Data 100

Principles and Techniques of Data Science

Michael Pham

Sp24

Contents

Week 1
Introduction to Data Science

The purpose of computing is insight,
not numbers.

— R. Hamming

1.1 Lecture 1 – 01/16/24

The course website is located at: https://ds100.org/sp24/.

1.1.1 Course Overview

Why Data Science Matters

Data is used everywhere, from science to sports to medicine. Claims using data also comes up often within discussions (especially about important issues).

Furthermore, Data Science enhances critical thinking. The world is complicated, and decisions are hard. This field fundamentally facilitates decision-making by quantitatively balancing trade-offs.

In order to quantify things reliably, we have to:

- Find relevant data;
- Recognize the limitations of said data;
- Ask the right questions;
- Make reasonable assumptions;
- Conduct appropriate analysis; and
- Synthesize and explain our insights.

At each step of this process, we must apply critical thinking and consider how our decisions can affect others.

What is Data Science?

Definition 1.1 (Data Science)**.** Data Science is the application of data-centric, computational, and inferential thinking to:

- Under the world (science), and
- Solve problems (engineering).

We note that good data analysis is **not**:

- Simple applications of a statistics recipe.
- Simple application of software.

There are many tools out there for data science, but they are ultimately just tools; we are the ones doing the important thinking.

1.1.2 Course Outline

Prerequisites

The official prerequisites are:

- **DATA 8**;
- **CS 61A**, DATA C88C, or ENGIN 7; and
- EE 16A, **MATH 54**, or STAT 89A.

The bolded course names are the ones that I have already taken.

Topics (Tentative)

The tentative list of topics that will be covered in this course is:

Tentative Topics

- Pandas and NumPy
- Relational Databases and SQL
- Exploratory Data Analysis
- Regular Expressions
- Visualization
  - matplotlib
  - Seaborn
  - plotly
- Sampling
- Probability and random variables
- Model design and loss formulation
- Linear Regression
- Feature Engineering
- Regularization, Bias-Variance Tradeoff, and Cross-Validation
- Gradient Descent
- Data Science in the Physical World
- Logistic Regression
- Clustering
- PCA

Course Components

With respect to lectures and assignments, the course is structured as follows:

Course Components

Mo	Tu	We	Th	Fr
	Live Lecture		Live Lecture	
	Discussion	Discussion		
	Office Hours	Office Hours	Office Hours	Office Hours
			Homework N-1 due	Homework N released
	Lab N-1 due			Lab N released

For lectures, note that there attendance is mandatory; participation will be graded on a 0/1 basis:

- Synchronous Participation: complete at least one participation poll question during the live lecture timeslot (11:00am-12:30pm, Tuesdays and Thursdays). As long as you submit a response to at least one poll question in this timeframe, you will receive synchronous attendance credit.
- Asynchronous Participation: complete all participation poll questions from the link provided on the course website within one week of the corresponding lecture.
- In both cases, participation is graded on completion, not correctness.

Also, if we submit all participation polls over the semester, there will be a 0.5% bonus points applied to the final overall grade.

Grading

The grading scheme for this class is as follows:

Grading Scheme

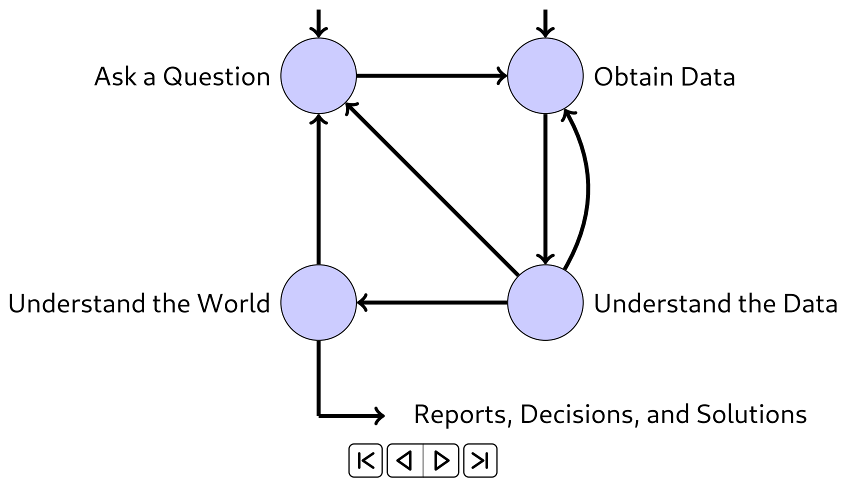
Category	
Homeworks	25%
Projects	10%
Labs	5%
Discussions	-
Lecture Participation	5%
Midterm Exam	22.5%
Final Exam	32.5%

Important:

- Midterm: Thursday, March 7, 7-9 PM PST.
- Final: Thursday, May 9, 8-11 AM PST.

1.1.3 The Data Science Lifecycle

The data science lifecycle is a high-level description of the data science workflow. Note in the diagram below that there are two distinct entry points.



The Data Science Lifecycle goes as follows:

1. **Question/Problem Formulation**
    - What do we want to know?
    - What problems are we trying to solve?
    - What hypotheses do we want to test?
    - What are our metrics for success?
2. **Data Acquisition and Cleaning**
    - What data do we have and what data do we need?
    - How will we sample more data?
    - Is our data representative of the population we want to study?
3. **Exploratory Data Analysis and Visualization**
    - How is our data organized, and what does it contain?
    - Do we already have the relevant data?
    - What are the biases, anomalies, or other issues with the data?
    - How do we transform the data to enable effective analysis?
4. **Prediction and Inference**
    - What does the data say about the world?
    - Does it answer our questions or accurately solve the problem?
    - How robust are our conclusions and can we trust the predictions?

1.2 Lecture 2 – 01/18/24

1.2.1 Tabular Data

Tabular data simply refers to data that is in a table, where each row represents one observation and each column representing some characteristic, or feature, of the observation.

In this class, we will be using the Python `pandas` library.

Figure 1.1: `pandas` logo.

With Pandas, we can accomplish a lot of things. Below are a list of some things we will be learning to use Pandas for in this course:

- Arranging data in a tabular format.
- Extract useful information filtered by specific conditions.
- Operate on data to gain new insights.
- Apply `NumPy` functions to our data.
- Perform vectorized computations to speed up our analysis.

1.2.2 DataFrames and Series

Definition 1.2 (DataFrames)**.** In the language of `pandas`, we refer to a table as a `DataFrame`, which is a collection of named columns called `Series`.

Definition 1.3 (Series)**.** A `Series` is a one-dimensional array-like object. It contains:

- A sequence of `values` of the same type.
- A sequence of data labels, called the index.

The Series Object

First, we look at some basic details about the `Series` object.

Code: Series Basics

```python
import pandas as pd

# Creating a Series object
s = pd.Series(["Welcome", "to", "data 100"])
s.index # RangeIndex(start=0, stop=3, step=1)
s.values # array(['welcome', 'to', 'data 100'], dtype='object')
```

We see that the `Series` object has `index` and `values` variables. Note that we can in fact set custom indices, as shown below:

Code: Custom Indexing

```python
# Creating Series with custom indexing
s = pd.Series([-1, 10, 2], index = ["a", "b", "c"])
s.index # Index(['a', 'b', 'c'], dtype='object')

# Changing Series' index
s.index = ["first", "second", "third"]
s.index # Index(['first', 'second', 'third'], dtype='object')
```

Finally, we can also select a single value or a set of values in a `Series` object using:

- A single label.
- A list of labels.
- A filtering condition.

Code: Selection

```python
# Creating a Series
s = pd.Series([4, -2, 0, 6], index = ["a", "b", "c", "d"])

# Selection
s["a"] # Single Label: [4]
s[["a", "c"]] # List of Labels: [4, 0]
s[s > 0] # Filtering Condition: [4, 6]
```

Note that for selection via filtering condition, we do the following:

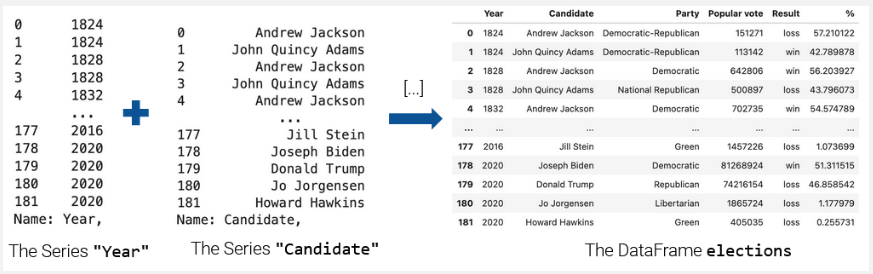
1. Apply a boolean condition to the `Series` that satisfy a particular condition.
2. Index into our `Series` using this boolean condition. `pandas` will select only the entries in the `Series` that satisfy the condition.

DataFrames

Typically, we imagine `Series` as columns within a `DataFrame` object.

One way to think of a `DataFrame` is a collection of `Series` object which shares the same index.

Figure 1.2: Relation between `DataFrame` and `Series`



```
0       1824
1       1824
2       1828
3       1828
4       1832
        ...
177     2016
178     2020
179     2020
180     2020
181     2020
Name: Year,
```

The Series "Year"

```
0           Andrew Jackson
1        John Quincy Adams
2           Andrew Jackson
3        John Quincy Adams
4           Andrew Jackson
               ...
177             Jill Stein
178           Joseph Biden
179           Donald Trump
180           Jo Jorgensen
181         Howard Hawkins
Name: Candidate,
```

The Series "Candidate"

	Year	Candidate	Party	Popular vote	Result	%
0	1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	1828	Andrew Jackson	Democratic	642806	win	56.203927
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	1832	Andrew Jackson	Democratic	702735	win	54.574789
...	...	...	...	...	...	...
177	2016	Jill Stein	Green	1457226	loss	1.073699
178	2020	Joseph Biden	Democratic	81268924	win	51.311515
179	2020	Donald Trump	Republican	74216154	loss	46.858542
180	2020	Jo Jorgensen	Libertarian	1865724	loss	1.177979
181	2020	Howard Hawkins	Green	405035	loss	0.255731

The DataFrame **elections**

The syntax for creating a `DataFrame` is `pandas.DataFrame(data, index, columns)`.

We have many approaches for creating a `DataFrame`. Below are some of the most common ones:

- From a CSV file.
- Using a list and column name(s).
- From a dictionary.
- From a `Series`.

Code: Creating `DataFrame` Objects

```python
# From a CSV File
elections = pd.read_csv("data/elections.csv")

# We can also create a DataFrame with one of the columns as the index value
elections = pd.read_csv("data/elections.csv", index_col="Year")

# From a list and column name(s)
pd.DataFrame([1, 2, 3], columns=["Numbers"]) # One column
pd.DataFrame([[1, "one"], [2, "two"]], columns = ["Number", "Description"]) # Multiple columns

# From a dictionary
pd.DataFrame({"Fruit":["Strawberry", "Orange"], "Price": [5.49, 3.99]}) # Specify columns of the
↪   DataFrame
pd.DataFrame([{"Fruit":"Strawberry", "Price":5.49}, {"Fruit":"Orange", "Price":3.99}]) # Specify the
↪   rows of the DataFrame

# From a Series
s_a = pd.Series(["a1", "a2", "a3"], index = ["r1", "r2", "r3"])
s_b = pd.Series(["b1", "b2", "b3"], index = ["r1", "r2", "r3"])

pd.DataFrame({"A-column":s_a, "B-column":s_b})
```

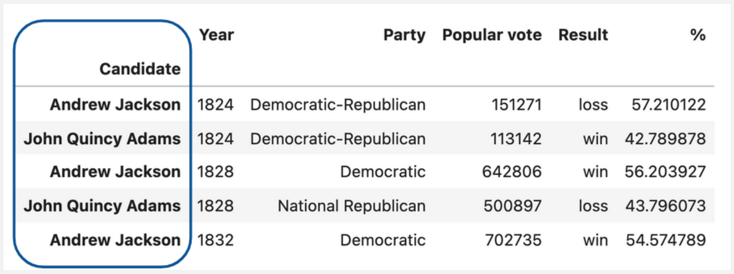
! Note that indices are not necessarily row numbers! We can set non-numeric values to be the index as well. Furthermore, they aren't necessarily unique either; we can have duplicate values in the index.

For example, suppose we ran the following code:

```python
elections = pd.read_csv("data/elections.csv", index_col = "Candidate")
```

Then, we will get the following `DataFrame`:

Figure 1.3: `DataFrame` with non-numeric, non-unique indexing.



	Year	Party	Popular vote	Result	%
Candidate					
Andrew Jackson	1824	Democratic-Republican	151271	loss	57.210122
John Quincy Adams	1824	Democratic-Republican	113142	win	42.789878
Andrew Jackson	1828	Democratic	642806	win	56.203927
John Quincy Adams	1828	National Republican	500897	loss	43.796073
Andrew Jackson	1832	Democratic	702735	win	54.574789

After creating a `DataFrame`, we can then change its index using `df.set_index(column_name)`. And if we happen to change our mind, we can simply do `df.reset_index()`.

However, while indexes are not necessarily unique, column names in `pandas` are almost always unique. Having two columns share a name is also just bad practice in general.

We can also extract basic, useful information about a `DataFrame`, such as its index, columns, and shape.

Code: Getting Basic DataFrame Information

```python
# Getting the row labels
elections.index

# Getting the column labels
elections.columns

# Getting the shape of the DataFrame, given in the form (row, col)
elections.shape
```

1.2.3 Data Extraction

When dealing with data, we want to be able to extract information that we want to work with specifically. Common ways we may want our data is as follows:

- Grab the first or last `n` rows in the `DataFrame`.
- Grab data with a certain label.
- Grab data at a certain position.

.head and .tail

The simplest scenarios: We want to extract the first or last `n` rows from the `DataFrame`.

- `df.head(n)` will return the first `n` rows of the DataFrame `df`.
- `df.tail(n)` will return the last `n` rows.

Label-based Extraction (.loc)

Something more complex is extracting data with specific columns or index labels. One way to do this is with `.loc`, which has the following format:

`df.loc[row_labels, column_labels]`

The `.loc` accessor lets us specify the labels of rows/columns we want to extract. These are the bolded values in a DataFrame.

Arguments to `.loc` can be:

- A list.
- A slice.
- A single value.

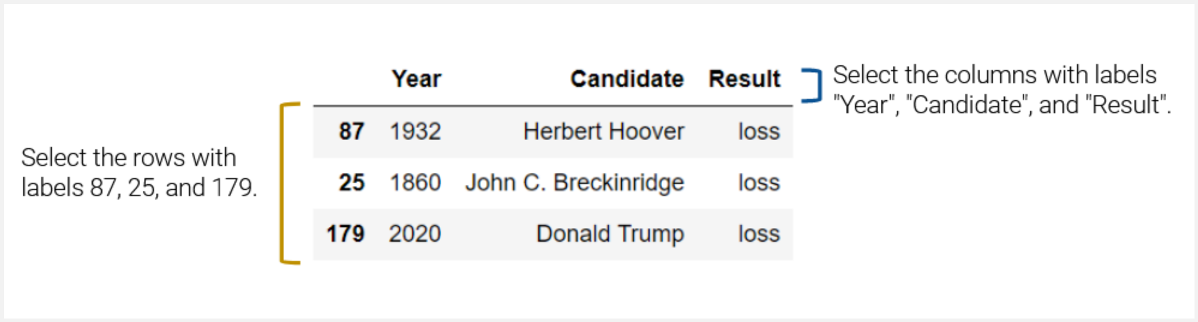
! Note that for `.loc`, the slice is, unlike normal Python syntax, **inclusive** of the right side.

Code: Extraction with `.loc`

```python
        # Using a list
        elections.loc[[87, 25, 179], ["Year", "Candidate", "Result"]]
```

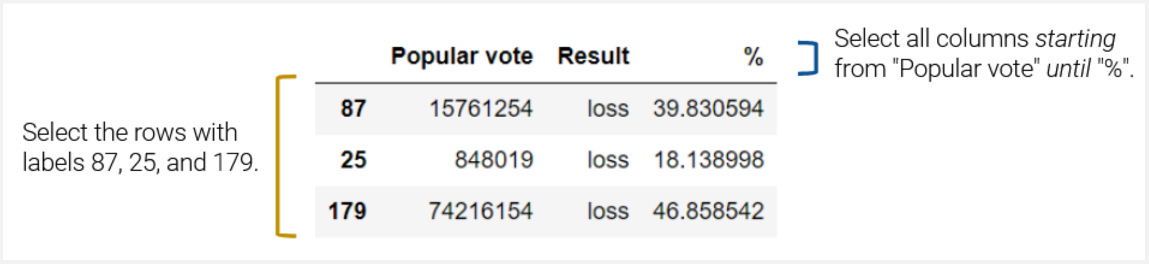
```python

# Using a slice
elections.loc[[87, 25, 179], "Popular vote":"%"]
elections.loc[:, ["Year", "Candidate", "Result"]] # Gets all rows
elections.loc[[87, 25, 179], :] # Gets all columns

# Using a single value.
elections.loc[[87, 25, 179], "Popular vote"] # Note that this returns the Popular vote series, with only the select indices
elections.loc[0, "Candidate"] # This returns the string value at index 0 from the Candidate column
```

Figure 1.4: `.loc` using a list.


Select the rows with labels 87, 25, and 179.
Select the columns with labels "Year", "Candidate", and "Result".

	Year	Candidate	Result
87	1932	Herbert Hoover	loss
25	1860	John C. Breckinridge	loss
179	2020	Donald Trump	loss

Figure 1.5: `.loc` using a slice.


Select the rows with labels 87, 25, and 179.
Select all columns *starting* from "Popular vote" *until* "%".

	Popular vote	Result	%
87	15761254	loss	39.830594
25	848019	loss	18.138998
179	74216154	loss	46.858542

Integer-based Extraction (.iloc)

A different scenario: We want to extract data according to its position.

To do this, we use `.iloc`, which allows us to specify the integers of rows and columns we wish to extract. It has the following format:

`df.iloc[row_integers, column_integers]`

Arguments to `.iloc` can be:

- A list.
- A slice.
- A single value.

! Note that while the slice was **inclusive** of the right side for `.loc`, it is **exclusive** for `.iloc`...! Very confusing!!

The syntax for `.iloc` is identical to `.loc`.

.loc versus .iloc

While `.loc` is both safer (i.e. if the order of the data gets shuffled around, our code will still work) and more readable (people actually know what columns we want), `.iloc` can still be useful!

Example 1.4 (Using `.iloc`). If we have a `DataFrame` of movie earnings sorted by earnings, we can use `.iloc` to get the median earnings for a given year by indexing into the middle.

Context-dependent Extraction ([])

Finally, we have the possibly most-complicated way of extracting data: context-dependent extraction. `[]` only takes one argument, which may be:

- A slice of row integers.
- A list of column labels.
- A single column label.

Code: Extraction with `[]`

```python
# Using a slice of row integers
elections[3:7]

# Using a list of column labels
elections[["Year", "Candidate", "Result"]]

# Using a single column label
elections["Candidate"]
```

Using `[]` is a lot more concise than either `.loc` or `.iloc`, hence it's often used more in practice than the other extraction methods.

WEEK 2
PANDAS, AND MORE PANDAS!

You're as soft as Po,
the Kung Fu Panda!

— Z. Sherwin

2.1 Lecture 3 – 01/23/24

2.1.1 Conditional Selection

We can extract rows that satisfy a given condition. Note that `.loc` and `[]` also accept boolean arrays as input; using this, only rows that correspond to `True` will be extracted.

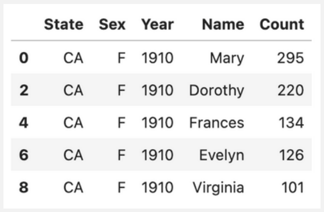
For example, consider the following code:

Code: Conditional Selection

```python
# Getting first ten rows
babynames_first_10_rows = babynames.loc[:9, :]

# Selection with []
babynames_first_10_rows[[True, False, True, False, True, False, True, False, True, False]]

# Selection with .loc
babynames_first_10_rows.loc[[True, False, True, False, True, False, True, False, True, False], :]
```

In both cases, the following DataFrame will be returned:

Figure 2.1: `DataFrame` with filtered rows.



	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
2	CA	F	1910	Dorothy	220
4	CA	F	1910	Frances	134
6	CA	F	1910	Evelyn	126
8	CA	F	1910	Virginia	101

Using this, we can then extend it to more powerful conditional selections; for example, we can select only rows that are female, *or* were born before 2000 in the DataFrame (or both!):

```
babynames[(babynames["Sex"] == "F") | (babynames["Year"] < 2000)]
```

This example above also shows us how we can use bitwise operators with our conditionals.

Bitwise Operators

If p and q are boolean arrays or `Series`, then:

Symbol	Usage	Meaning
$\sim$	$\sim p$	Negation of p
$\vert$	$p \vert q$	p or q
$\&$	$p \& q$	p and q
^	p ^ q	p xor q

While boolean array selection is useful, it can make our code overly verbose for more complicated conditions. Luckily, `pandas` offers many alternatives:

- `.isin`
- `.str.startswith`
- `.groupby.filter`

Code: Using `.isin`

```
names = ["Bella", "Alex", "Narges", "Lisa"]

# .isin
babynames[babynames["Name"].isin(names)] # Returns a Series whose entry is True if the corresponding
↪   name in babynames is found in the names list

# .str.startswith
babynames[babynames["Name"].str.startswith("N")] # Returns a Series whose entry is True if the
↪   corresponding name starts with N
```

2.1.2 Adding, Removing, and Modifying Columns

Adding and Modifying Columns

To add a column, we proceed as follows:

1. Use [] to reference the desired new column.
2. Assign this column to a Series or array of the appropriate length.

We can also modify a column's name or values. The latter's steps are similar to adding a column. For the former, we can use the `.rename()` function which takes in a dictionary mapping old column names to the new one.

Below, we show how to add, then modify a column:

Code: Adding and Modifying Columns

```python
# Create a Series of the length of each name
babyname_lengths = babynames["Name"].str.len()

# Add a column named "name_lengths" that includes the length of each name
babynames["name_lengths"] = babyname_lengths

# Modify the "name_lengths" column to be one less than its original value
babynames["name_lengths"] = babynames["name_lengths"]-1

# Rename "name_lengths" to "Length"
babynames = babynames.rename(columns={"name_lengths":"Length"})
```

Removing Columns

Removing columns a bit more tricky. First, if we want to drop a column, we can use `.drop()`.

! Note that `.drop()` assumes that we're dropping a row by default; to drop columns, we have to use `axis = "columns"`...!
Furthermore, we **must** reassign to the updated DataFrame after dropping a row/column. By default, `pandas` methods create a copy of the DataFrame, without changing the original DataFrame at all. To apply our changes, we must update our DataFrame to this new, modified copy.

Code: Removing Columns

```python
# Does NOT modify babynames DataFrame
babynames.drop("Length", axis="columns")

# Does modify babynames (through reassignment)
babynames = babynames.drop("Length", axis="columns")
```

2.1.3 Useful Utility Functions

As stated previously, one of the libraries which we will use is `NumPy`, which has a lot of useful operations such as `np.mean()` and `np.max()`.

And with the `pandas` library, we have access to a wider array of useful functions. Some which will be covered in this lecture are:

- `.shape`: returns the shape of a `DataFrame` or `Series` in the form (rows, cols).
- `.size`: returns the total number of entries in a `DataFrame` or `Series`.
- `.describe()`: returns a "description" of a `DataFrame` or `Series` that lists summary statistics of the data.
  - For `DataFrame`: count, mean, std, min/max, quartiles.
  - For `Series`: count, unique, top, freq
- `.sample()`: samples a random selection of rows from a `DataFrame`.
  - By default, it is without replacement. Use `replace=True` for replacement.
  - Naturally, can be chained with other methods and operators.
- `.value_counts()`: counts the number of occurrences of each unique value in a `Series`.

- `.unique()`: returns an array of every unique value in a `Series`.
- `.sort_values()`: sort a `DataFrame` (or `Series`).
  - `Series.sort_values()` will automatically sort all values in the `Series`.
  - `DataFrame.sort_values(column_name)` must specify the name of the column to be used for sorting.

! Note that for `.sort_values()`, rows are sorted in ascending order.
We can do `.sort_values(ascending=False)` to sort in descending order.

Custom Sorting

Suppose that we wanted to sort using some custom sorting system. To accomplish this task, there are three main ways of doing so:

1. Creating a temporary column and sort Based on the new column.
2. Sorting using the `key` argument.
3. Sorting using the `map` function.

Code: Custom Sorting

```python
# Create a Series of the length of each name
babyname_lengths = babynames["Name"].str.len()

# Approach 1: Create a temporary column
babynames["name_lengths"] = babyname_lengths # Add a column named "name_lengths" that includes the
↪   length of each name
babynames = babynames.sort_values(by="name_lengths", ascending=False)

# Approach 2: Sorting using the key Argument
babynames.sort_values("Name", key=lambda x: x.str.len(), ascending=False)

# Approach 3: Sorting using the map Function
def dr_ea_count(string):
  return string.count('dr') + string.count('ea')

babynames["dr_ea_count"] = babynames["Name"].map(dr_ea_count) # Use map to apply dr_ea_count to each
↪   name in the "Name" column
babynames = babynames.sort_values(by="dr_ea_count", ascending=False)
```

2.2 Lecture 4 – 01/25/24

2.2.1 Grouping

Our goal:

- Group together rows that fall under the same category.
  - For example, group together all rows from the same year.
- Perform an operation that aggregates across all rows in the category.

- For example, sum up the total number of babies born in that year.

Grouping is a powerful tool as we can perform large operations all at once, and summarize trends within our dataset.

Group Basics

To do this, we can use the `.groupby()` operation, which involves splitting a DataFrame up, applying a function, and then combining the result.

When we call `.groupby()` on a DataFrame, it generates `DataFrameGroupBy` objects; these are like subframes which contains all rows corresponding to the same group.

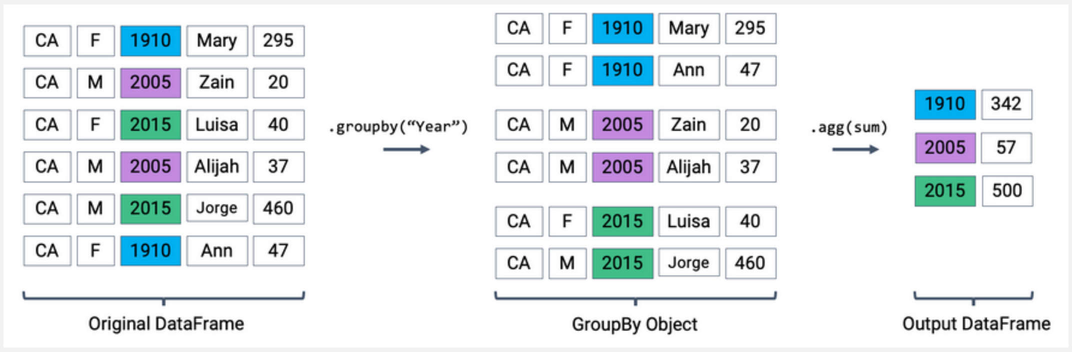
When using `.groupby()`, note that we can't directly work with the `DataFrameGroupBy` objects; instead, we use some aggregation function with `.agg()` to transform them back into a `DataFrame`.

The general syntax for the `.groupby()` function is:

`dataframe.groupby(column_name).agg(aggregation_function)`.

For example, suppose we ran `babynames[["Year", "Count"]].groupby("Year").agg(sum)`, which returns the total number of babies born each year. The result is shown in the figure below:

Figure 2.2: Demonstration of `.groupby()`.



Aggregation Functions

A number of things can go inside of the `.agg()` function:

`Python` Functions:

- `.agg(sum)`
- `.agg(max)`
- `.agg(min)`

`NumPy` Functions:

- `.agg(np.sum)`
- `.agg(np.max)`
- `.agg(np.min)`
- `.agg(np.mean)`

`pandas` Functions:

- `.agg("sum")`
- `.agg("max")`
- `.agg("min")`
- `.agg("mean")`
- `.agg("first")`
- `.agg("last")`

Now, with this in mind, we show some alternatives to finding the total number of babies born each year:

Code: Alternatives with `.groupby()`

```python
# Approach 1: Using .agg(sum)
babynames[["Year", "Count"]].groupby("Year").agg(sum)

# Approach 2: Using .sum()
babynames.groupby("Year")[["Count"]].sum()

# Approach 3: Using .sum(numeric_only=True)
babynames.groupby("Year").sum(numeric_only=True)
```

Case Study: Name Popularity

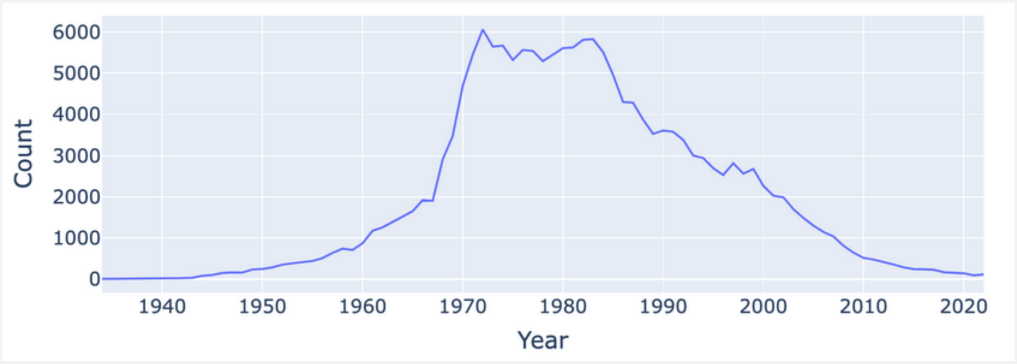
Suppose that we wanted to find the female baby name which has fallen in popularity the most in California.
For example, the code below gives us the number of Jennifer's that are born in California per year.

Code: Jennifer's Popularity

```python
f_babynames = babynames[babynames["Sex"] == "F"]
f_babynames = f_babynames.sort_values(["Year"])
jenn_counts_series = f_babynames[f_babynames["Name"] == "Jennifer"]["Count"]
```

A line plot of the data is shown below:

Figure 2.3: Line plot of Jennifer's born in California per year.



Well, looking at the count, it certainly looks like the name Jennifer has decreased in popularity!
But, how do we actually *define* "fallen in popularity"...?

- To do this, let's create a metric: "Ratio to Peak" (RTP). This is the ratio of babies born with a given name in 2022 to the maximum number of babies born with that name in any year.

Going back to the Jennifer example, in 1972, we hit peak Jennifer. 6,065 Jennifers were born. In 2022, there were only 114 Jennifer's. So, calculating the RTP, we get:

$$114/6065 = 0.018796372629843364$$

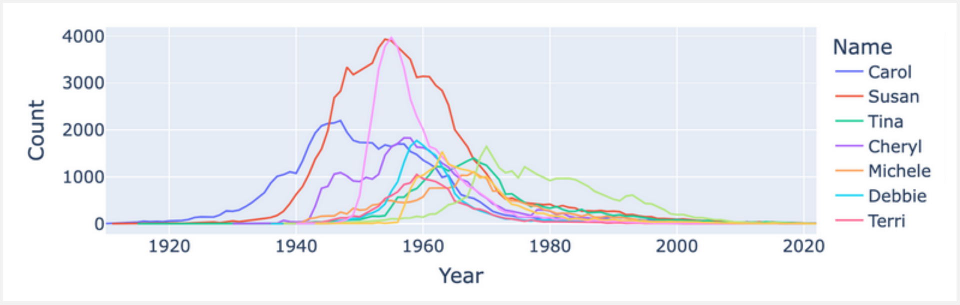
So, we can write a function `ratio_to_peak()`, then use grouping to calculate the RTP of each name:

Code: Calculating the RTP

```python
def ratio_to_peak(series):
  return series.iloc[-1] / max(series)

# This WILL error!
f_babynames.groupby("Name").agg(ratio_to_peak) # Applies ratio_to_peak() onto all columns, even
↪  Str-type ones!!

# Calculates the RTP. Returns a table with a Year and Count column
rtp_table = f_babynames.groupby("Name")[["Year","Count"]].agg(ratio_to_peak)

# Calculates the RTP. Returns a table with a Count column
rtp_table = f_babynames.groupby("Name")[["Count"]].agg(ratio_to_peak)
rtp_table = rtp_table.rename(columns = {"Count": "Count RTP"}) # Renames Count column to Count RTP

# Finds the top ten lowest RTP names
top10 = rtp_table.sort_values("Count RTP").head(10).index
# Plots the data
px.line(f_babynames[f_babynames["Name"].isin(top10)], x = "Year", y = "Count", color = "Name")
```

And after running the code above, we see that "Debra" has fallen in popularity the most out of all the names in California! For a visual representation of the ten names with the lowest RTP, we refer to the following graph:

Figure 2.4: Line graph of names which has fallen in popularity the most in California.



All in all, remember that the result of a groupby operation applied to a `DataFrame` is a `DataFrameGroupBy` object.

And once we're given a `DataFrameGroupBy` object, we can use various functions to generate `DataFrames` (or `Series`). Note that `.agg()` is only one choice! Some other options are:

- `df.groupby(col).mean()`
- `df.groupby(col).sum()`
- `df.groupby(col).min()`
- `df.groupby(col).max()`
- `df.groupby(col).first()`
- `df.groupby(col).last()`
- `df.groupby(col).size()`
- `df.groupby(col).count()`
- `df.groupby(col).filter()`

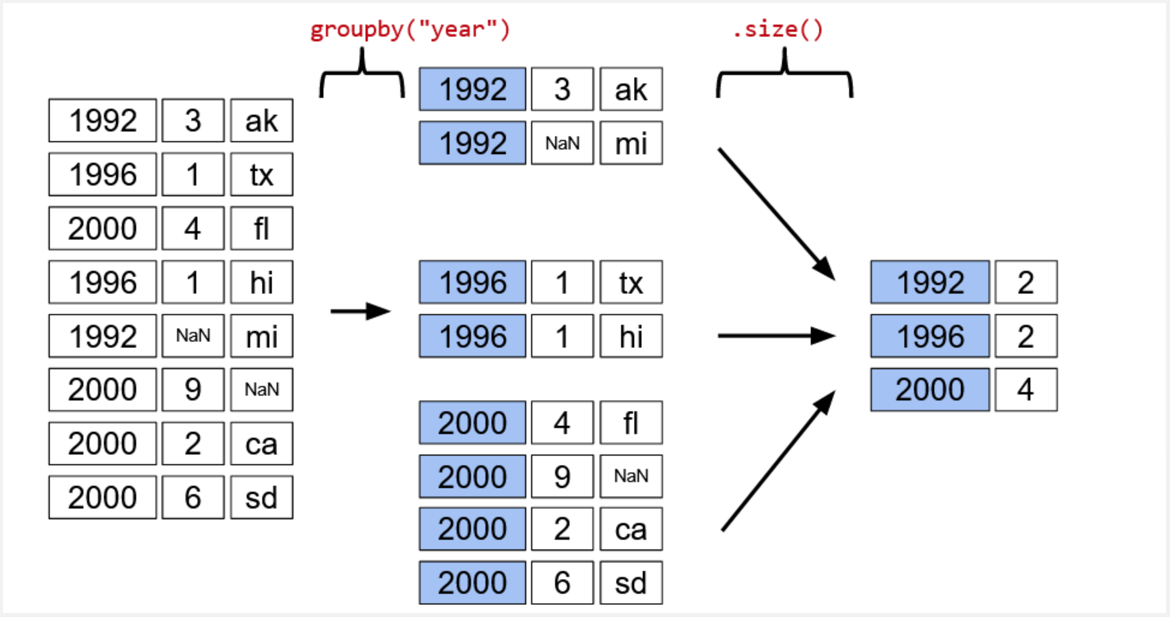
See `https://pandas.pydata.org/docs/reference/groupby.html` for a list of `DataFrameGroupBy` methods that we can use!

groupby.size() versus groupby.count()

One of the more subtle differences between some functions is that of `.groupby().size()` and `.groupby().count()`.

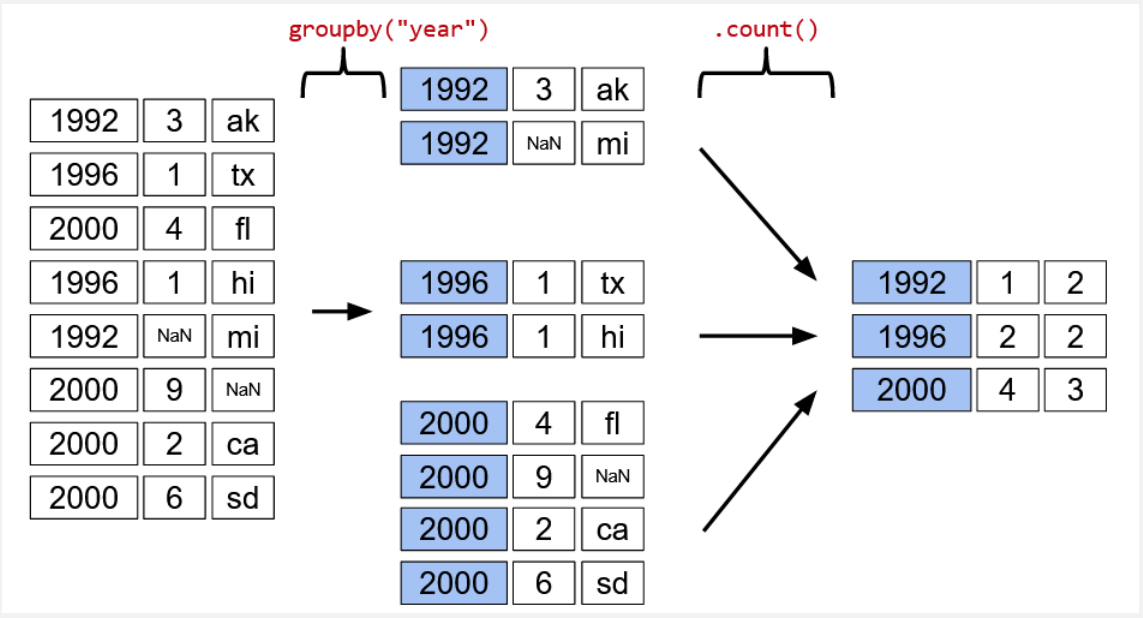
`.size()` returns a `Series` object counting the number of rows in each group. This is similar to `.value_counts()`, though it does not sort the index based on the frequency of entries.

Figure 2.5: `groupby.size()` demo.



On the other hand, with `.count()`, it returns a `DataFrame` with the counts of non-missing values in each column.

Figure 2.6: `groupby.count()` demo.



Filtering by Groups

Another common use for groups is to filter data.

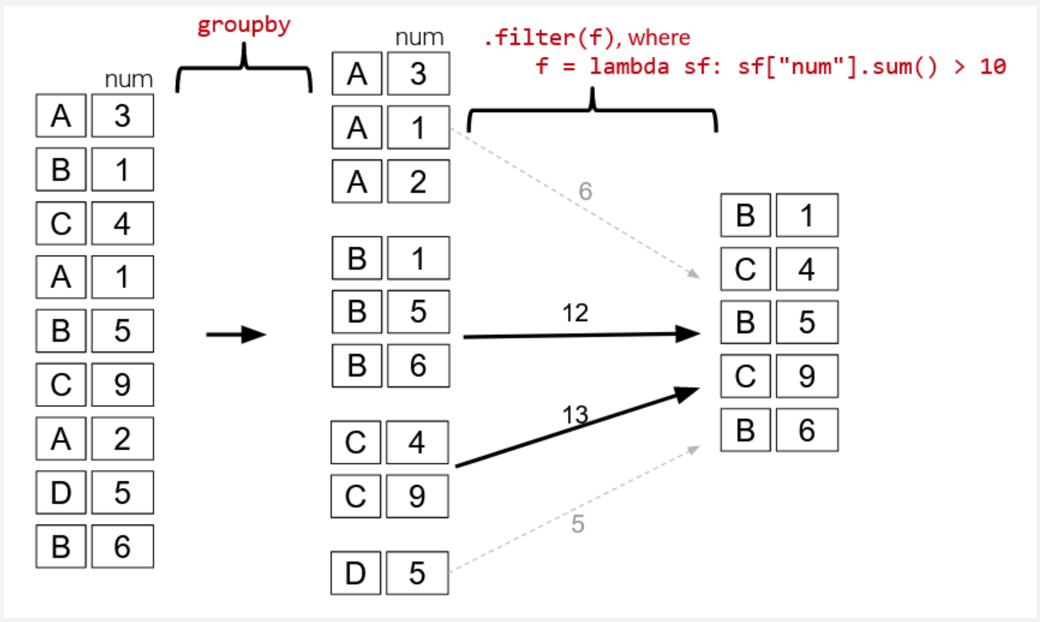
- `groupby.filter()` takes an argument `func`.
- `func` is a function that:
  - Takes a DataFrame as input.
  - Returns either `True` or `False`.
- `filter` applies `func` to each group/sub-`DataFrame`:
  - If `func` returns `True` for a group, then all rows belonging to the group are preserved.
  - If `func` returns `False` for a group, then all rows belonging to that group are filtered out.

!

Some caveats to filtering:

- Filtering is done per group, not per row. This is different from boolean filtering!
- Unlike `agg()`, the column we grouped on does NOT become the index!

Figure 2.7: Filtering by groups demo.


groupby
num
.filter(f), where
f = lambda sf: sf["num"].sum() > 10
6
12
13
5