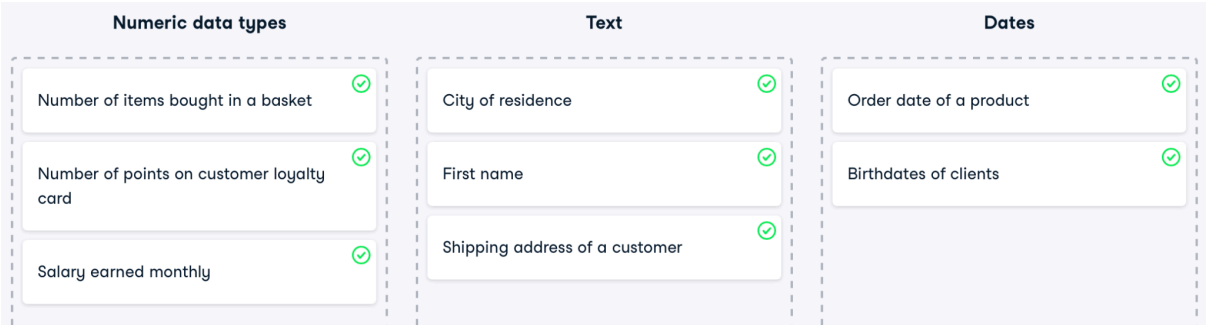
1.

Data type constraints

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

Python data type
`str`
`int`
`float`
`bool`
`datetime`
`category`

First we can check all data types by **pd.info()**, then we can change the data of the datafare to the correct one by using **pd.astype()** and taking other necessary steps.We also need to be very careful if a **column is containing text/numerical/categorical/datetime** etc



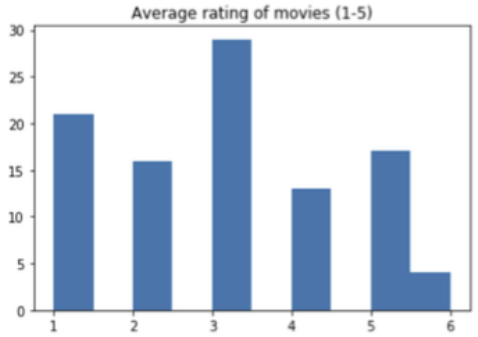
type data!! **Among themes some text/numerical /dates can also be categorical.** Check for sure by **assert df[‘column_name’].dtype == ‘int’/……**

2. Data range constraints

Suppose some numeric data or some date should be in range like a rating can be done between 0 to 5 or a date is valid from past to till present date not a future date or some categories may have 3 types 1 to 3 , 4 is invalid there etc. So we can check that by plotting a

histogram to see the data's range :

```
import matplotlib.pyplot as plt
plt.hist(movies['avg_rating'])
plt.title('Average rating of movies (1-5)')
```



Note: import datetime as dt toady = dt.date.today() dt.day/year/month()

Some methods to deal with this data's
Select : movies[movies['column_name'] > threshold]

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

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

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

Note: Here we can apply the rules for dealing the missing data or resampling technique.

3.Uniqueness constraints or Duplicates values

There can be 2 types of duplicated data, fully or partially **pd.duplicated()** values.

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)
```

We can drop the fully duplicated values by:

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

Or can also

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()
```

Note: Here we can or should apply the rules for dealing the missing data or resampling technique.

4.Membership constraints

This is kind of similar to the numerical range or date validation. **In categories there might be some predefined values out of those values other values are invalid.**

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`

We can deal with this type of data by dropping / resampling / inferring categories.

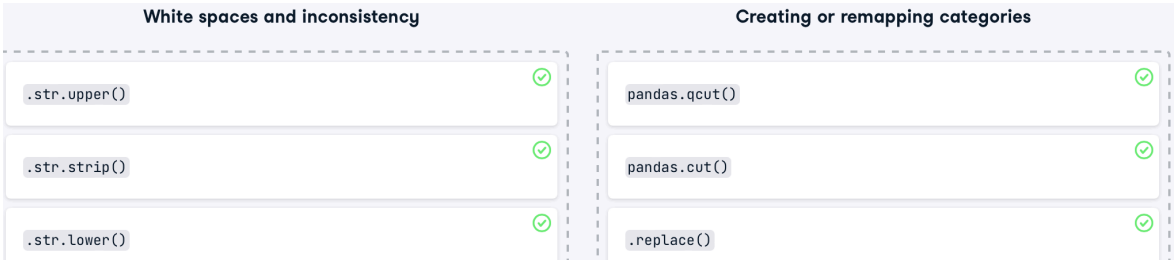
Note: We can use pd.unique() to see all existing categories in the column.

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]
```

Categorical variables

Some methods for dealing with this data's



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)

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
```

```
# 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)
```

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
```

```
# 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
```

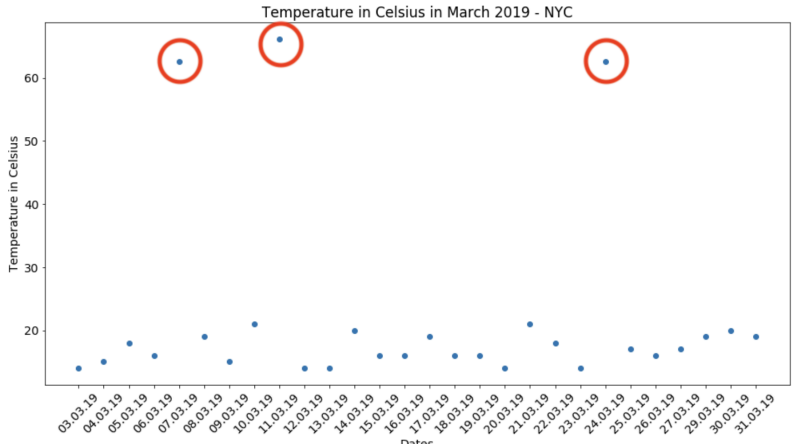
****Note: In pandas, str.replace() is used for substring replacements within string columns, whereas pd.replace() replaces values throughout the entire DataFrame, not just within strings. str.replace() works at the element level, modifying specific substrings, while pd.replace() acts on the entire DataFrame, replacing specific values across all columns.**

5.Cleaning text data

Based on the text we may need to clean the text data in various ways. Like for first name we can strip the string or can slice or can replace the unnecessary parts like DR. Mr. Ms. etc or may need to replace the -{(... etc by using regex (**pd.replace(to_replace= r'[^a-zA-Z\s]', value=“ ”, regex= True)**) or may can only take the text’s with length of some certain like up to 40 or something **new_df = pd[pd[“text”].str.len() <= 40] or** may need to modify the ph numbers or else. **Or we can use method 9. Comparing strings to clean the data also.**

6.Uniformity

"Uniformity in the context of a column refers to the consistency of data within that column. In other words, all the values in a specific column should adhere to the same type or unit. **For instance, if a column represents temperatures, it is desirable for all values to be consistently either in Celsius or Fahrenheit. Or the dates column all dates should be in one format not in different formats Or money or currency should not vary like all should be in Taka or euro or dollar or weights all should be in kg or mg etc** .The goal is to avoid a mixture of units within the same column; instead, all data points should follow a uniform standard to ensure clarity and consistency in the dataset." We can use the scatter plot to explore the types like:



We a deal with this datas such as:

```
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
```

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')
```

Or we can fixed our date format as our preference:

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

7.Cross field validation

"Cross-field validation stands out as a crucial step in data cleaning, often overlooked but essential. This process involves validating data across multiple fields using assertions, checks for converting to numeric values, examination of data types, length verification up to specific thresholds, and assessment against various constraints. Implementing cross-field validation ensures the integrity and coherence of the dataset by confirming that relationships and dependencies between different fields align with the defined criteria, contributing to a more robust and accurate dataset."

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 passenger totals
inconsistent_pass = flights[~passenger_equ]
consistent_pass = flights[passenger_equ]
```

Or can check the age is correct or not by

```
# 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]
```

8. Completeness

"Completeness is a key aspect of a dataset, indicating the extent to which it contains all necessary information. A complete dataset lacks missing values or significant gaps, providing a comprehensive and accurate representation of the underlying phenomenon. Incomplete datasets may compromise analyses and hinder the reliability of insights. Ensuring data completeness is essential for producing meaningful and trustworthy results, enabling informed decision-making based on a comprehensive understanding of the available information. Regular checks and appropriate handling of missing data contribute to maintaining the completeness of the dataset." **We can check it using the pd.isna() /pd.isna().sum() function or one more library****

Missing Completely at Random: No systematic relationship between a column's missing values and other or own values.

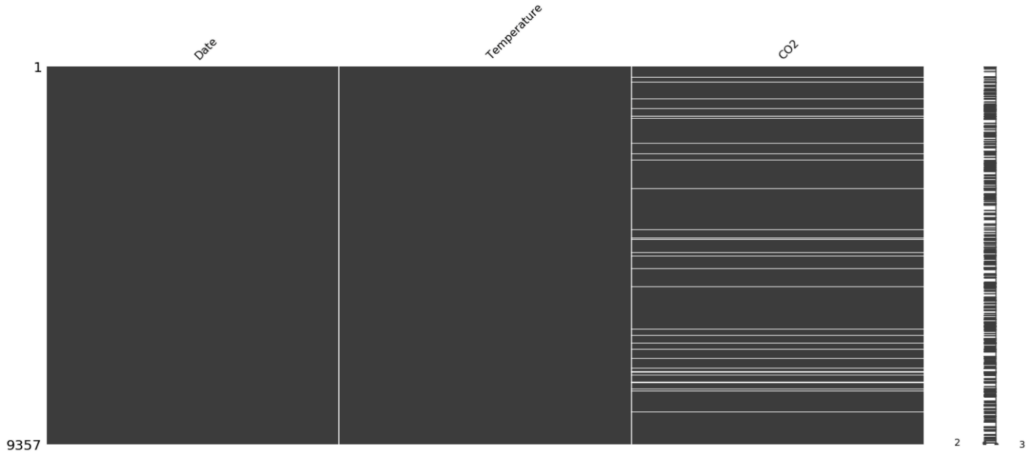
Missing at Random: There is a systematic relationship between a column's missing values and other observed values.

Missing not at Random: There is a systematic relationship between a column's missing values and unobserved values.

Missingno

Useful package for visualizing and understanding missing data

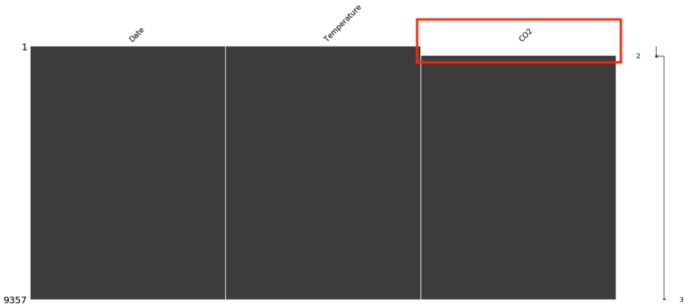
```
import missingno as msno
import matplotlib.pyplot as plt
# Visualize missingness
msno.matrix(airquality)
plt.show()
```



#Isolate the missing and complete values aside the dataframe

missing = pd[pd[“column_name”].isna()].describe()
complete= pd[~pd[“column_name”].isna()].describe()

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



Some ways to deal with this

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

Replacing with statistical measures

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

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

****Referred to another pdf for handling techniques of missing data - Link****

9.Record linkage- Comparing strings

We can use this technique to fix the closed words similarity score and use in so many cases like **inconsistent category or membership constraints or in cleaning text data etc.**

Minimum edit distance

In the video exercise, you saw how minimum edit distance is used to identify how similar two strings are. As a reminder, minimum edit distance is the minimum number of steps needed to reach from *String A* to *String B*, with the operations available being:

- **Insertion of a new character.**
- **Deletion of an existing character.**
- **Substitution of an existing character.**
- **Transposition of two existing consecutive characters.**

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` , `thefuzz` , `textdistance` ..

For example

Simple string comparison

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

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

```
86
```

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 thefuzz 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)]
```

10.Record linkage- Generating pairs

To link or not to link?

Similar to joins, record linkage is the act of linking data from different sources regarding the same entity or topic. But unlike joins, record linkage does not require exact matches between different pairs of data, and instead can find close matches using string similarity. This is why record linkage is effective when there are no common unique keys between the data sources you can rely upon when linking data sources such as a unique identifier.

Record linkage	Regular joins
Using an `address` column to join two DataFrames, with the address in each DataFrame being formatted slightly differently.	Two basketball DataFrames with a common unique identifier per game.
Two customer DataFrames containing names and address, one with a unique identifier per customer, one without.	Consolidating two DataFrames containing details on DataCamp courses, with each DataCamp course having its own unique identifier.
Merging two basketball DataFrames, with columns `team_A`, `team_B`, and `time` and differently formatted team names between each DataFrame.	

When we need to join 2 dataframes , we might need to check the common data entries or similar info rows between the two dataframes.

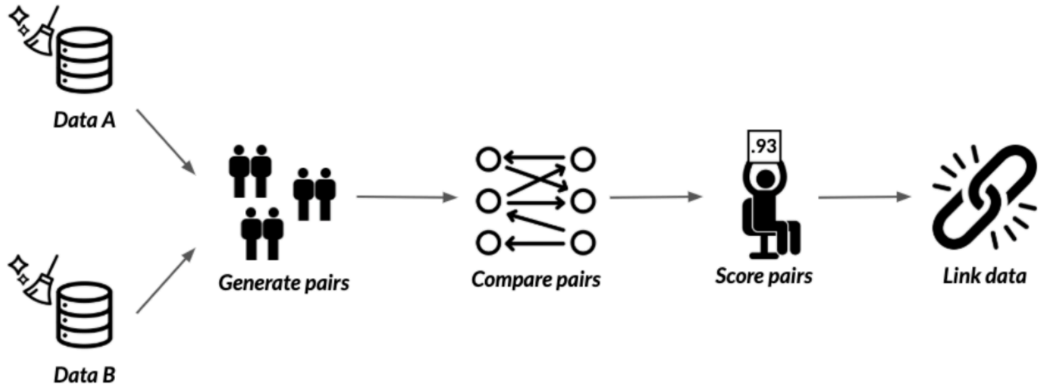
When joins won't work

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 ✓

In those case we can use the record linkage to merge the 2 dataframes having infos about some similar topic.

Record linkage



So Blocking is creating or generating paris based on some matching column in this case state column:

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)
```

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)
```

Now after that if the column matches then it will contain 1 else 0. Then to a certain threshold we want to find the pairs only we want to take. Like in this case the threshold is 2 , which means we want to take the rows only where at least 2 columns contain 1 means the same data in the same column in the

same row in both dataframes.

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.Record linkage- Linking DataFrames

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)
```