Pandas II

Introduction to Pandas syntax and operators

Announcement

Extra Credit:

- (1pt) Course Evaluation
- (2pts) Write a one-page review for any external references

> Freedom in choosing what you want to read

- Use plain language to summarize what you read, treat your audience as non-experts
- No more than 800 words, at most 2 figures
- Must include a short paragraph of motivation: why you pick this article/ paper
- Must include the source of reference (link, title…)
- Must include a paragraph of discussion: how this reading can benefit you/ others.

New Syntax / Concept Summary

- Operations on String series, e.g. babynames[“Name”].str.startswith()
- Creating and dropping columns.
  - Creating temporary columns is often convenient for sorting.
- Passing an index as an argument to loc.
  - Useful as an alternate way to sort a dataframe.
- Groupby: Output of .groupby(“Name”) is a DataFrameGroupBy object. Condense back into a DataFrame or Series with:
  - groupby.agg
  - groupby.size
  - groupby.filter
  - and more...
- *Pivot tables: An alternate way to group by exactly two columns.*
- *Merge: A method to join two dataframes*

Baby Names Exploration

Goal 1: Find the most popular name in California in 2018 (Filter & Sort)
Goal 2: Find all names that start with J. (startswith)
Goal 3: Sort names by length. (.str.len, sort_values)
Goal 4: Find the name whose popularity has changed the most. (ammd)
Goal 5: Count the number of female and male babies born in each year.

groupby(["Year", "Sex"]) vs. pivot_table

The pivot table more naturally represents our data.

```
babynames.groupby(["Year", "Sex"]).agg(sum).head(6)
```

Year	Sex	Count
1910	F	5950
	M	3213
1911	F	6602
	M	3381
1912	F	9803
	M	8142

```
babynames_pivot = babynames.pivot_table(
    index='Year', # the rows (turned into index)
    columns='Sex', # the column values
    values='Count', # the field(s) to processed in each group
    aggfunc=np.max, # group operation
)
babynames_pivot.head(6)
```

Year	Sex F	Sex M
1910	295	237
1911	390	214
1912	534	501
1913	584	614
1914	773	769
1915	998	1033

The MultiIndex

If we group a Series (or DataFrame) by multiple Series and then perform an aggregation operation, the resulting Series (or Dataframe) will have a `MultiIndex`.

```
babynames.groupby(["Name","Year"]).sum()
```

		Count
Name	**Year**	
Aadan	**2008**	7
	2009	6
	2014	5
Aadarsh	**2019**	6
Aaden	**2007**	20
...	**...**	...
Zyra	**2020**	15
Zyrah	**2011**	5
	2016	5
	2017	6
	2020	5

The resulting DataFrame has:

- One column: Count
- A MultiIndex, where results of aggregate function are indexed by Name first, then Year.

Regular Expressions

Using string methods and regular expressions to work with textual data

Goals For This Lecture

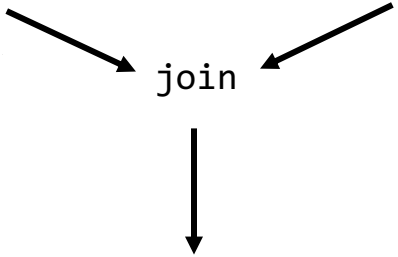
Working With Text Data

- Canonicalizing text data.
- Extracting data from text.
  - Using **split.**
  - Using **regular expressions**.

String Canonicalization

Goal 1: Joining Tables with Mismatched Labels

	County	State
0	De Witt County	IL
1	Lac qui Parle County	MN
2	Lewis and Clark County	MT
3	St John the Baptist Parish	LA



	County	Population
0	DeWitt	16798
1	Lac Qui Parle	8067
2	Lewis & Clark	55716
3	St. John the Baptist	43044

A Joining Problem

	County	State
0	De Witt County	IL
1	Lac qui Parle County	MN
2	Lewis and Clark County	MT
3	St John the Baptist Parish	LA

`join`

???

	County	Population
0	DeWitt	16798
1	Lac Qui Parle	8067
2	Lewis & Clark	55716
3	St. John the Baptist	43044

To join our tables we'll need to **canonicalize** the county names.

- Canonicalize: Convert data that has more than one possible presentation into a standard form. (Standardization)

Canonicalizing County Names

County
De Witt County
Lac qui Parle County
Lewis and Clark County
St John the Baptist Parish

County
DeWitt
Lac Qui Parle
Lewis & Clark
St. John the Baptist

```
def canonicalize_county(county_name):
    return (
        county_name
        .lower()                # lower case
        .replace(' ', '')       # remove spaces
        .replace('&', 'and')    # replace &
        .replace('.', '')       # remove dot
        .replace('county', '') # remove county
        .replace('parish', '') # remove parish
    )
```

```
def canonicalize_county(county_name):
    return (
        county_name
        .lower()                # lower case
        .replace(' ', '')       # remove spaces
        .replace('&', 'and')    # replace &
        .replace('.', '')       # remove dot
        .replace('county', '') # remove county
        .replace('parish', '') # remove parish
    )
```

County
dewitt
lacquiparle
lewisandclark
stjohnthebaptist

Canonicalization

Canonicalization:

- Replace each string with a unique representation.
- Feels very “hacky”, but messy problems often have messy solutions.

Can be done slightly better but not by much →

- Code is very brittle! Requires maintenance.

```python
def canonicalize_county(county_name):
    return (
        county_name
        .lower()                # lower case
        .replace(' ', '')       # remove spaces
        .replace('&', 'and')    # replace &
        .replace('.', '')       # remove dot
        .replace('county', '') # remove county
        .replace('parish', '') # remove parish
    )
```

Tools used:

Replacement	`str.replace(‘&’, ‘and’)`
Deletion	`str.replace(‘ ‘, ‘’)`
Transformation	`str.lower()`

Extracting From Text Using Split

Goal 2: Extracting Date Information

Suppose we want to extract times and dates from web server logs that look like the following:

169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET /stat141/Winter04/ HTTP/1.1" 200 2585 "http://anson.ucdavis.edu/courses/"

Goal 2: Extracting Date Information

Suppose we want to extract times and dates from web server logs that look like the following:

```
169.237.46.168 - - [26/Jan/2014:10:47:58 -
0800] "GET /stat141/Winter04/ HTTP/1.1" 200
2585 "http://anson.ucdavis.edu/courses/"
```

Slicing with fixed length/ index

Extracting Date Information

169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET /stat141/Winter04/ HTTP/1.1" 200 2585 "http://anson.ucdavis.edu/courses/"

One possible solution:

```
day, month, rest = line.split(' [')[1].split(']')[0].split('/')
```

```
year, hour, minute, seconds = rest.split(' ')[0].split(':')
time_zone = rest.split(' ')[1]
```

Extracting Date Information

169.237.46.168 - - [26/Jan/2014:10:47:58 -0800] "GET /stat141/Winter04/ HTTP/1.1" 200 2585 "http://anson.ucdavis.edu/courses/"

One possible solution:

```
day, month, rest = line.split(' [')[1].split(']')[0].split('/')
```

```
year, hour, minute, seconds = rest.split(' ')[0].split(':')
time_zone = rest.split(' ')[1]
```

What if webserver changes log formats, or has a bug?

⇒ *This solution breaks!! (brittle)*

Regular Expression Basics

Extracting Date Information

Earlier we saw that we can hack together code that uses split to extract info:

```
day, month, rest = line.split(' [')[1].split(']')[0].split('/')
```

```
year, hour, minute, seconds = rest.split(' ')[0].split(':')
time_zone = rest.split(' ')[1]
```

An alternate approach is to use a so-called “regular expression”:

- Implementation provided in the re library built into Python.
- We’ll spend some time today working up to expressions like shown below.

```
import re
pattern = r'\[(\d+)/(\w+)/(\d+):(\d+):(\d+):(\d+) (.+)\]'
day, month, year, hour, minute, second, time_zone = re.search(pattern, line).groups()
```

Regular Expressions

A *formal language* is a set of strings, typically described implicitly.

- Example: “The set of all strings of length < 10 that contain ‘horse’”

A *regular language* is a formal language that can be described by a *regular expression* (which we will define soon).

Example: [0-9]{3}-[0-9]{4}-[0-9]{4}

The language of cell phone number is described by this regular expression.

3 of any digit, then a dash, then 4 of any digit, then a dash, then 4 of any digit.

```
text = “My cell phone number is 123-4548-6789.";
pattern = r"[0-9]{3}-[0-9]{4}-[0-9]{4}"
re.findall(pattern, text)
```

Regular Expression Syntax

The four basic operations for regular expressions.

- Can technically do anything with just these basic four (albeit tediously).

operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA|BAAB	AA BAAB	every other string
closure (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA
parenthesis	1	A(A|B)AAB	AAAAB ABAAB	every other string
		(AB)*A	A ABABABABA	AA ABBA

Regular Expression Syntax

AB*: A then zero or more copies of B: A, AB, ABB, ABBB

(AB)*: Zero or more copies of AB: ABABABAB, ABAB, AB,

Matches the empty string!

operation	order	example	matches	does not match
concatenation	3	AABAAB	AABAAB	every other string
or	4	AA|BAAB	AA BAAB	every other string
closure (zero or more)	2	AB*A	AA ABBBBBBA	AB ABABA
parenthesis	1	A(A|B)AAB	AAAAB ABAAB	every other string
		(AB)*A	A ABABABABA	AA ABBA

https://regexr.com/

There are a ton of nice resources out there to experiment with regular expressions (e.g. sublime text, python, etc).