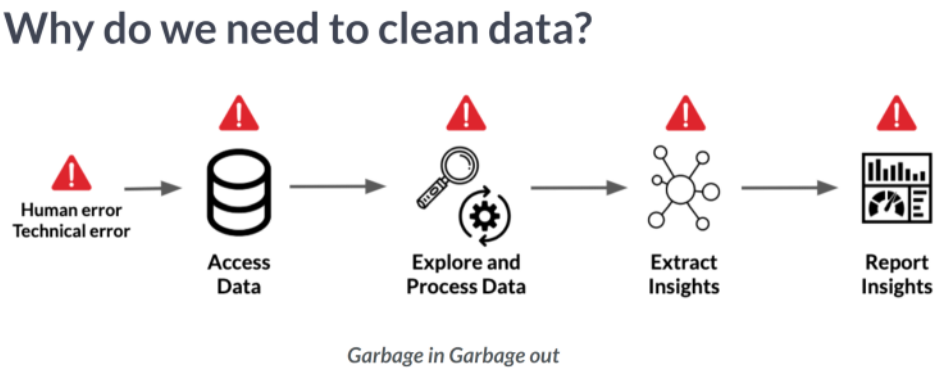
Cleaning Data In Python

Friday, 14 August 2020 2:37 PM

Common Data Problems



Data type constraints

Datatype	Example	Python data type
Text data	First name, last name, address ...	`str`
Integers	# Subscribers, # products sold ...	`int`
Decimals	Temperature, $ exchange rates ...	`float`
Binary	Is married, new customer, yes/no, ...	`bool`
Dates	Order dates, ship dates ...	`datetime`
Categories	Marriage status, gender ...	`category`

Strings to integers

```
# Import CSV file and output header
sales = pd.read_csv('sales.csv')
sales.head(2)
```

```
   SalesOrderID    Revenue    Quantity
0         43659     23153$          12
1         43660      1457$           2
```

```
# Get data types of columns
sales.dtypes
```

```
SalesOrderID    int64
Revenue         object
Quantity        int64
dtype: object
```

String to integers

```
# Get DataFrame information
sales.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 31465 entries, 0 to 31464
Data columns (total 3 columns):
SalesOrderID     31465 non-null int64
Revenue          31465 non-null object
Quantity         31465 non-null int64
dtypes: int64(2), object(1)
memory usage: 737.5+ KB
```

String to integers

```
# Print sum of all Revenue column
sales['Revenue'].sum()
```

```
'23153$1457$36865$32474$472$27510$16158$5694$6876$40487$807$6893$9153$6895$4216..
```

```
# Remove $ from Revenue column
sales['Revenue'] = sales['Revenue'].str.strip('$')
sales['Revenue'] = sales['Revenue'].astype('int')
```

```
# Verify that Revenue is now an integer
assert sales['Revenue'].dtype == 'int'
```

The assert statement

```
# This will pass
assert 1+1 == 2
```

```
# This will not pass
assert 1+1 == 3
```

```
AssertionError
          assert 1+1 == 3
AssertionError:
```

Numeric or categorical?

```
...   marriage_status    ...
...                 3    ...
...                 1    ...
...                 2    ...
```

`0` = Never married `1` = Married `2` = Separated `3` = Divorced

```
df['marriage_status'].describe()
```

```
       marriage_status
...
mean              1.4
std               0.20
min               0.00
50%               1.8 ...
```

Numeric or categorical?

```
# Convert to categorical
df["marriage_status"] = df["marriage_status"].astype('category')
df.describe()
```

```
       marriage_status
count              241
unique               4
top                  1
freq               120
```

Motivation

Can future sign-ups exist?

```
# Import date time
import datetime as dt
today_date = dt.date.today()
user_signups[user_signups['subscription_date'] > dt.date.today()]
```

```
   subscription_date  user_name         ...            Country
0         01/05/2021      Marah         ...              Nauru
1         09/08/2020     Joshua         ...            Austria
2         04/01/2020      Heidi         ...             Guinea
3         11/10/2020       Rina         ...       Turkmenistan
4         11/07/2020  Christine         ...    Marshall Islands
5         07/07/2020     Ayanna         ...              Gabon
```

How to deal with out of range data?

- Dropping data
- Setting custom minimums and maximums
- Treat as missing and impute
- Setting custom value depending on business assumptions

Movie example

```
import pandas as pd
# Output Movies with rating > 5
movies[movies['avg_rating'] > 5]
```

```
         movie_name  avg_rating
23  A Beautiful Mind           6
65   La Vita e Bella           6
77            Amelie           6
```

```
# Drop values using filtering
movies = movies[movies['avg_rating'] <= 5]
# Drop values using .drop()
movies.drop(movies[movies['avg_rating'] > 5].index, inplace = True)
# Assert results
assert movies['avg_rating'].max() <= 5
```

Movie example

```
# Convert avg_rating > 5 to 5
movies.loc[movies['avg_rating'] > 5, 'avg_rating'] = 5
```

```
# Assert statement
assert movies['avg_rating'].max() <= 5
```

Remember, no output means it passed

Date range example

```
import datetime as dt
import pandas as pd
# Output data types
user_signups.dtypes
```

```
subscription_date    object
user_name            object
Country              object
dtype: object
```

```
# Convert to DateTime
user_signups['subscription_date'] = pd.to_datetime(user_signups['subscription_date'])
```

```
# Assert that conversion happened
assert user_signups['subscription_date'].dtype == 'datetime64[ns]'
```

Date range example

```
today_date = dt.date.today()
```

Drop the data

```
# Drop values using filtering
user_signups = user_signups[user_signups['subscription_date'] < today_date]
# Drop values using .drop()
user_signups.drop(user_signups[user_signups['subscription_date'] > today_date].index, inplace = True)
```

Hardcode dates with upper limit

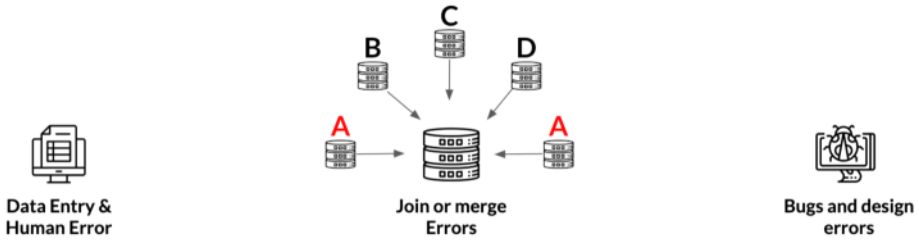
```
# Drop values using filtering
user_signups.loc[user_signups['subscription_date'] > today_date, 'subscription_date'] = today_date
# Assert is true
assert user_signups.subscription_date.max().date() <= today_date
```

What are duplicate values?

Most columns have the same values

first_name	last_name	address	height	weight
Justin	Saddlemyer	Boulevard du Jardin Botanique 3, Bruxelles	193 cm	87 kg
Justin	Saddlemyer	Boulevard du Jardin Botanique 3, Bruxelles	***194 cm***	87 kg

Why do they happen?



How to find duplicate values?

```
# Get duplicates across all columns
duplicates = height_weight.duplicated()
print(duplicates)
```

```
1       False
...     ....
22      True
23      False
...     ...
```

How to find duplicate values?

```
# Get duplicate rows
duplicates = height_weight.duplicated()
height_weight[duplicates]
```

	first_name	last_name	address	height	weight
100	Mary	Colon	4674 Ut Rd.	179	75
101	Ivor	Pierce	102-3364 Non Road	168	88
102	Cole	Palmer	8366 At, Street	178	91
103	Desirae	Shannon	P.O. Box 643, 5251 Consectetuer, Rd.	196	83

How to find duplicate rows?

The `.duplicated()` method

`subset` : List of column names to check for duplication.

`keep` : Whether to keep **first** (`'first'`), **last** (`'last'`) or **all** (`False`) duplicate values.

```
# Column names to check for duplication
column_names = ['first_name','last_name','address']
duplicates = height_weight.duplicated(subset = column_names, keep = False)
```

How to find duplicate rows?

```
# Output duplicate values
height_weight[duplicates]
```

	first_name	last_name	address	height	weight
1	Ivor	Pierce	102-3364 Non Road	168	66
22	Cole	Palmer	8366 At, Street	178	91
28	Desirae	Shannon	P.O. Box 643, 5251 Consectetuer, Rd.	195	83
37	Mary	Colon	4674 Ut Rd.	179	75
100	Mary	Colon	4674 Ut Rd.	179	75
101	Ivor	Pierce	102-3364 Non Road	168	88
102	Cole	Palmer	8366 At, Street	178	91
103	Desirae	Shannon	P.O. Box 643, 5251 Consectetuer, Rd.	196	83

How to find duplicate rows?

```
# Output duplicate values
height_weight[duplicates].sort_values(by = 'first_name')
```

	first_name	last_name	address	height	weight
22	Cole	Palmer	8366 At, Street	178	91
102	Cole	Palmer	8366 At, Street	178	91
28	Desirae	Shannon	P.O. Box 643, 5251 Consectetuer, Rd.	195	83
103	Desirae	Shannon	P.O. Box 643, 5251 Consectetuer, Rd.	196	83
1	Ivor	Pierce	102-3364 Non Road	168	66
101	Ivor	Pierce	102-3364 Non Road	168	88
37	Mary	Colon	4674 Ut Rd.	179	75
100	Mary	Colon	4674 Ut Rd.	179	75

How to find duplicate rows?

```
# Output duplicate values
height_weight[duplicates].sort_values(by = 'first_name')
```

	first_name	last_name	address	height	weight
22	Cole	Palmer	8366 At, Street	178	91
102	Cole	Palmer	8366 At, Street	178	91
28	Desirae	Shannon	P.O. Box 643, 5251 Consectetuer, Rd.	195	83
103	Desirae	Shannon	P.O. Box 643, 5251 Consectetuer, Rd.	196	83
1	Ivor	Pierce	102-3364 Non Road	168	66
101	Ivor	Pierce	102-3364 Non Road	168	88
37	Mary	Colon	4674 Ut Rd.	179	75
100	Mary	Colon	4674 Ut Rd.	179	75

How to find duplicate rows?

```
# Output duplicate values
height_weight[duplicates].sort_values(by = 'first_name')
```

	first_name	last_name	address	height	weight
22	Cole	Palmer	8366 At, Street	178	91
102	Cole	Palmer	8366 At, Street	178	91
28	Desirae	Shannon	P.O. Box 643, 5251 Consectetuer, Rd.	195	83
103	Desirae	Shannon	P.O. Box 643, 5251 Consectetuer, Rd.	196	83
1	Ivor	Pierce	102-3364 Non Road	168	66
101	Ivor	Pierce	102-3364 Non Road	168	88
37	Mary	Colon	4674 Ut Rd.	179	75
100	Mary	Colon	4674 Ut Rd.	179	75

How to treat duplicate values?

The `.drop_duplicates()` method

`subset` : List of column names to check for duplication.

`keep` : Whether to keep **first** (`'first'`), **last** (`'last'`) or **all** (`False`) duplicate values.

`inplace` : Drop duplicated rows directly inside DataFrame without creating new object (`True`).

```
# Drop duplicates
height_weight.drop_duplicates(inplace = True)
```

How to treat duplicate values?

```
# Output duplicate values
column_names = ['first_name','last_name','address']
duplicates = height_weight.duplicated(subset = column_names, keep = False)
height_weight[duplicates].sort_values(by = 'first_name')
```

	first_name	last_name	address	height	weight
28	Desirae	Shannon	P.O. Box 643, 5251 Consectetuer, Rd.	195	83
103	Desirae	Shannon	P.O. Box 643, 5251 Consectetuer, Rd.	196	83
1	Ivor	Pierce	102-3364 Non Road	168	66
101	Ivor	Pierce	102-3364 Non Road	168	88

How to treat duplicate values?

The `.groupby()` and `.agg()` methods

```
# Group by column names and produce statistical summaries
column_names = ['first_name','last_name','address']
summaries = {'height': 'max', 'weight': 'mean'}
height_weight = height_weight.groupby(by = column_names).agg(summaries).reset_index()
# Make sure aggregation is done
duplicates = height_weight.duplicated(subset = column_names, keep = False)
height_weight[duplicates].sort_values(by = 'first_name')
```

first_name	last_name	address	height	weight

Text and Categorical data Problems

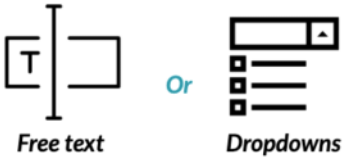
Categories and membership constraints

Predefined finite set of categories

Type of data	Example values	Numeric representation
Marriage Status	`unmarried` , `married`	`0` , `1`
Household Income Category	`0-20K` , `20-40K` ,...	`0` , `1` ,..
Loan Status	`default` , `payed` , `no_loan`	`0` , `1` , `2`

*Marriage status can **only** be* `unmarried` _or_ `married`

Why could we have these problems?



Free text Dropdowns

Data Entry Errors **Parsing Errors**

How do we treat these problems?

Dropping Data

Remapping Categories

Inferring Categories

An example

```
# Read study data and print it
study_data = pd.read_csv('study.csv')
study_data
```

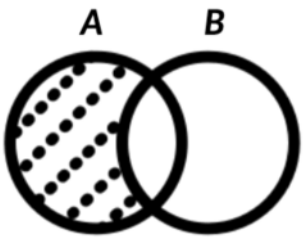
```
      name   birthday blood_type
1     Beth 2019-10-20         B-
2 Ignatius 2020-07-08         A-
3     Paul 2019-08-12         O+
4    Helen 2019-03-17         O-
5 Jennifer 2019-12-17         Z+
6  Kennedy 2020-04-27         A+
7    Keith 2019-04-19        AB+
```

```
# Correct possible blood types
categories
```

```
  blood_type
1         O-
2         O+
3         A-
4         A+
5         B+
6         B-
7        AB+
8        AB-
```

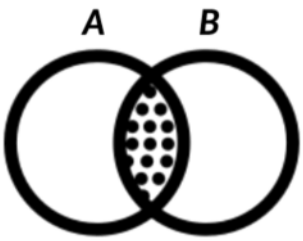
A note on joins

Anti Joins



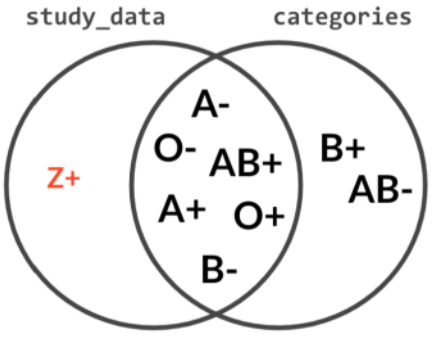
What is in A and not in B

Inner Joins



What is in <u>both</u> A and B

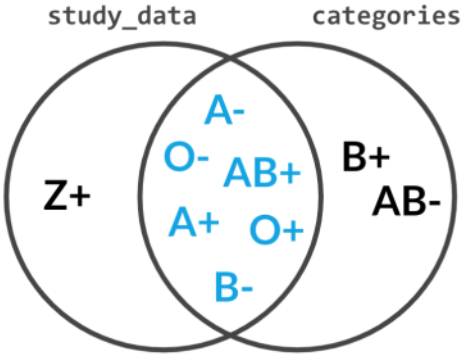
A left anti join on blood types



What is in `study_data` only

Returns only rows containing Z+

An inner join on blood types



What is in `study_data` and `categories` only

Returns all the rows except those containing Z+ , B+ and AB-

Finding inconsistent categories

```
inconsistent_categories = set(study_data['blood_type']).difference(categories['blood_type'])
print(inconsistent_categories)
```

```
{'Z+'}
```

```
# Get and print rows with inconsistent categories
inconsistent_rows = study_data['blood_type'].isin(inconsistent_categories)
study_data[inconsistent_rows]
```

```
      name   birthday blood_type
5 Jennifer 2019-12-17         Z+
```

Dropping inconsistent categories

```
inconsistent_categories = set(study_data['blood_type']).difference(categories['blood_type'])
inconsistent_rows = study_data['blood_type'].isin(inconsistent_categories)
inconsistent_data = study_data[inconsistent_rows]
# Drop inconsistent categories and get consistent data only
consistent_data = study_data[~inconsistent_rows]
```

```
      name   birthday blood_type
1     Beth 2019-10-20         B-
2 Ignatius 2020-07-08         A-
3     Paul 2019-08-12         O+
4    Helen 2019-03-17         O-
```

What type of errors could we have?

I) Value inconsistency

- *Inconsistent fields:* `'married'`, `'Maried'`, `'UNMARRIED'`, `'not married'` ..
- _Trailing white spaces: _ `'married '`, `' married '` ..

II) Collapsing too many categories to few

- *Creating new groups:* `0-20K`, `20-40K` categories ... from continuous household income data
- *Mapping groups to new ones:* Mapping household income categories to 2 `'rich'`, `'poor'`

III) Making sure data is of type `category` (seen in Chapter 1)

Value consistency

Capitalization: `'married'`, `'Married'`, `'UNMARRIED'`, `'unmarried'`..

```
# Get marriage status column
marriage_status = demographics['marriage_status']
marriage_status.value_counts()
```

```
unmarried    352
married      268
MARRIED      204
UNMARRIED    176
dtype: int64
```

Value consistency

```
# Get value counts on DataFrame
marriage_status.groupby('marriage_status').count()
```

	household_income	gender
marriage_status		
MARRIED	204	204
UNMARRIED	176	176
married	268	268
unmarried	352	352

Value consistency

```
# Capitalize
marriage_status['marriage_status'] = marriage_status['marriage_status'].str.upper()
marriage_status['marriage_status'].value_counts()
```

```
UNMARRIED    528
MARRIED      472
```

```
# Lowercase
marriage_status['marriage_status'] = marriage_status['marriage_status'].str.lower()
marriage_status['marriage_status'].value_counts()
```

```
unmarried    528
married      472
```

Value consistency

Trailing spaces: `'married '` , `'married'` , `'unmarried'` , `' unmarried'` ..

```
# Get marriage status column
marriage_status = demographics['marriage_status']
marriage_status.value_counts()
```

```
 unmarried   352
unmarried    268
married      204
married      176
dtype: int64
```

Value consistency

```
# Strip all spaces
demographics = demographics['marriage_status'].str.strip()
demographics['marriage_status'].value_counts()
```

```
unmarried    528
married      472
```

Collapsing data into categories

Create categories out of data: `income_group` column from `income` column.

```
# Using qcut()
import pandas as pd
group_names = ['0-200K', '200K-500K', '500K+']
demographics['income_group'] = pd.qcut(demographics['household_income'], q = 3,
                                       labels = group_names)
# Print income_group column
demographics[['income_group', 'household_income']]
```

```
     category  household_income
0   200K-500K  189243
1       500K+  778533
..
```

Collapsing data into categories

Create categories out of data: `income_group` column from `income` column.

```
# Using cut() - create category ranges and names
ranges = [0,200000,500000,np.inf]
group_names = ['0-200K', '200K-500K', '500K+']
# Create income group column
demographics['income_group'] = pd.cut(demographics['household_income'], bins=ranges,
                                      labels=group_names)
demographics[['income_group', 'household_income']]
```

```
     category  Income
0      0-200K  189243
1       500K+  778533
```

Collapsing data into categories

Map categories to fewer ones: reducing categories in categorical column.

`operating_system` column is: `'Microsoft', 'MacOS', 'IOS', 'Android', 'Linux'`

`operating_system` column should become: `'DesktopOS', 'MobileOS'`

```
# Create mapping dictionary and replace
mapping = {'Microsoft':'DesktopOS', 'MacOS':'DesktopOS', 'Linux':'DesktopOS',
           'IOS':'MobileOS', 'Android':'MobileOS'}
devices['operating_system'] = devices['operating_system'].replace(mapping)
devices['operating_system'].unique()
```

```
array(['DesktopOS', 'MobileOS'], dtype=object)
```

What is text data?

Type of data	Example values
Names	`Alex` , `Sara` ...
Phone numbers	`+96171679912` ...
Emails	`adel@datacamp.com`..
Passwords	...

Common text data problems

1) Data inconsistency:

`+96171679912` or `0096171679912` or ..?

2) Fixed length violations:

Passwords needs to be at least 8 characters

3) Typos:

`+961.71.679912`

Example

```
phones = pd.read_csv('phones.csv')
print(phones)
```

```
             Full name      Phone number
0       Noelani A. Gray  001-702-397-5143
1        Myles Z. Gomez  001-329-485-0540
2          Gil B. Silva  001-195-492-2338
3    Prescott D. Hardin   +1-297-996-4904
4    Benedict G. Valdez  001-969-820-3536
5      Reece M. Andrews              4138
6        Hayfa E. Keith  001-536-175-8444
7       Hedley I. Logan  001-681-552-1823
8      Jack W. Carrillo  001-910-323-5265
9       Lionel M. Davis  001-143-119-9210
```

Inconsistent data format

Length violation

Example

```
phones = pd.read_csv('phones.csv')
print(phones)
```

```
             Full name   Phone number
0       Noelani A. Gray  0017023975143
1        Myles Z. Gomez  0013294850540
2          Gil B. Silva  0011954922338
3    Prescott D. Hardin  0012979964904
4    Benedict G. Valdez  0019698203536
5      Reece M. Andrews            NaN
6        Hayfa E. Keith  0015361758444
7       Hedley I. Logan  0016815521823
8      Jack W. Carrillo  0019103235265
9       Lionel M. Davis  0011431199210
```

Fixing the phone number column

```
# Replace "+" with "00"
phones["Phone number"] = phones["Phone number"].str.replace("+", "00")
phones
```

	Full name	Phone number
0	Noelani A. Gray	001-702-397-5143
1	Myles Z. Gomez	001-329-485-0540
2	Gil B. Silva	001-195-492-2338
3	Prescott D. Hardin	001-297-996-4904
4	Benedict G. Valdez	001-969-820-3536
5	Reece M. Andrews	4138
6	Hayfa E. Keith	001-536-175-8444
7	Hedley I. Logan	001-681-552-1823
8	Jack W. Carrillo	001-910-323-5265
9	Lionel M. Davis	001-143-119-9210

Fixing the phone number column

```
# Replace "-" with nothing
phones["Phone number"] = phones["Phone number"].str.replace("-", "")
phones
```

	Full name	Phone number
0	Noelani A. Gray	0017023975143
1	Myles Z. Gomez	0013294850540
2	Gil B. Silva	0011954922338
3	Prescott D. Hardin	0012979964904
4	Benedict G. Valdez	0019698203536
5	Reece M. Andrews	4138
6	Hayfa E. Keith	0015361758444
7	Hedley I. Logan	0016815521823
8	Jack W. Carrillo	0019103235265
9	Lionel M. Davis	0011431190210

Fixing the phone number column

```
# Replace phone numbers with lower than 10 digits to NaN
digits = phones['Phone number'].str.len()
phones.loc[digits < 10, "Phone number"] = np.nan
phones
```

```
              Full name   Phone number
0       Noelani A. Gray  0017023975143
1        Myles Z. Gomez  0013294850540
2          Gil B. Silva  0011954922338
3    Prescott D. Hardin  0012979964904
4    Benedict G. Valdez  0019698203536
5      Reece M. Andrews           NaN
6        Hayfa E. Keith  0015361758444
7       Hedley I. Logan  0016815521823
8      Jack W. Carrillo  0019103235265
```

Fixing the phone number column

```
# Find length of each row in Phone number column
sanity_check = phone['Phone number'].str.len()
```

```
# Assert minmum phone number length is 10
assert sanity_check.min() >= 10
```

```
# Assert all numbers do not have "+" or "-"
assert phone['Phone number'].str.contains("+|-").any() == False
```

Remember, `assert` *returns nothing if the condition passes*

We set the pattern plus bar pipe minus, the bar pipe here is basically an or statement, so we're trying to find matches for either symbols. We chain it with the any method which returns True if any element in the output of our dot-str-contains is True, and test whether the it returns False.

But what about more complicated examples?

```
phones.head()
```

```
          Full name    Phone number
0     Olga Robinson  +(01706)-25891
1       Justina Kim    +0500-571437
2    Tamekah Henson      +0800-1111
3     Miranda Solis   +07058-879063
4  Caldwell Gilliam  +(016977)-8424
```

Regular expressions in action

```
# Replace letters with nothing
phones['Phone number'] = phones['Phone number'].str.replace(r'\D+', '')
phones.head()
```

```
          Full name Phone number
0     Olga Robinson   0170625891
1       Justina Kim   0500571437
2    Tamekah Henson     08001111
3     Miranda Solis  07058879063
4  Caldwell Gilliam   0169778424
```

To do this, we use the dot str dot replace method with the pattern we want to replace with an empty string.

Advance Data Problems

Uniformity

Column	Unit
Temperature	`32°C` is also `89.6°F`
Weight	`70 Kg` is also `11 st.`
Date	`26-11-2019` is also `26, November, 2019`
Money	`100$` is also `10763.90¥`

An example

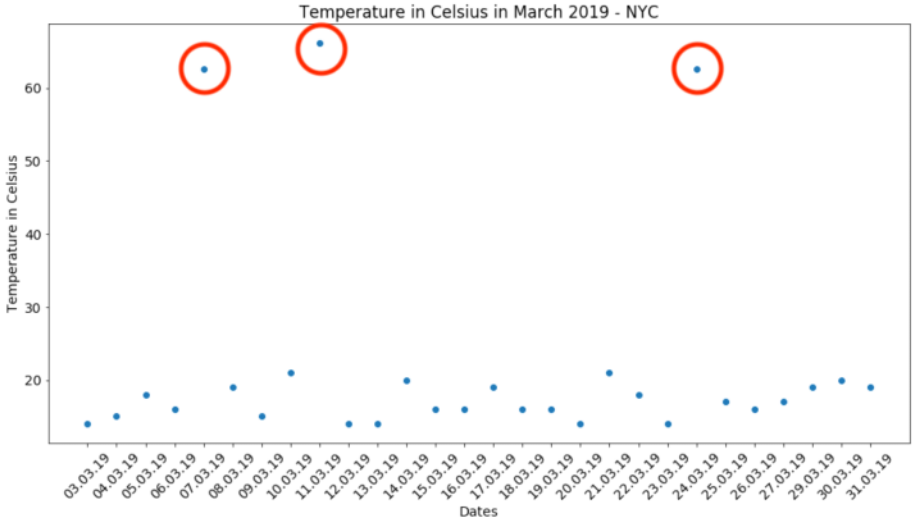
```
temperatures = pd.read_csv('temperature.csv')
temperatures.head()
```

	Date	Temperature
0	03.03.19	14.0
1	04.03.19	15.0
2	05.03.19	18.0
3	06.03.19	16.0
4	07.03.19	62.6

this value here is most likely Fahrenheit, not Celsius.

An example

```
# Import matplotlib
import matplotlib.pyplot as plt

# Create scatter plot
plt.scatter(x = 'Date', y = 'Temperature', data = temperatures)

# Create title, xlabel and ylabel
plt.title('Temperature in Celsius March 2019 - NYC')
plt.xlabel('Dates')
plt.ylabel('Temperature in Celsius')

# Show plot
plt.show()
```



They all must be fahrenheit.

Treating temperature data

$$C = (F - 32) \times \frac{5}{9}$$

```
temp_fah = temperatures.loc[temperatures['Temperature'] > 40, 'Temperature']
temp_cels = (temp_fah - 32) * (5/9)
temperatures.loc[temperatures['Temperature'] > 40, 'Temperature'] = temp_cels
```

```
# Assert conversion is correct
assert temperatures['Temperature'].max() < 40
```

statement, by making sure the maximum value of temperature is less than 40.

We chose 40 because it's a common sense maximum for Celsius temperatures in New York City.

Treating date data

```
birthdays.head()
```

	Birthday	First name	Last name
0	27/27/19	Rowan	Nunez
1	03-29-19	Brynn	Yang
2	March 3rd, 2019	Sophia	Reilly
3	24-03-19	Deacon	Prince
4	06-03-19	Griffith	Neal

??

MM-DD-YY

Month D, YYYY

Datetime formatting

`datetime` is useful for representing dates

Date	`datetime` format
25-12-2019	`%d-%m-%Y`
December 25th 2019	`%c`
12-25-2019	`%m-%d-%Y`
...	...

`pandas.to_datetime()`

- Can recognize most formats automatically
- Sometimes fails with erroneous or unrecognizable formats

Treating date data

```
# Converts to datetime - but won't work!
birthdays['Birthday'] = pd.to_datetime(birthdays['Birthday'])
```

```
ValueError: month must be in 1..12
```

```
# Will work!
birthdays['Birthday'] = pd.to_datetime(birthdays['Birthday'],
                                       # Attempt to infer format of each date
                                       infer_datetime_format=True,
                                       # Return NA for rows where conversion failed
                                       errors = 'coerce')
```

Treating date data

```
birthdays.head()
```

```
    Birthday First name Last name
0        NaT       Rowan     Nunez
1 2019-03-29       Brynn      Yang
2 2019-03-03      Sophia    Reilly
3 2019-03-24      Deacon    Prince
4 2019-06-03    Griffith      Neal
```

day day year, being set to NAT, which represents missing values in Pandas for datetime objects.

Treating date data

```
birthdays['Birthday'] = birthdays['Birthday'].dt.strftime("%d-%m-%Y")
birthdays.head()
```

```
     Birthday First name Last name
0         NaT       Rowan     Nunez
1  29-03-2019       Brynn      Yang
2  03-03-2019      Sophia    Reilly
3  24-03-2019      Deacon    Prince
4  03-06-2019    Griffith      Neal
```

Treating ambiguous date data

Is `2019-03-08` *in August or March?*

- Convert to `NA` and treat accordingly
- Infer format by understanding data source
- Infer format by understanding previous and subsequent data in DataFrame

Motivation

```
import pandas as pd

flights = pd.read_csv('flights.csv')
flights.head()
```

	flight_number	economy_class	business_class	first_class	total_passengers
0	DL140	100	60	40	200
1	BA248	130	100	70	300
2	MEA124	100	50	50	200
3	AFR939	140	70	90	300
4	TKA101	130	100	20	250

Cross field validation

*The use of **multiple** fields in a dataset to sanity check data integrity*

	flight_number	economy_class		business_class		first_class		total_passengers
0	DL140	100	+	60	+	40	=	200
1	BA248	130	+	100	+	70	=	300
2	MEA124	100	+	50	+	50	=	200
3	AFR939	140	+	70	+	90	=	300
4	TKA101	130	+	100	+	20	=	250

```
sum_classes = flights[['economy_class', 'business_class', 'first_class']].sum(axis = 1)
passenger_equ = sum_classes == flights['total_passengers']
# Find and filter out rows with inconsistent passengers
inconsistent_pass = flights[~passenger_equ]
consistent_pass = flights[passenger_equ]
```

Cross field validation

```
users.head()
```

```
   user_id  Age   Birthday
0    32985   22 1998-03-02
1    94387   27 1993-12-04
2    34236   42 1978-11-24
3    12551   31 1989-01-03
4    55212   18 2002-07-02
```

Cross field validation

```
import pandas as pd
import datetime as dt

# Convert to datetime and get today's date
users['Birthday'] = pd.to_datetime(users['Birthday'])
today = dt.date.today()
# For each row in the Birthday column, calculate year difference
age_manual = today.year - users['Birthday'].dt.year
# Find instances where ages match
age_equ = age_manual == users['Age']
# Find and filter out rows with inconsistent age
inconsistent_age = users[~age_equ]
consistent_age = users[age_equ]
```

What to do when we catch inconsistencies?

Dropping Data

Set to missing and impute

Apply rules from domain knowledge

What is missing data?

Occurs when no data value is stored for a variable in an observation

Can be represented as `NA` , `nan` , `0` , `.` ...

Technical error

Human error

Airquality example

```
import pandas as pd
airquality = pd.read_csv('airquality.csv')
print(airquality)
```

```
            Date  Temperature  CO2
987   20/04/2004         16.8  0.0
2119  07/06/2004         18.7  0.8
2451  20/06/2004        -40.0  NaN
1984  01/06/2004         19.6  1.8
8299  19/02/2005         11.2  1.2
...          ...          ...  ...
```

Airquality example

```
# Return missing values
airquality.isna()
```

```
       Date  Temperature    CO2
987   False        False  False
2119  False        False  False
2451  False        False   True
1984  False        False  False
8299  False        False  False
```

Airquality example

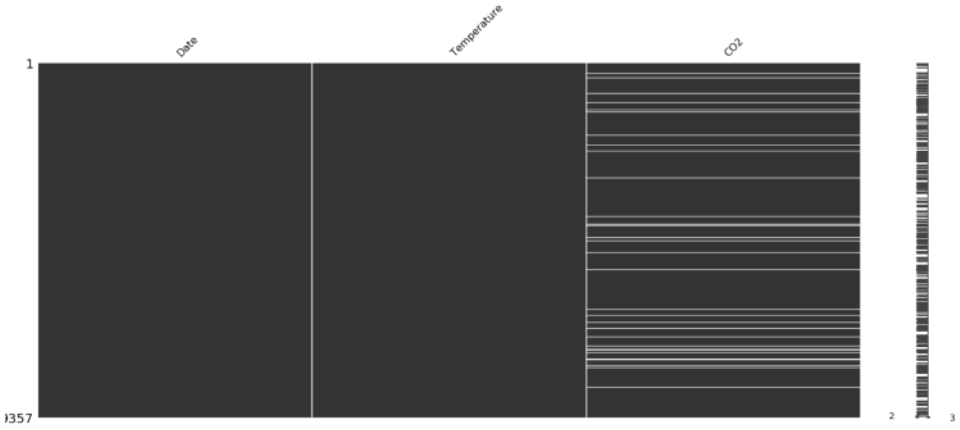
```
# Get summary of missingness
airquality.isna().sum()
```

```
Date              0
Temperature       0
CO2             366
dtype: int64
```

Missingno

Useful package for visualizing and understanding missing data

```
import missingno as msno
import matplotlib.pyplot as plt
# Visualize missingness
msno.matrix(airquality)
plt.show()
```



Airquality example

```
# Describe complete DataFramee
complete.describe()
```

	Temperature	CO2
count	8991.000000	8991.000000
mean	18.317829	1.739584
std	8.832116	1.537580
min	-1.900000	0.000000
...	...	...
max	44.600000	11.900000

```
# Describe missing DataFramee
missing.describe()
```

	Temperature	CO2
count	366.000000	0.0
mean	-39.655738	NaN
std	5.988716	NaN
min	-49.000000	NaN
...	...	...
max	-30.000000	NaN

```
sorted_airquality = airquality.sort_values(by = 'Temperature')
msno.matrix(sorted_airquality)
plt.show()
```

This is because values are sorted from smallest to largest by default.

This essentially confirms that CO2 measurements are lost for really low temperatures.

Missingness types

Missing Completely at Random

(MCAR)

No systematic relationship between missing data and other values

Data entry errors when inputting data

Missing at Random

(MAR)

*Systematic relationship between missing data and other **observed** values*

Missing ozone data for high temperatures

Missing Not at Random

(MNAR)

*Systematic relationship between missing data and **unobserved** values*

Missing temperature values for high temperatures

How to deal with missing data?

Simple approaches:

1. Drop missing data
2. Impute with statistical measures *(mean, median, mode..)*

More complex approaches:

1. Imputing using an algorithmic approach
2. Impute with machine learning models

Dealing with missing data

```
airquality.head()
```

	Date	Temperature	CO2
0	05/03/2005	8.5	2.5
1	23/08/2004	21.8	0.0
2	18/02/2005	6.3	1.0
3	08/02/2005	-31.0	NaN
4	13/03/2005	19.9	0.1

Dropping missing values

```
# Drop missing values
airquality_dropped = airquality.dropna(subset = ['CO2'])
airquality_dropped.head()
```

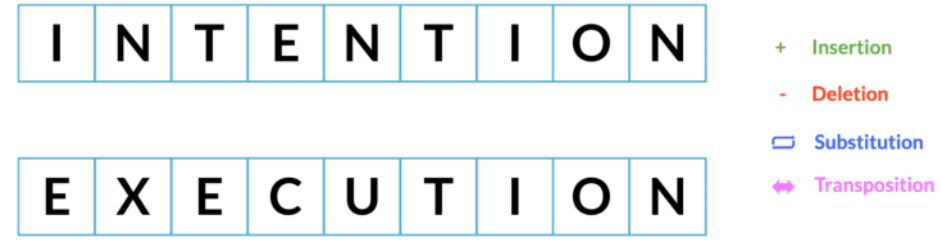
	Date	Temperature	CO2
0	05/03/2005	8.5	2.5
1	23/08/2004	21.8	0.0
2	18/02/2005	6.3	1.0
4	13/03/2005	19.9	0.1
5	02/04/2005	17.0	0.8

Replacing with statistical measures

```
co2_mean = airquality['CO2'].mean()
airquality_imputed = airquality.fillna({'CO2': co2_mean})
airquality_imputed.head()
```

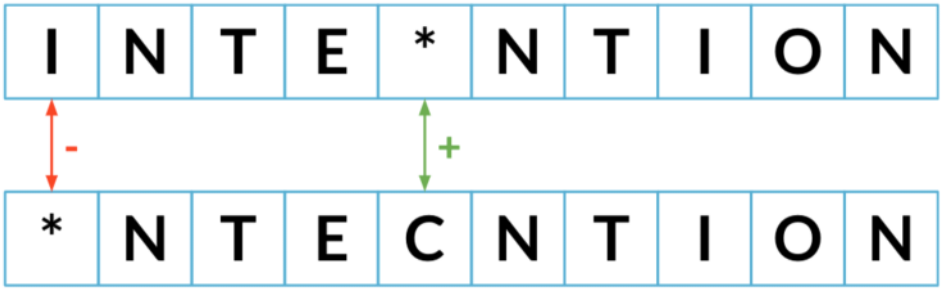
	Date	Temperature	CO2
0	05/03/2005	8.5	2.500000
1	23/08/2004	21.8	0.000000
2	18/02/2005	6.3	1.000000
3	08/02/2005	-31.0	1.739584
4	13/03/2005	19.9	0.100000

Minimum edit distance



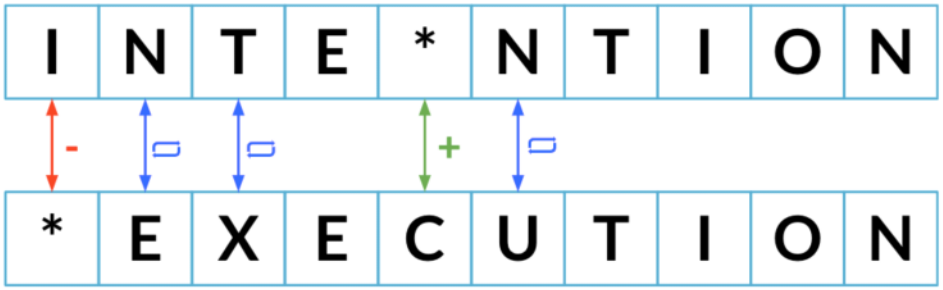
Least possible amount of steps needed to transition from one string to another

Minimum edit distance



We first start off by deleting I from intention, and adding C between E and N.

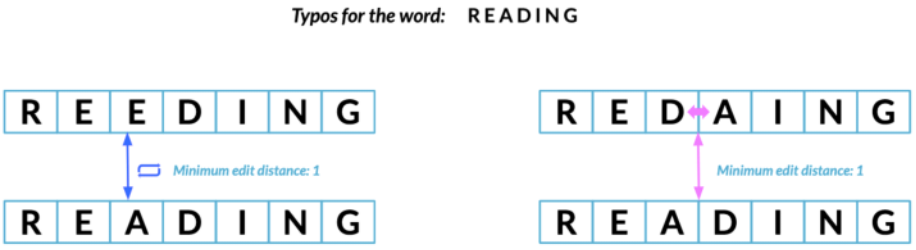
Minimum edit distance



Then we substitute the first N with E, T with X, and N with U, leading us to execution!

Minimum edit distance: 5

Minimum edit distance



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`

Simple string comparison

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

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

```
86
```

from 0 to 100 with 0 being not similar at all, 100 being an exact 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
```

Comparison with arrays

```
# 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"` ...

String similarity!

Collapsing categories with string matching

```
print(survey)
```

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

```
categories
```

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

Collapsing all of the state

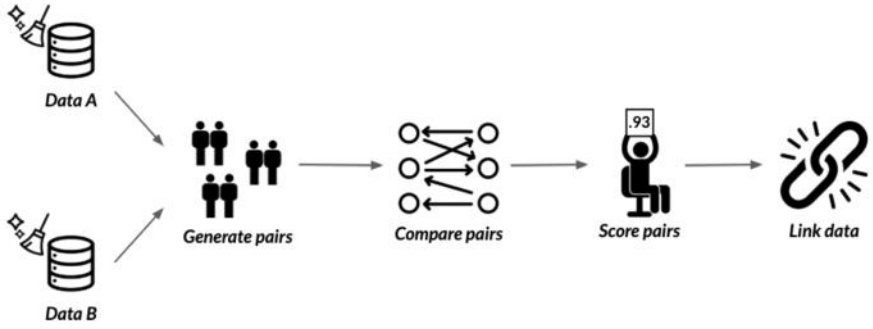
```
# For each correct category
for state in categories['state']:

  # 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
```

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	8 pm	
NBA: Bulls vs Rockets	9 pm	
NBA: Heat vs Lakers	7 pm	
NBA: Grizzlies vs Heat	10 pm	✓
NBA: Heat vs Cavaliers	9 pm	✓

Record linkage



The `recordlinkage` package

Our DataFrames

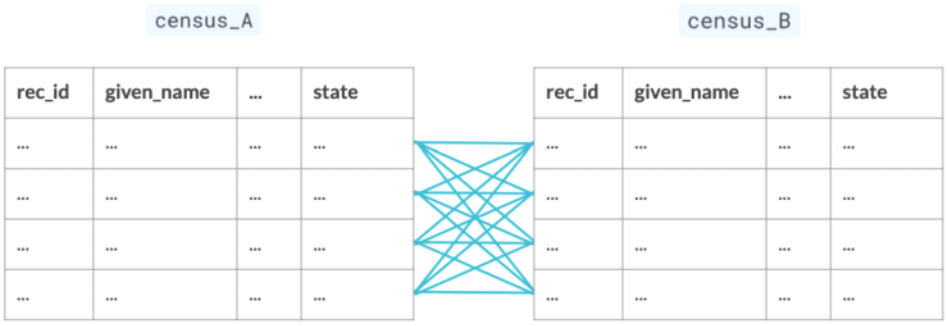
`census_A`

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

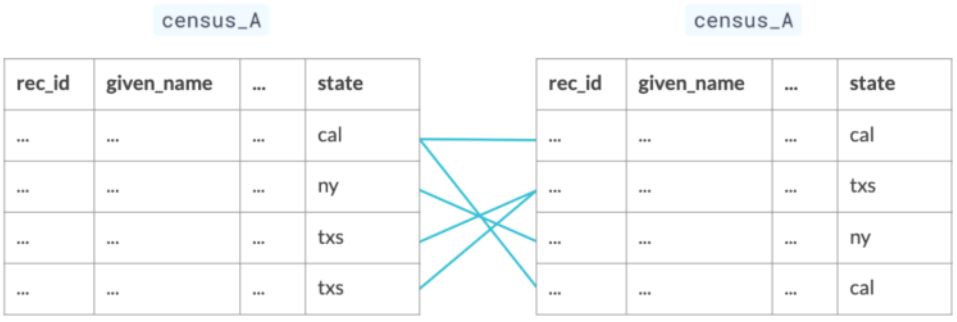
`census_B`

```
              given_name  surname date_of_birth            suburb state  address_1
rec_id
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



Blocking



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)
```

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'])
```

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)
```

Finding matching pairs

```
print(potential_matches)
```

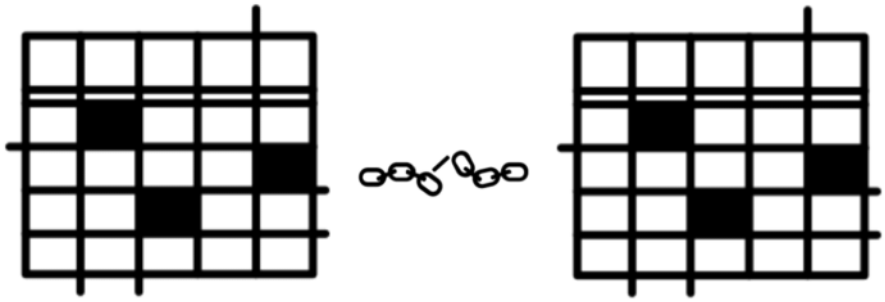
```
                          date_of_birth  state  surname  address_1
rec_id_1     rec_id_2
rec-1070-org rec-561-dup-0            0      1      0.0        0.0
             rec-2642-dup-0           0      1      0.0        0.0
             rec-608-dup-0            0      1      0.0        0.0
...
rec-1631-org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

What we're doing now



census_A census_B

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
...		...	...	...	...

link both DataFrames on, where the value is 1 for a match, and 0 otherwise.

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

Here we choose the second index column, which represents row indices of census B.

Get the indices

```
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)
```

```
# 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)
```