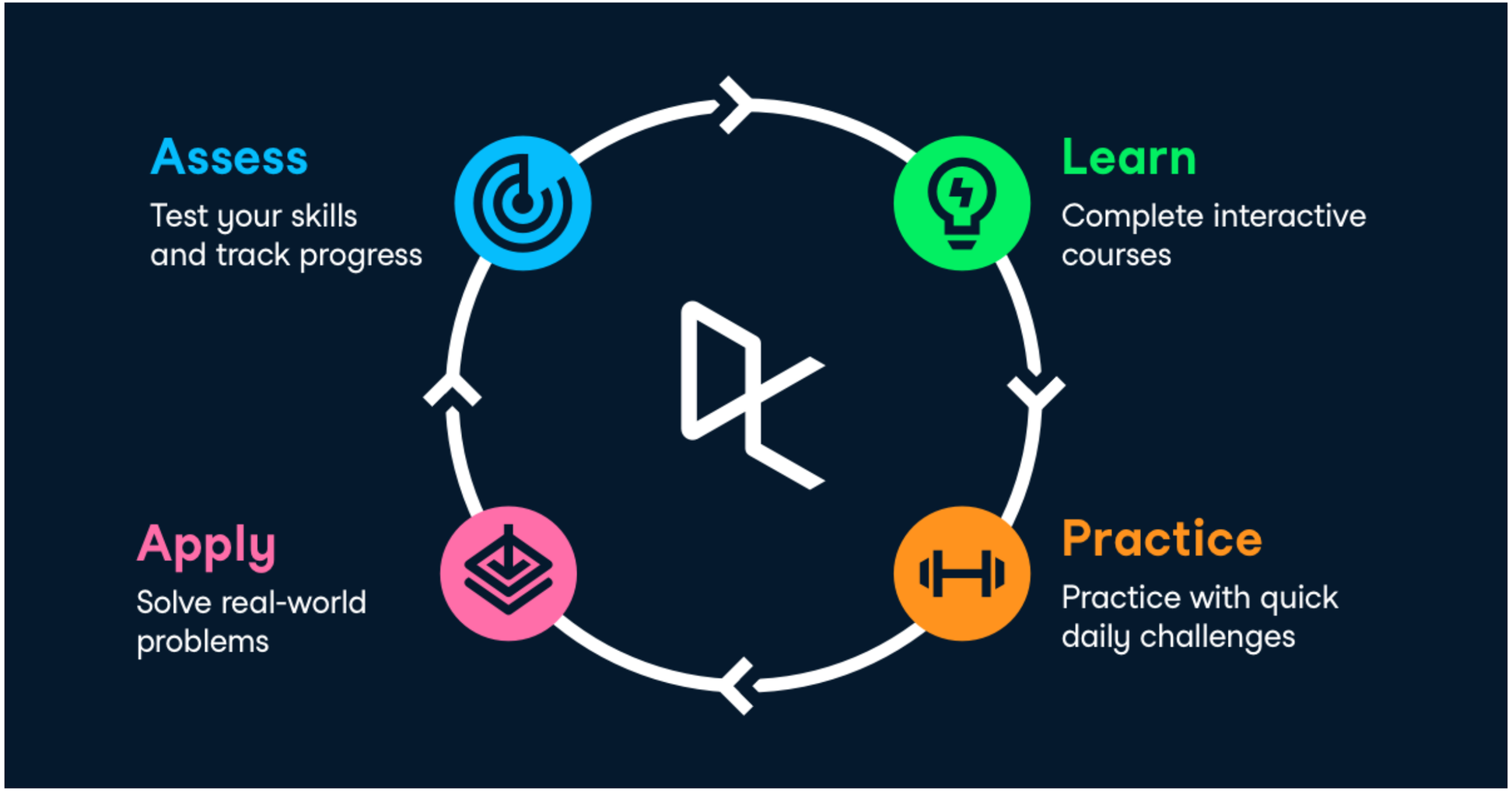


Chapter 4 - Record linkage

More to learn on DataCamp!

- **Working with Dates and Times in Python**
- **Regular Expressions in Python**
- **Dealing with Missing Data in Python**
- And more!

More to learn!



More to learn!

Thank you!

CLEANING DATA IN PYTHON