CS410: Principles and Techniques of Data Science

Module 4: Pandas

https://drive.google.com/drive/folders/1QIHrTPwihUDg7rxAsf4H21AJsWwkv_cj?usp=sharing

Pandas - Subsetting

Introduction

- Pandas is the standard Python package for working with dataframes
- *Dataframes* are one of the most widely used ways to represent data tables
- Data scientists work with data stored in tables

Dataframe that holds information about popular dog breeds:

breed	grooming	food_cost	kids	size
Labrador Retriever	weekly	466.0	high	medium
German Shepherd	weekly	466.0	medium	large
Beagle	daily	324.0	high	small
Golden Retriever	weekly	466.0	high	medium
Yorkshire Terrier	daily	324.0	low	small
Bulldog	weekly	466.0	medium	medium
Boxer	weekly	466.0	high	medium

Each row represents a single record - a single dog breed.

Each column represents a feature about the record - for example, the `grooming` column represents how often each dog breed needs to be groomed.

Introduction

	grooming	food_cost	kids	size
breed				
Labrador Retriever	weekly	466.0	high	medium
German Shepherd	weekly	466.0	medium	large
Beagle	daily	324.0	high	small
Golden Retriever	weekly	466.0	high	medium
Yorkshire Terrier	daily	324.0	low	small
Bulldog	weekly	466.0	medium	medium
Boxer	weekly	466.0	high	medium

- Dataframes have labels for both columns and rows.
  - Has a column labeled `grooming` and a row labeled `German Shepherd`.
- The columns and rows of a dataframe are ordered - we can refer to the Labrador Retriever row as the first row of the dataframe.
- Within a column, data have the same type. The `food_cost` column contains numbers, and the `size` column contains categories.
  But data types can be different within a row.

Subsetting

We often want to subset the specific data that they plan to use.

The New York Times article that talks about Prince Harry and Meghan’s unique choice for their new baby daughter’s name: Lilibet (Williams, 2021). The article has an interview with Pamela Redmond, an expert on baby names, who talks about interesting trends in how people name their kids. For example, she says that names that start with the letter “L” have become very popular in recent years, while names that start with the letter “J” were most popular in the 1970s and 1980s.
Are these claims reflected in data?

First, import the package as `pd`, the canonical abbreviation:

```
import pandas as pd
```

Dataset

Dataset: babynames.csv **(Source:** [Department, 2021]**)**

```
baby = pd.read_csv('babynames.csv')
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

The data in the baby table comes from the US Social Security department, which records the baby name and birth sex for birth certificate purposes.

DataFrames and Indices

Dataframe has rows and columns.

Column labels

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

Row labels (index)

By default, `pandas` assigns row labels as incrementing numbers starting from 0.

Data at the row labeled `0` and column labeled `Name` has the data `'Liam'`.

DataFrames and Indices

Dataframes can also have strings as row labels.

Column labels

	grooming	food_cost	kids	size
breed				
Labrador Retriever	weekly	466.0	high	medium
German Shepherd	weekly	466.0	medium	large
Beagle	daily	324.0	high	small
Golden Retriever	weekly	466.0	high	medium
Yorkshire Terrier	daily	324.0	low	small
Bulldog	weekly	466.0	medium	medium
Boxer	weekly	466.0	high	medium

Row labels (index)

Row labels are called the **index** of a dataframe.

Index only represents row labels, not data.

Dataframe of dog breeds has 4 columns of data, not 5, since the index doesn't count as a column.

Slicing

Slicing is an operation that creates a new dataframe by taking a subset of rows or columns out of another dataframe.

To take slices of a dataframe in `pandas`, we use the `.loc` and `.iloc`.

`.loc` - select rows and columns using their labels.

To get the data in the row labeled `1` and column labeled `Name`:

```
baby.loc[1, 'Name']
```

The first argument is the row label

The second argument is the column label

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

Slicing

To slice out multiple rows or column, you can use Python slice syntax instead of individual values:

```
baby.loc[0:3, 'Name':'Count']
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

	Name	Sex	Count
0	Liam	M	19659
1	Noah	M	18252
2	Oliver	M	14147
3	Elijah	M	13034

Slicing

To get an entire column of data, pass an empty slice as the first argument:

```
baby.loc[:, 'Count']
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

```
0          19659
1          18252
2          14147
             ...
2020719        5
2020720        5
2020721        5
Name: Count, Length: 2020722, dtype: int64
```

Slicing

To select specific columns of a dataframe, pass a list into `.loc`:

```
# And here's the dataframe with only Name and Year columns
baby.loc[:, ['Name', 'Year']]
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

	Name	Year
0	Liam	2020
1	Noah	2020
2	Oliver	2020
...	...	...
2020719	Verona	1880
2020720	Vertie	1880
2020721	Wilma	1880

2020722 rows × 2 columns

Slicing

```
# Shorthand for baby.loc[:, 'Name']
baby['Name']
```

```
0                 Liam
1                 Noah
2               Oliver
                  ...
2020719         Verona
2020720         Vertie
2020721          Wilma
Name: Name, Length: 2020722, dtype: object
```

```
# Shorthand for baby.loc[:, ['Name', 'Count']]
baby[['Name', 'Count']]
```

	Name	Count
0	Liam	19659
1	Noah	18252
2	Oliver	14147
...	...	...
2020719	Verona	5
2020720	Vertie	5
2020721	Wilma	5

2020722 rows × 2 columns

Difference between .loc and .iloc

Slicing using `.iloc` works similarly to `.loc`, except that `.iloc` uses the *positions* of rows and columns rather than labels.

dogs.csv

breed	grooming	food_cost	kids	size
Labrador Retriever	weekly	466.0	high	medium
German Shepherd	weekly	466.0	medium	large
Beagle	daily	324.0	high	small
Golden Retriever	weekly	466.0	high	medium
Yorkshire Terrier	daily	324.0	low	small
Bulldog	weekly	466.0	medium	medium
Boxer	weekly	466.0	high	medium

```
dogs.iloc[0:3, 0:2]
```

breed	grooming	food_cost
Labrador Retriever	weekly	466.0
German Shepherd	weekly	466.0
Beagle	daily	324.0

```
dogs.loc['Labrador Retriever':'Beagle', 'grooming':'food_cost']
```

breed	grooming	food_cost
Labrador Retriever	weekly	466.0
German Shepherd	weekly	466.0
Beagle	daily	324.0

Filtering Rows

filter rows - to take subsets of rows using some criteria

To find the most popular baby names in 2020: Filter rows to keep only the rows where the `Year` is 2020.

`baby`

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

`baby['Year']`

```
0          2020
1          2020
2          2020
           ...
2020719    1880
2020720    1880
2020721    1880
Name: Year, Length: 2020722, dtype: int64
```

`baby['Year'] == 2020`

```
0           True
1           True
2           True
           ...
2020719    False
2020720    False
2020721    False
Name: Year, Length: 2020722, dtype: bool
```

Filtering Rows

Passing a Series of booleans into .loc only keeps rows where the Series has a True value.

```
baby.loc[baby['Year'] == 2020, :]
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
31267	Zylynn	F	5	2020
31268	Zynique	F	5	2020
31269	Zynlee	F	5	2020

31270 rows × 4 columns

Filtering Rows

Filtering has a shorthand.

```
baby[baby['Year'] == 2020]
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
31267	Zylynn	F	5	2020
31268	Zynique	F	5	2020
31269	Zynlee	F	5	2020

31270 rows × 4 columns

Filtering Rows

To find the most common names in 2020, sort the data frame by `Count` in descending order.

```
(baby[baby['Year'] == 2020]
 .sort_values('Count', ascending=False)
 .head(7)  # take the first seven rows
)
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
13911	Emma	F	15581	2020
2	Oliver	M	14147	2020
13912	Ava	F	13084	2020
3	Elijah	M	13034	2020
13913	Charlotte	F	13003	2020

Liam, Noah, and Emma were the most popular baby names in 2020.

Example: How recently has Luna become a popular name?

The New York Times article mentions that the name “Luna” was almost nonexistent before 2000 but has since grown to become a very popular name for girls.

When exactly did Luna become popular?

Example: How recently has Luna become a popular name?

The New York Times article mentions that the name “Luna” was almost nonexistent before 2000 but has since grown to become a very popular name for girls.

When exactly did Luna become popular?

1. Filter: keep only rows with `'Luna'` in the `Name` column.
2. Filter: keep only rows with `'F'` in the `Sex` column.
3. Slice: keep the `Count` and `Year` columns.

```
luna = baby[baby['Name'] == 'Luna']  # [1]
luna = luna[luna['Sex'] == 'F']      # [2]
luna = luna[['Count', 'Year']]       # [3]
```

Pandas - Aggregating

Aggregating

Data scientists aggregate rows together to make summaries of data.

Eg: a dataset containing daily sales can be aggregated to show monthly sales instead.

Two common operations for aggregating data: *grouping* and *pivoting*

Dataset: babynames.csv **(Source:** [Department, 2021]**)**

```
baby = pd.read_csv('babynames.csv')
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

Group Aggregate

To find out the total number of babies born, sum the `Count` column:

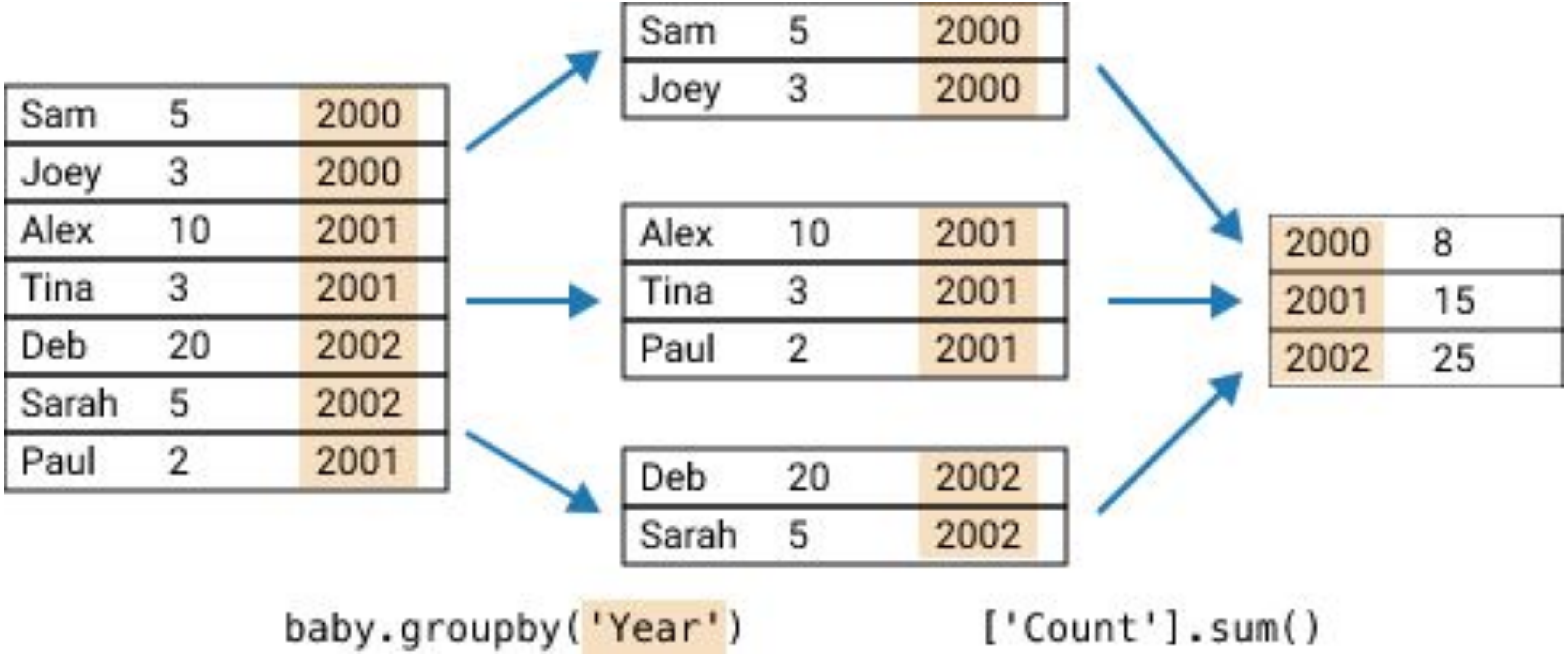
```
baby['Count'].sum()          -> 352554503
```

Group Aggregate

<u>Are U.S. births trending upwards over time?</u>

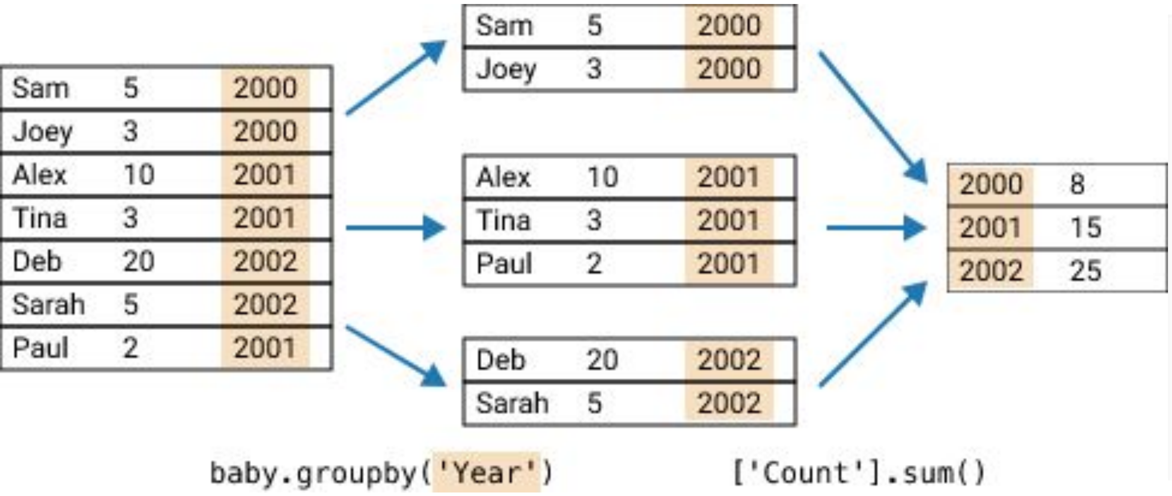
We should sum the `Count` column within each year rather than taking the sum over the entire dataset.

grouping followed by **aggregating**



Group Aggregate

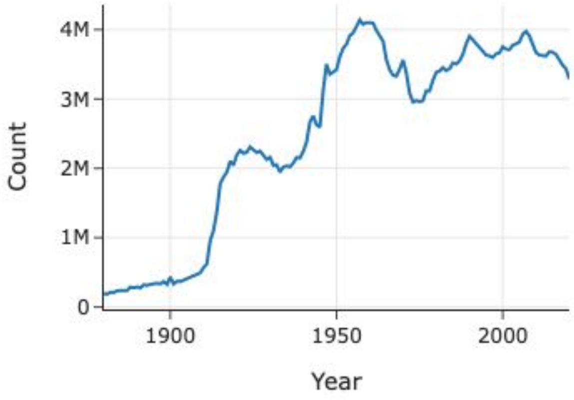
```
(baby                        # the dataframe
  .groupby('Year')           # column(s) to group
  ['Count']                  # column(s) to aggregate
  .sum()                     # how to aggregate
)
```




```
Year
1880      194419
1881      185772
1882      213385
           ...
2018     3487193
2019     3437438
2020     3287724
Name: Count, Length: 141, dtype: int64
```

Group Aggregate

```
counts_by_year = baby.groupby('Year')['Count'].sum().reset_index()
px.line(counts_by_year, x='Year', y='Count', width=350, height=250)
```



What do we see in this plot?

Suspiciously few babies born before 1920. May be the Social Security Administration was created in 1935, so its data for prior births could be less complete.

Notice the dip when World War II began in 1939, and the post-war Baby Boomer era from 1946-1964.

Grouping Multiple Columns

We pass multiple columns into `.groupby` as a list to group by multiple columns at once.

Eg: we can group by both year and sex to see how many male and female babies were born over time.

```
counts_by_year_and_sex = (baby
 .groupby(['Year', 'Sex'])
 ['Count']
 .sum()
)
```

```
Year  Sex
1880  F        83929
      M       110490
1881  F        85034
                 ...
2019  M      1785527
2020  F      1581301
      M      1706423
Name: Count, Length: 282, dtype: int64
```

Grouping Multiple Columns

The `counts_by_year_and_sex` has a multi-level index with two levels, one for each column that was grouped.

It can be a bit tricky to work with multilevel indices, so we can reset the index to go back to a dataframe with a single index.

```
counts_by_year_and_sex.reset_index()
```

		Count
Year	**Sex**	
1880	**F**	83929
	M	110490
1881	**F**	85034
...	**...**	...
2019	**M**	1785527
2020	**F**	1581301
	M	1706423

282 rows × 1 columns

	Year	Sex	Count
0	1880	F	83929
1	1880	M	110490
2	1881	F	85034
...	...	...	...
279	2019	M	1785527
280	2020	F	1581301
281	2020	M	1706423

282 rows × 3 columns

Custom Aggregation Functions

After grouping, `pandas` gives us flexible ways to aggregate the data.

```
(baby
 .groupby('Year')
 ['Count']
 .sum()
 )
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

```
Year
1880      194419
1881      185772
1882      213385
           ...
2018     3487193
2019     3437438
2020     3287724
Name: Count, Length: 141, dtype: int64
```

`pandas` also supplies other aggregation functions, like `.mean()`, `.size()`, and `.first()`. Here's the same grouping using `.max()`:

```
(baby
 .groupby('Year')
 ['Count']
 .max()
 )
```

```
Year
1880      9655
1881      8769
1882      9557
          ...
2018     19924
2019     20555
2020     19659
Name: Count, Length: 141, dtype: int64
```

Custom Aggregation Functions

Sometimes `pandas` doesn't have the exact aggregation function we want to use.

A custom aggregation function `.agg(fn)`, where `fn` is a function that we define.

To find the difference between the largest and smallest values within each group (the range of the data), we could first define a function called `data_range`, then pass that function into `.agg()`.

```
def data_range(counts):
    return counts.max() - counts.min()


(baby
 .groupby('Year')
 ['Count']
 .agg(data_range)
 )
```

```
Year
1880     9650
1881     8764
1882     9552
         ...
2018    19919
2019    20550
2020    19654
Name: Count, Length: 141, dtype: int64
```

Example: Have People Become More Creative With Baby Names?

Find whether the number of *unique* baby names per year has increased over time.

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

Example: Have People Become More Creative With Baby Names?

Find whether the number of *unique* baby names per year has increased over time.

We start by defining a `count_unique` function that counts the number of unique values in a series. Then, we pass that function into `.agg()`.

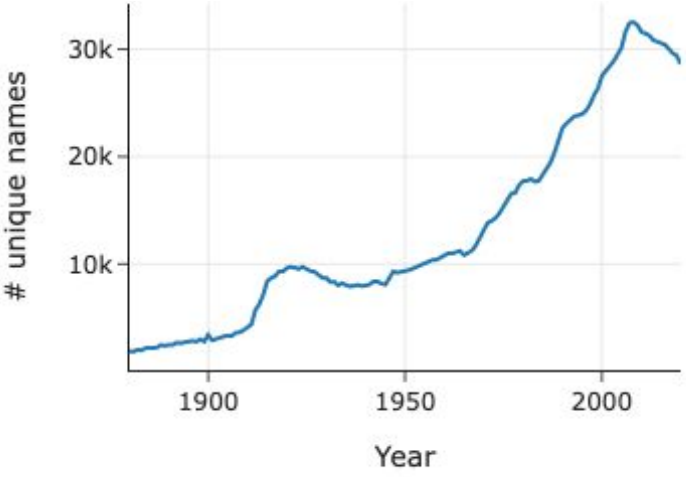
```
def count_unique(s):
    return len(s.unique())

unique_names_by_year = (baby
 .groupby('Year')
 ['Name']
 .agg(count_unique)  # aggregate using the custom count_unique function
)
```

```
Year
1880     1889
1881     1829
1882     2012
         ...
2018    29619
2019    29417
2020    28613
Name: Name, Length: 141, dtype: int64
```

Example: Have People Become More Creative With Baby Names?

```
px.line(unique_names_by_year.reset_index(),
         x='Year', y='Name',
         labels={'Name': '# unique names'},
          width=350, height=250)
```




```
Year
1880     1889
1881     1829
1882     2012
         ...
2018    29619
2019    29417
2020    28613
Name: Name, Length: 141, dtype: int64
```

We see that the number of unique names has generally increased over time.

Pivoting

A convenient way to arrange the results of a group and aggregation when grouping with two columns.

```
mf_pivot = pd.pivot_table(
    baby,
    index='Year',     # Column to turn into new index
    columns='Sex',    # Column to turn into new columns
    values='Count',   # Column to aggregate for values
    aggfunc=sum)      # Aggregation function
```

Sex	F	M
Year		
1880	83929	110490
1881	85034	100738
1882	99699	113686
...	...	...
2018	1676884	1810309
2019	1651911	1785527
2020	1581301	1706423

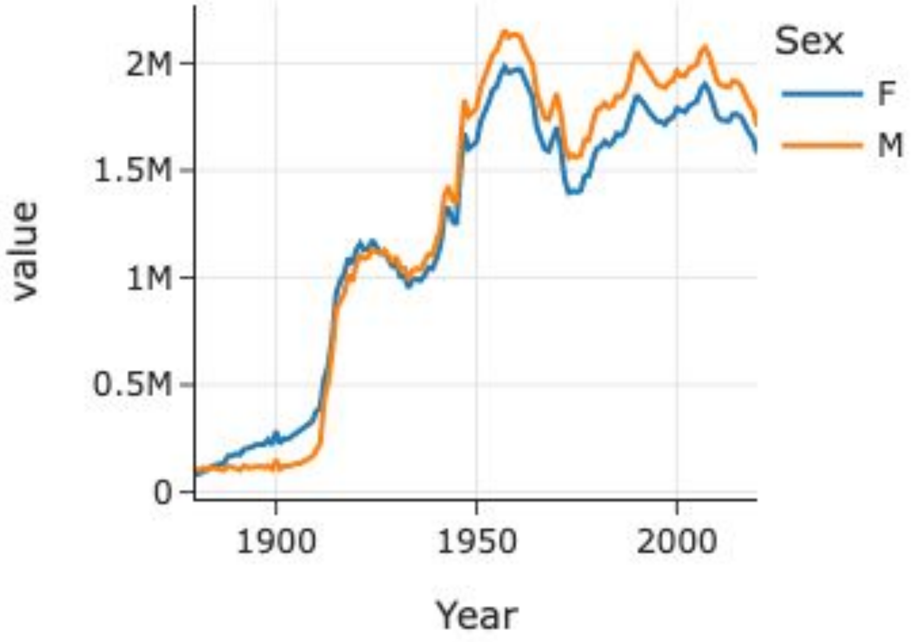
141 rows × 2 columns

Pivoting

Pivot tables are useful for quickly summarizing data using two attributes and are often seen in articles and papers.

The `px.line()` function work well with pivot tables, since the function draws one line for each column of data in the table:

```
px.line(mf_pivot, width=350, height=250)
```



Pandas - Joining

Introduction

Data scientists very frequently want to *join* two or more data frames together in order to connect data values across dataframes.

Eg: an online bookstore might have one dataframe with the books each user has ordered and a second dataframe with the genres of each book. By joining the two dataframes together, the data scientist can see what genres each user prefers.

Introduction

Baby names data: The New York Times article talks about how certain categories of names have become more or less popular over time. It mentions that mythological names like Julius and Cassius have become popular, while baby names like Susan and Debbie have become less popular.

How has the popularity of these categories changed over time?

Data

[Names and categories in the NYT article]

```
nyt = pd.read_csv('nyt_names.csv')
```

	nyt_name	category
0	Lucifer	forbidden
1	Lilith	forbidden
2	Danger	forbidden
...	...	...
20	Venus	celestial
21	Celestia	celestial
22	Skye	celestial

23 rows × 2 columns

[Baby names]

```
baby = pd.read_csv('babynames.csv')
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

Inner Joins

We'll make smaller versions of the `baby` and `nyt` tables so it's easier to see what happens when we join tables together.

```
nyt_small = nyt.iloc[[11, 12, 14]].reset_index(drop=True)
```

	nyt_name	category
0	Karen	boomer
1	Julius	mythology
2	Freya	mythology

```
names_to_keep = ['Julius', 'Karen', 'Noah']
baby_small = (baby
 .query("Year == 2020 and Name in @names_to_keep")
 .reset_index(drop=True)
)
```

	Name	Sex	Count	Year
0	Noah	M	18252	2020
1	Julius	M	960	2020
2	Karen	M	6	2020
3	Karen	F	325	2020
4	Noah	F	305	2020

Inner Joins

To join tables in `pandas`, use the `.merge()` method:

```
baby_small.merge(nyt_small,
                 left_on='Name',          # column in left table to match
                  right_on='nyt_name')    # column in right table to match
```

	Name	Sex	Count	Year
0	Noah	M	18252	2020
1	Julius	M	960	2020
2	Karen	M	6	2020
3	Karen	F	325	2020
4	Noah	F	305	2020

+

	nyt_name	category
0	Karen	boomer
1	Julius	mythology
2	Freya	mythology

=

	Name	Sex	Count	Year	nyt_name	category
0	Julius	M	960	2020	Julius	mythology
1	Karen	M	6	2020	Karen	boomer
2	Karen	F	325	2020	Karen	boomer

New table has the columns of both `baby_small` and `nyt_small` tables.

The rows with the name Noah are gone. And the remaining rows have their matching `category` from `nyt_small`.

Inner Joins

```
baby_small.merge(    nyt_small    )
```

Noah	M	18252	2020
Julius	M	960	2020
Karen	M	6	2020
Karen	F	325	2020
Noah	F	305	2020

Karen	boomer
Julius	myth
Freya	myth

(non-matching values dropped)

Result:

Julius	M	960	2020	Julius	myth
Karen	M	6	2020	Karen	boomer
Karen	F	325	2020	Karen	boomer

For inner joins (the default), rows that don't have matching values are dropped.

The Noah rows in `baby_small` don't have matches in `nyt_small`, so they are dropped.
Also, the Freya row in `nyt_small` doesn't have matches in `baby_small`, so it's dropped as well.

Only the rows with a match in both tables stay in the final result.

Left Join

Sometimes we want to keep rows without a match instead of dropping them entirely.

Other types of joins - left, right, and outer - that keep rows even when they don’t have a match.

In a *left join*, rows in the left table without a match are kept in the final result.

```
baby_small.merge(    nyt_small, how='left')
```

Noah	M	18252	2020
Julius	M	960	2020
Karen	M	6	2020
Karen	F	325	2020
Noah	F	305	2020

Karen	boomer
Julius	myth
Freya	myth

(non-matching in left table kept)

Result:

Noah	M	18252	2020	None	None
Julius	M	960	2020	Julius	myth
Karen	M	6	2020	Karen	boomer
Karen	F	325	2020	Karen	boomer
Noah	F	305	2020	None	None

Left Join

To do a left join in `pandas`, use `how='left'` in the call to `.merge()`:

```
baby_small.merge(nyt_small,
                 left_on='Name',
                 right_on='nyt_name',
                  how='left')
```

	Name	Sex	Count	Year	nyt_name	category
0	Noah	M	18252	2020	NaN	NaN
1	Julius	M	960	2020	Julius	mythology
2	Karen	M	6	2020	Karen	boomer
3	Karen	F	325	2020	Karen	boomer
4	Noah	F	305	2020	NaN	NaN

The Noah rows are kept in the final table. Since those rows didn't have a match in the `nyt_small` dataframe, the join leaves `NaN` values in the `nyt_name` and `category` columns.
Also, the Freya row in `nyt_small` is still dropped.

Right Join

A *right join* works similarly to the left join, except that non-matching rows in the right table are kept instead of the left table:

```
baby_small.merge(nyt_small,
                 left_on='Name',
                 right_on='nyt_name',
                  how='right')
```

	Name	Sex	Count	Year	nyt_name	category
0	Karen	M	6.0	2020.0	Karen	boomer
1	Karen	F	325.0	2020.0	Karen	boomer
2	Julius	M	960.0	2020.0	Julius	mythology
3	NaN	NaN	NaN	NaN	Freya	mythology

Outer Join

An *outer join* keeps rows from both tables even when they don't have a match.

```
baby_small.merge(nyt_small,
                 left_on='Name',
                 right_on='nyt_name',
                  how='outer')
```

	Name	Sex	Count	Year	nyt_name	category
0	Noah	M	18252.0	2020.0	NaN	NaN
1	Noah	F	305.0	2020.0	NaN	NaN
2	Julius	M	960.0	2020.0	Julius	mythology
3	Karen	M	6.0	2020.0	Karen	boomer
4	Karen	F	325.0	2020.0	Karen	boomer
5	NaN	NaN	NaN	NaN	Freya	mythology

Example: Popularity of NYT Name Categories

Using the full data frames

`baby`

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

`nyt`

	nyt_name	category
0	Lucifer	forbidden
1	Lilith	forbidden
2	Danger	forbidden
...	...	...
20	Venus	celestial
21	Celestia	celestial
22	Skye	celestial

23 rows × 2 columns

How the Popularity of NYT Name Categories Changed over Time

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

+

	nyt_name	category
0	Lucifer	forbidden
1	Lilith	forbidden
2	Danger	forbidden
...	...	...
20	Venus	celestial
21	Celestia	celestial
22	Skye	celestial

23 rows × 2 columns

=

	category	Year	Count
0	boomer	1880	292
1	boomer	1881	298
2	boomer	1882	326
...	...	...	...
647	mythology	2018	2944
648	mythology	2019	3320
649	mythology	2020	3489

650 rows × 3 columns

Example: Popularity of NYT Name Categories

To know how the popularity of name categories in `nyt` have changed over time.

1. Inner join `baby` with `nyt`.
2. Group the table by `category` and `Year`
3. Aggregate the counts using a sum.

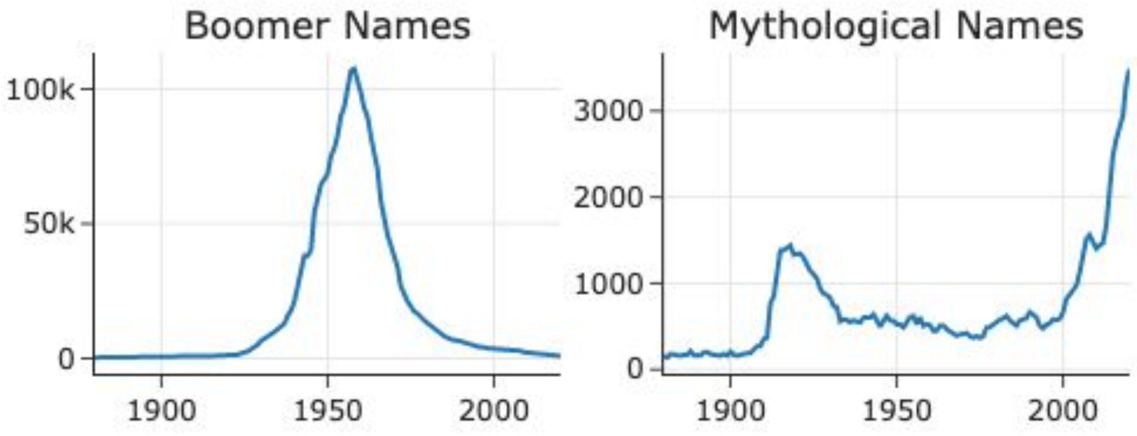
```
cate_counts = (
    baby.merge(nyt, left_on='Name', right_on='nyt_name')    # [1]
    .groupby(['category', 'Year'])                          # [2]
    ['Count']                                               # [3]
    .sum()                                                  # [3]
    .reset_index()
)
```

	category	Year	Count
0	boomer	1880	292
1	boomer	1881	298
2	boomer	1882	326
...	...	...	...
647	mythology	2018	2944
648	mythology	2019	3320
649	mythology	2020	3489

650 rows × 3 columns

Example: Popularity of NYT Name Categories

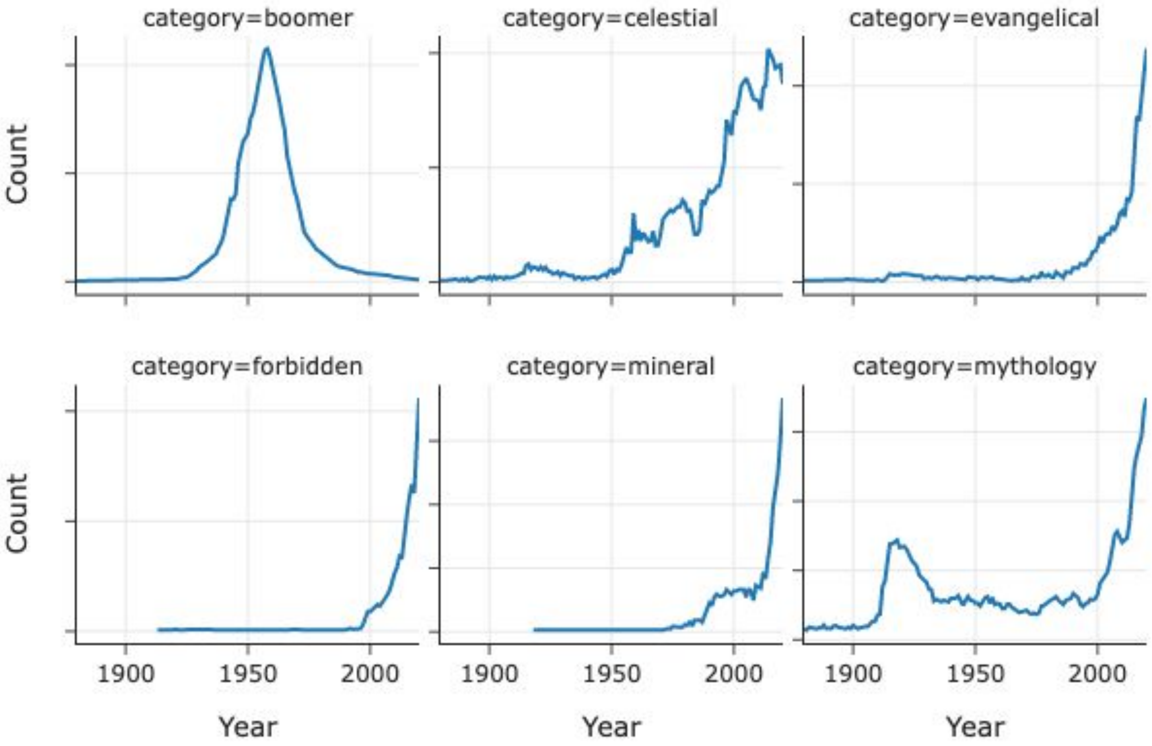
Plot of the popularity of `boomer` names and `mythology` names:



As the NYT article claims, “baby boomer” names have become less popular after 2000, while mythological names have become more popular.

Example: Popularity of NYT Name Categories

Plot the popularities of all the categories at once.



Summary

When joining dataframes together, we match rows using the `.merge()` function.

It's important to consider the type of join (inner, left, right, or outer) when joining dataframes.

Pandas - Transforming

Introduction

Data scientists transform dataframe columns when they need to change each value in a feature in the same way.

Eg: if a feature contains heights of people in feet, a data scientist might want to transform the heights to centimeters.

We'll introduce *apply*, an operation that transforms columns of data using a user-defined function.

```
baby = pd.read_csv('babynames.csv')
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

Question

The New York Times article mentions that names starting with the letter “L” and “K” became popular after 2000.

On the other hand, names starting with the letter “J” peaked in popularity in the 1970s and 1980s and have dropped off in popularity since.

How to verify these claims?

Question

The New York Times article mentions that names starting with the letter “L” and “K” became popular after 2000.

On the other hand, names starting with the letter “J” peaked in popularity in the 1970s and 1980s and have dropped off in popularity since.

We approach this problem using the following steps:

1. Transform the `Name` column into a new column that contains the first letters of each value in `Name`.
2. Group the dataframe by the first letter and year.
3. Aggregate the name counts by summing.

To complete the first step, we’ll *apply* a function to the `Name` column.

Apply

`.apply()` method takes in a function and applies it to each value in the series.

Eg: To find the lengths of each name, we apply the `len` function.

```
names = baby['Name']

names.apply(len)
```

	Name	Sex	Count	Year
0	Liam	M	19659	2020
1	Noah	M	18252	2020
2	Oliver	M	14147	2020
...	...	...	...	...
2020719	Verona	F	5	1880
2020720	Vertie	F	5	1880
2020721	Wilma	F	5	1880

2020722 rows × 4 columns

```
0          4
1          4
2          6
          ..
2020719    6
2020720    6
2020721    5
Name: Name, Length: 2020722, dtype: int64
```

Apply

To extract the first letter of each name, define a custom function and pass it into `.apply()`.

```
def first_letter(string):
    return string[0]
names.apply(first_letter)
```

```
0          L
1          N
2          O
          ..
2020719    V
2020720    V
2020721    W
Name: Name, Length: 2020722, dtype: object
```

Apply

Using `.apply()` is similar to using a `for` loop.

```
result = []
for name in names:
    result.append(first_letter(name))
```

Apply

Assign the first letters to a new column in the dataframe:

```
letters = baby.assign(Firsts=names.apply(first_letter))
                          or
baby['Firsts'] = names.apply(first_letter)
```

	Name	Sex	Count	Year	Firsts
0	Liam	M	19659	2020	L
1	Noah	M	18252	2020	N
2	Oliver	M	14147	2020	O
...	...	...	...	...	...
2020719	Verona	F	5	1880	V
2020720	Vertie	F	5	1880	V
2020721	Wilma	F	5	1880	W

2020722 rows × 5 columns

This mutates the `baby` table by adding a new column called `Firsts`.

`.assign()` doesn't mutate the `baby` table; it creates a new dataframe instead.

Mutating data frames isn't wrong but can be a common source of bugs.

Example: Popularity of “L” Names

We can use the `letters` dataframe to see the popularity of first letters over time.

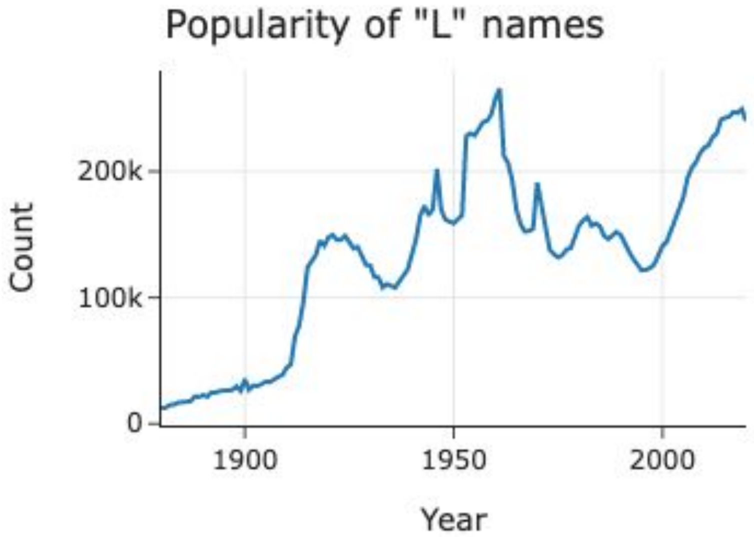
```
letter_counts = (letters
 .groupby(['Firsts', 'Year'])
 ['Count']
 .sum()
 .reset_index()
)
```

	Firsts	Year	Count
0	A	1880	16740
1	A	1881	16257
2	A	1882	18790
...	...	...	...
3638	Z	2018	55996
3639	Z	2019	55293
3640	Z	2020	54011

3641 rows × 3 columns

Example: Popularity of “L” Names

```
px.line(letter_counts.loc[letter_counts['Firsts'] == 'L'],
                 x='Year', y='Count', title='Popularity of "L" names',
                 width=350, height=250)
```

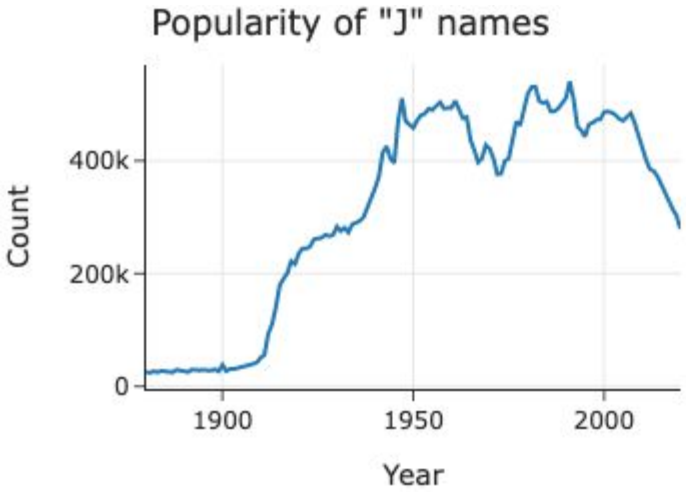


The plot shows that “L” names were popular in the 1960s, dipped in the decades after, but have indeed resurged in popularity after 2000.

Example: Popularity of “J” Names

What about “J” names?

```
px.line(letter_counts.loc[letter_counts['Firsts'] == 'J'],
                 x='Year', y='Count', title='Popularity of "J" names',
                 width=350, height=250)
```



The NYT article says that “J” names were popular in the 1970s and 80s. The plot agrees, and also shows that they have become less popular after 2000.

The Price of Apply

The power of `.apply()` is its flexibility - you can call it with any function that takes in a single data value and outputs a single data value.

Its flexibility has a price, though. Using `.apply()` can be slow, since `pandas` can't optimize arbitrary functions. For example, using `.apply()` for numeric calculations is much slower than using vectorized operations:

```
%%timeit

# Calculate the decade using vectorized operators
baby['Year'] // 10 * 10
```

```
20.5 ms ± 442 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%%timeit

def decade(yr):
    return yr // 10 * 10

# Calculate the decade using apply
baby['Year'].apply(decade)
```

```
549 ms ± 35.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

`The version using .apply() is 30 times slower!`

Summary

We looked at data frames, why they're useful, and how to work with them using `pandas` code.

Subsetting, aggregating, joining, and transforming are useful in nearly every data analysis.

We'll rely on these operations often in the rest of the course.

THANK YOU!