QSS20: Modern Statistical Computing

Session 04: Merging and basic regex

Goal for next few sessions

- Some course housekeeping
- Exact matching: types of joins
  - Inner joins
  - Outer joins
  - Left joins
  - Right joins
- Basic regex for two purposes:
  1. Clean join fields for exact matching/merges
  2. Clean join fields for fuzzy/probabilistic matching/merges
- Fuzzy/probabilistic matching and merges

Goal for next few sessions

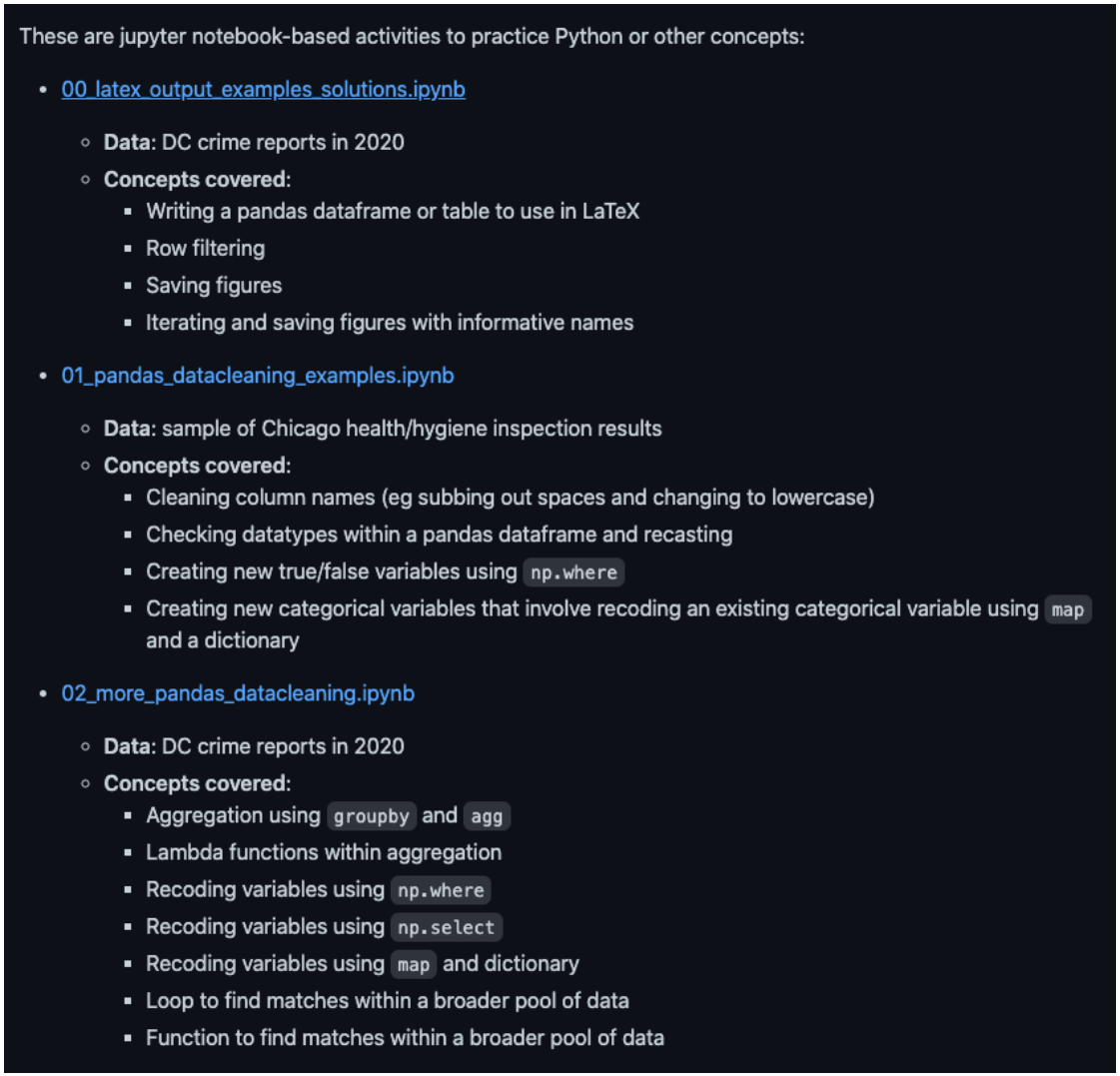
- **Some course housekeeping**
- Exact matching: types of joins
  - Inner joins
  - Outer joins
  - Left joins
  - Right joins
- Basic regex for two purposes:
  1. Clean join fields for exact matching/merges
  2. Clean join fields for fuzzy/probabilistic matching/merges
- Fuzzy/probabilistic matching and merges

PSET 1 Submission

- **Group**: tonight 1159 PM EST (will give extra time during break for u guys to meet and coordinate submission)
- **Individual**: same deadline unless using some/all of four free late days. If using all four, due at Saturday 04.25 at 1159 PM EST
- I'll give an emoji reaction on issue when i've seen it so you know it worked; i'll also comment on issue if i'm having trouble running your code

Organization of activity-based practice code

https://github.com/rebeccajohnson88/qss20_slides_activities#readme



These are jupyter notebook-based activities to practice Python or other concepts:

- 00_latex_output_examples_solutions.ipynb
  - **Data**: DC crime reports in 2020
  - **Concepts covered**:
    - Writing a pandas dataframe or table to use in LaTeX
    - Row filtering
    - Saving figures
    - Iterating and saving figures with informative names
- 01_pandas_datacleaning_examples.ipynb
  - **Data**: sample of Chicago health/hygiene inspection results
  - **Concepts covered**:
    - Cleaning column names (eg subbing out spaces and changing to lowercase)
    - Checking datatypes within a pandas dataframe and recasting
    - Creating new true/false variables using `np.where`
    - Creating new categorical variables that involve recoding an existing categorical variable using `map` and a dictionary
- 02_more_pandas_datacleaning.ipynb
  - **Data**: DC crime reports in 2020
  - **Concepts covered**:
    - Aggregation using `groupby` and `agg`
    - Lambda functions within aggregation
    - Recoding variables using `np.where`
    - Recoding variables using `np.select`
    - Recoding variables using `map` and dictionary
    - Loop to find matches within a broader pool of data
    - Function to find matches within a broader pool of data

Updated course schedule

https://rebeccajohnson88.github.io/qss20/docs/course_schedule.html

Tuesday 04-20	Intro to merging		Problem set one
Thursday 04-22	Merging: probabilistic merge and more regex		
Tuesday 04-27	Merging (continued) and PSET 1 review	Regular expressions for pattern matching	Final project step 1
Thursday 04-29	SQL via Python		
Tuesday 05-04	Text as data part one		Final project step 2
Thursday 05-06	Text as data part two		
Tuesday 05-11	TBD		
Thursday 05-13	Python: spatial data using geopandas		Problem set two
Tuesday 05-18	Python: reading data from APIs and basic web scraping		
Thursday 05-20	High-performance computing		Final project step 3
Tuesday 05-25	TBD		
Thursday 05-27	Workflow: Beamer and Tikz graphics		
			Slides for final

Steps towards final project

1. Make sure you can access this private repo and DM me if you need re-sent invite: https://github.com/rebeccajohnson88/qss20_s21_proj
2. Join `#sip_finalproject` on Slack
3. Will post details on Canvas tomorrow for `Final project Step 1`, due Tuesday 04.27 alongside the DataCamp assignment, but broadly: (1) sign up for background reading, (2) copy over LaTeX/Overleaf template I'll share, and (3) write $<$ 1 page memo outlining data used in the background reading, key takeaways, and interesting and feasible follow-up questions

Mid-term evaluation of our course

- Will circulate anonymous feedback survey later this week covering course pace, clarity, and what's going more versus less well

Goal for next few sessions

- Some course housekeeping
- **Exact matching: types of joins**
  - Inner joins
  - Outer joins
  - Left joins
  - Right joins
- Basic regex for two purposes:
  1. Clean join fields for exact matching/merges
  2. Clean join fields for fuzzy/probabilistic matching/merges
- Fuzzy/probabilistic matching and merges

Working example: have dataset on Dartmouth students and want to merge in background information about their district

- **Main or "left" dataset**

Student	**Year**	**District**	**NCES ID**
Rebecca	2021	New Trier High School	1728200
Jennifer	2022	Hanover High	3302670
Jason	2022	Homeschool	NA
⋮			

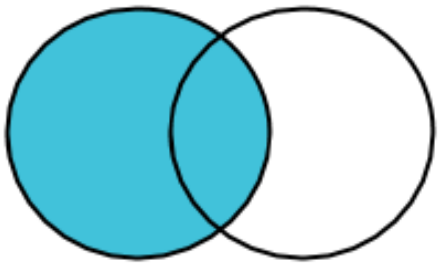
- **Auxiliary or "right" dataset**

District	**NCES ID**	**% FRPL**
New Trier HS	1728200	X%
Hanover HS	3302670	Y%
Lebanon HS	4107380	Z%
⋮		

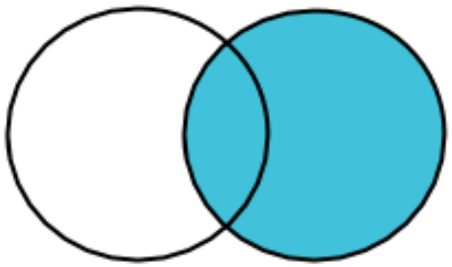
Possible join keys

- **Unique identifier:** used for "exact matching"— or a Yes/No match on that basis
  - E.g., is the NCES ID of New Trier found in the dataset of demographics?
- **Other identifiers:** can be used for either "exact match" or for "probabilistic/fuzzy matching"
  - **Probabilistic:** what's the likelihood that "New Trier district" and "New Trier HS" are the same entity?

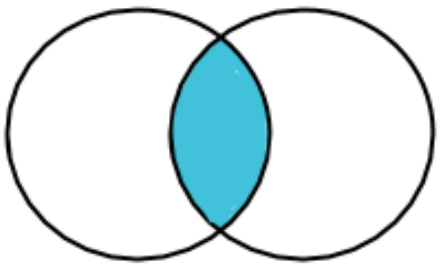
Conceptual overview of four types of joins



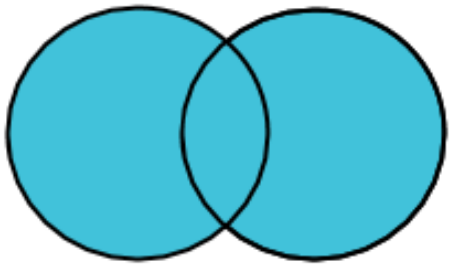
Left Join



Right Join



Inner Join



Full Outer Join

Source: Trifacta

Inner join in this context

In words: "drop all students whose districts don't appear in the demographics data; drop all districts that don't appear in the Dartmouth student data"

- **Main or "left" dataset**

Student	Year	District	NCES ID
Rebecca	2021	New Trier High School	1728200
Jennifer	2022	Hanover High	3302670
Jason	2022	Homeschool	NA
⋮			

- **Auxiliary or "right" dataset**

District	NCES ID	% FRPL
New Trier HS	1728200	X%
Hanover HS	3302670	Y%
Lebanon HS	4107380	Z%
⋮		

Outer join in this context

In words: "keep all students from the student-level data; keep all schools from the school-level data; even if there's not an overlap"

Student	**Year**	**District**	**NCES ID**	**% FRPL**
Rebecca	2021	New Trier High School	1728200	X%
Jennifer	2022	Hanover High	3302670	Y%
Jason	2022	Homeschool	NA	NA
NA	NA	NA	4107380	Z%
⋮				

Left join in this context

In words: "keep all students from the student-level data; drop any school from the school-level data that doesn't merge onto a student"

- **Main or "left" dataset**

Student	Year	District	NCES ID
Rebecca	2021	New Trier High School	1728200
Jennifer	2022	Hanover High	3302670
Jason	2022	Homeschool	NA
⋮			

- **Auxiliary or "right" dataset**

District	NCES ID	% FRPL
New Trier HS	1728200	X%
Hanover HS	3302670	Y%
Lebanon HS	4107380	Z%
⋮		

Right join in this context

In words: "drop students who don't have a school in the school-level data; keep all schools from the student-level data even those that don't merge onto any student"

- **Main or "left" dataset**

Student	Year	District	NCES ID
Rebecca	2021	New Trier High School	1728200
Jennifer	2022	Hanover High	3302670
Jason	2022	Homeschool	NA
⋮			

- **Auxiliary or "right" dataset**

District	NCES ID	% FRPL
New Trier HS	1728200	X%
Hanover HS	3302670	Y%
Lebanon HS	4107380	Z%
⋮		

How do we code these different types of joins in practice? Example with left join and join key has same colname in both

```
1
2 ## perform a left join on the student data
3 ## and schools data
4 stud_wschool = pd.merge(students,
5                         schools,
6                         how = "left",
7                         on = "NCES ID",
8                         indicator = "student_mergestatus")
```

- `how`: argument to tell it inner, left, right, outer, or cross; defaults to inner
- `on`: name of join key (in this case single key)
- `indicator`: optional arg to add a col to the resulting data (string is what to call it) that helps diagnose merge status; good for post-merge dx

Example with inner join and join key has different name

```
1
2 ## perform a left join on the student data
3 ## and schools data
4 stud_wschool = pd.merge(students,
5                         schools,
6                         how = "inner",
7                         left_on = "NCES ID",
8                         right_on = "ncesnumeric")
```

Example with left join and multiple join keys

```
1
2 ## perform a left join on the student data
3 ## and schools data
4 stud_wschool = pd.merge(students,
5                         schools,
6                         how = "left",
7                         left_on = ["NCES ID",
8                         "Dist name"],
9                         right_on = ["ncesnumeric",
10                        "distnamechar"],
11                        indicator = "student_mergestatus")
```

Non-exhaustive checklist of merge diagnostics

1. How many rows were in each data before the merge? What about after?
2. If doing a left join, did we properly retain all left-hand side rows?
3. **For strings as join keys:** if a lot of rows were lost in a merge, could that be due to spelling/punctuation variations in a character join key?
4. **For numeric identifiers as join keys:** if a lot of rows were lost in a merge, could that be due to things like the id having leading zeros and those being stripped at some stage? (e.g., one dataset identifies an entity as 002548; another as 2548)

Next up: basic regex to improve match rates for strings as join keys

- In example below, what if we didn't have the NCES ID numeric identifier? Ways to improve match rates for spelling variations (sometimes called `entity resolution`)

Student	**Year**	**District**
Rebecca	2021	New Trier High School
Jennifer	2022	Hanover High
Jason	2022	Homeschool
⋮		

District	**% FRPL**
New Trier HS	X%
Hanover HS	Y%
Lebanon HS	Z%
⋮	

Goal for next few sessions

- Some course housekeeping
- Exact matching: types of joins
  - Inner joins
  - Outer joins
  - Left joins
  - Right joins
- **Basic regex for two purposes:**
  1. Clean join fields for exact matching/merges
  2. Clean join fields for fuzzy/probabilistic matching/merges
- Fuzzy/probabilistic matching and merges

Working example

Want to clean school names and classify them into different types (elementary; middle school; high school; charter; alternative; etc...). E.g.:

Central Columbia Ms
Riley County High
Jarrell H S
Trumbull School
SAN GABRIEL ELEMENTARY
Plains El
Pond Hill School
Franklin Elementary
P.S. 119
ANDREW CARNEGIE MIDDLE

Example of variety of names that all match the pattern within `str.contains`

```
cep_optin['is_elem'] =
np.where(cep_optin.schoolname_lower.str.contains("\s+elem",
regex = True), True, False)
```

Examples of True show a lot of variation that could make merges to other data difficult...

paint branch elementary

stewart county elementary school

stove prairie elementary school

winchester avenue elementary school

oak hill elem.

lewis and clark elem.

saunemin elem school

desert springs elementary school

fifth district elementary

linden elementary school

`re` module provides more flexible alternative to pandas `str` methods

- Need to import at top: `import re`
- General structure: `re.something(r''somepattern'', some_str)`
- Challenging part is constructing the pattern that captures what you want to capture

First example

- We want to standardize the school names so that if it's an elementary school, it has the string "elemschool" after its other identifiers. E.g.:

original	**cleaned**
paint branch elementary	paint branch elemschool
stewart county elementary school	stewart county elemschool
stove prairie elementary school	⋮
winchester avenue elementary school	
oak hill elem.	
lewis and clark elem.	
saunemin elem school	
desert springs elementary school	
fifth district elementary	
linden elementary school	

Approach 1 using re.sub

Construct a pattern to match all identified variations you want to sub out. One natural inclination is to try to enumerate the variations and use | (or), e.g.:

```
## define pattern
elem_pattern = r"elementary|elem|elem\\.|elementary school"

## replace in one string
one_str_clean = re.sub(elem_pattern,
                       "elemschool", one_str)

## replace for all strings in the column schoolname_lower
all_str_clean = [re.sub(elem_pattern, "elemschool",
                        one_str)
                        for one_str in df.schoolname_lower]
```

But this leads to issues...

Issues with earlier pattern

(1) Sees `elementary` and subs it out but leaves school in; (2) leaves in . after elem. since matches `elem`

orig_name	cleaned_name
paint branch elementary	paint branch elemschool
stewart county elementary school	stewart county elemschool school
stove prairie elementary school	stove prairie elemschool school
winchester avenue elementary school	winchester avenue elemschool school
oak hill elem.	oak hill elemschool.
lewis and clark elem.	lewis and clark elemschool.
saunemin elem school	saunemin elemschool school
desert springs elementary school	desert springs elemschool school
fifth district elementary	fifth district elemschool
linden elementary school	linden elemschool school

Making the pattern more robust to variations

```
## define pattern
elem_pattern_try2 = r"(elem.*)(\s+)?(school)?"
```

Breaking down each component:

- Creating groups using ( ): these help us define groups of characters to look at together
- `elem.*`: matches elem, elem., and elementary (would maybe want to make more restrictive if we had schools named things like element that we didn't want to match)
- Multiple spaces: uses "metacharacter" to match 1+ spaces:

  `\s+`
- ? : **optional pattern**, or match if this pattern occurs but also match if it doesn't (in this case, it's useful since schools like fifth district elementary have nothing afterwards)
- (school)?: similarly, sometimes ends with school and we want to replace; other times just ends with elementary or elem

Executing with re.sub

```
## define pattern
elem_pattern_try2 = r"(elem.*)(\s+)?(school)?"

## replace for all strings in the column schoolname_lower
all_str_clean_try2 = [re.sub(elem_pattern_try2, "elemschool",
                    one_str)
                    for one_str in df.schoolname_lower]
```

Better result than our first attempt :)

orig_name	cleaned_name
paint branch elementary	paint branch elemschool
stewart county elementary school	stewart county elemschool
stove prairie elementary school	stove prairie elemschool
winchester avenue elementary school	winchester avenue elemschool
oak hill elem.	oak hill elemschool
lewis and clark elem.	lewis and clark elemschool
saunemin elem school	saunemin elemschool
desert springs elementary school	desert springs elemschool
fifth district elementary	fifth district elemschool
linden elementary school	linden elemschool

Other `re` operations

- In previous example, we:
  1. Defined a pattern: different variations of elementary school
  2. Used `re.sub(pattern, replacement, string)` to substitute something else when we match the pattern
- In other examples, we may want to:
  - Define a pattern that characterizes different subparts of a string (e.g., elementary is one part we want to extract; school is another)
  - Match that pattern
  - Extract the matches and do something

First, we can use "metacharacters" or shortcuts to match general types of patterns

https://www.geeksforgeeks.org/python-regex-metacharacters/

Common ones:

- \d or [0-9]: numbers
- [A-Z]: uppercase alpha (any; can do subsets like [ABC])
- [a-z]: lowercase alpha
- \w: alphanumeric (so numbers of letters)
- + : match one or more appearances (e.g., if we have two usernames—rebeccajohnson8; rebeccajohnson88; rebeccajohnson796–could do "[a-z]+\d+" to match all versions)
- *: match anything
- ^: match at beginning
- $: match at end
- {x, y}: match a pattern of length x to length y (e.g., if rather than capture something repeated 1 or more times, we wanted to capture numbers that look like ages and could be length 1-3, write as "\d{1,3}"

Then, three methods that do similar things

1. `re.findall(pattern, string)`
   - **What it does/returns**: scans the string from left to right and returns the matches in a list
   - **Useful for**: parsing a string and then recombining elements (e.g., `elementary` could be list element 0; `school` could be list element 1)
2. `re.search(pattern, string)`
   - **What it does/returns**: scans the entire string and returns the substring(s) regardless of where it appears in the string. If no matches found, returns `None`. If matches found, returns MatchObject
   - **Useful for:** checking *if* there's a match (since can check whether result is equal to None); same use above of getting substrings
3. `re.match(pattern, string)`
   - **What it does/returns:** Operates similarly to `re.search()` but rather than searching the entire string, only returns if found at beginning of string

Both `re.match()` and `re.search()` only return the first appearance of a pattern; if want to return all occurrences (e.g., count how many times "Trump" appears in a single tweet), can use either `re.findall()` or `re.finditer()`

Focusing on `re.findall`: executing

Example: match words before and after the charter pattern

```
## charter pattern
charter_pattern = r"(.*)\s+(charter)(\s+)?(\w+)?"

## findall
test_charter_findall = [re.findall(charter_pattern,
                                   school)
                        for school in combined_examples]
```

What this returns: a list of lists

orig_name
buffalo collegiate charter school
thomas edison charter academy
moving everest charter school
life source international charter
south valley academy charter school
neighborhood charter school of harle
brighter choice charter school-girls
children's community charter
frontier elementary school
columbus humanities, arts and...
okemos public montessori-central
pawhuska es
east valley senior high
glenpool es
number 27
south fork elementary

```
[[('buffalo collegiate', 'charter', ' ', 'school')],
 [('thomas edison', 'charter', ' ', 'academy')],
 [('moving everest', 'charter', ' ', 'school')],
 [('life source international', 'charter', '', '')],
 [('south valley academy', 'charter', ' ', 'school')],
 [('neighborhood', 'charter', ' ', 'school')],
 [('brighter choice', 'charter', ' ', 'school')],
 [("children's community", 'charter', '', '')],
 [],
 [],
 [],
 [],
 [],
 [],
 [],
 []]
```

What if we use same pattern and use `re.search` instead on the list of school names? How to execute

```
## charter pattern
charter_pattern = r"(.*)\s+(charter)(\s+)?(\w+)?"

## search
test_charter_search = [re.search(charter_pattern,
                                 school)
                       for school in combined_examples]
```

A list of a MatchObject if match exists; None otherwise

orig_name
buffalo collegiate charter school
thomas edison charter academy
moving everest charter school
life source international charter
south valley academy charter school
neighborhood charter school of harle
brighter choice charter school-girls
children's community charter
frontier elementary school
columbus humanities, arts and...
okemos public montessori-central
pawhuska es
east valley senior high
glenpool es
number 27
south fork elementary

```
[<re.Match object; span=(0, 33), match='buffalo collegiate
 <re.Match object; span=(0, 29), match='thomas edison char
 <re.Match object; span=(0, 29), match='moving everest cha
 <re.Match object; span=(0, 33), match='life source intern
 <re.Match object; span=(0, 35), match='south valley acade
 <re.Match object; span=(0, 27), match='neighborhood chart
 <re.Match object; span=(0, 30), match='brighter choice ch
 <re.Match object; span=(0, 28), match="children's communi
 None,
 None,
 None,
 None,
 None,
 None,
 None,
 None]
```

MatchObject isn't useful in its own right; how do we extract matches? .group() method on MatchObject

Example with: `thomas edison charter academy`

```
### here, we're just focusing on the 2nd match (thomas edison
    charter academy)
### and we're getting the first group from that match
thomas_match = test_charter_search[1]
thomas_match

### example where we're just getting the first group
### (name of school before charter)
thomas_firstgroup = thomas_match.group(1)
```

Prints: `thomas edison`

Generalizing to show all groups for that one match

```
### iterate over all groups and print
for i in range(0, len(thomas_match.groups())+1):
    print("Group " + str(i) + " is: ")
    print(thomas_match.group(i))

## see error if we go beyond actual number of
## groups thomas_match.group(5)
```

```
Group 0 is:
thomas edison charter academy
Group 1 is:
thomas edison
Group 2 is:
charter
Group 3 is:

Group 4 is:
academy
```

Break for group activity

Goal for next few sessions

- Some course housekeeping
- Exact matching: types of joins
  - Inner joins
  - Outer joins
  - Left joins
  - Right joins
- Basic regex for two purposes:
  1. Clean join fields for exact matching/merges
  2. Clean join fields for fuzzy/probabilistic matching/merges
- **Fuzzy/probabilistic matching and merges**

Working example: which businesses received PPP loans?

Focal dataset: sample of PPP loans for Winnetka businesses

Business name	NAICScode	City	State	Zip
CLASSIC KIDS, LLC	541921	Winnetka	IL	60093
NORTH SHORE COUNTRY DAY SCHOOL	611110	Winnetka	IL	60093

Other data:

Business name	City	State	Zip
CLASSIC KIDS	Newport Beach	CA	92660
CLASSIC KIDS UPPER WEST	Manhattan	NY	10024
CLASSIC KIDS	Winnetka	IL	60093
CLASSIC KIDS PHOTOGRAPHY	Chicago	IL	60614

What's the role of fuzzy/probabilistic matching?

- **Exact match:** would find no matches in previous example since there's no Classic Kids, LLC in the Yelp data; `pd.merge` fails us
- **Probabilistic match:**
  1. Compares a given pair of records
  2. Using 1+ fields—e.g., business name; zip code; address—what's the probability that the pair is a match?

General workflow for probabilistic matching, regardless of package

1. **Preprocess the relevant fields in the data:** none of these algorithms are magic bullets; each can have significant gains from basic string preprocessing of the relevant fields (e.g., should we remove `LLC`?; how are street addresses formulated)
2. **Decide if/what to "block" or exact match on:** when creating the candidate pairs, what's a *must have* field where if they don't match exactly, you rule out as a candidate pair?
   - **How do you decide this:** fields that are more reliably formatted (e.g., two-digit state)
   - **Main advantages:** potentially reduces false positives; reduces runtime/computational load
3. **If blocking, creating candidate pairs based on blocking variables:** if we blocked on state, for instance, this would leave the two IL businesses as candidate pairs for our focal business
4. **Decide on what fields to match "fuzzily":** these are things like name, address, etc. that might have typos/different spellings. The two components are:
   - How to define similarity: string distance functions
   - What threshold counts as similar enough
5. **Within candidate pairs, look at those fuzzy fields**
6. **Aggregate across fields to decide on "likely match" or "likely not"**

Specific workflow depends on (1) manual versus (2) package

1. In activity code, we'll (1) first do things manually and then (2) use a package
2. Packages in Python:
   - `recordlinkage`: focus of example code
   - **Others:** `fuzzy-matcher`; `sklearn` if we have a small set of "true matches" and want to build a model that predicts matches
3. Packages in R: `fast-link`; `RecordLinkage`

Whole group and then small-group activity

05_merging_session2_blank.ipynb