LECTURE 3

Pandas, Part I

Introduction to `pandas` syntax, operators, and functions

Data Science| Fall 23@ Knowledge Stream

Sana Jabbar

Recall

Data Retrieval from indices

Row and Col Selection

- A list
- A slice
- A single value

Using [ ]

```
weird = pd.DataFrame({1:["topdog","botdog"], "1":["topcat","botcat"]})
weird
```

Try to predict the output of the following:

1. weird[1].
2. weird[[“1”]]
3. weird[1:]

Recall

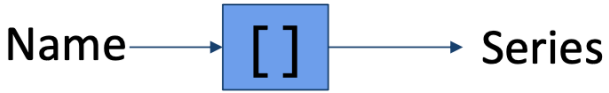
Row and Col Selection

- A list
- A slice
- A single value

Data Retrieval from indices

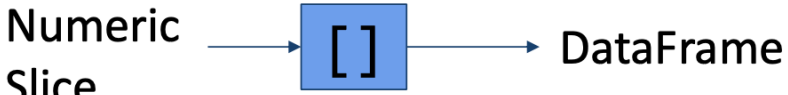
Using loc, iloc, and [ ]

`elections[0]` will not work unless the elections data frame has a column whose name is the numeric zero



Single Column Selection

Multiple Column Selection



(Multiple) Row Selection

Note: It is actually possible for columns to have names that are non-string types, e.g., numeric, datetime, etc

Agenda

Lecture 03

- Conditional selection
- Adding, removing, and modifying columns
- Useful utility functions
- Custom sorts
- Grouping

Conditional Selection

Lecture 03

- **Conditional selection**
- Adding, removing, and modifying columns
- Useful utility functions
- Custom sorts
- Grouping

Boolean Array Input for `.loc` and `[ ]`

We learned to extract data according to its **integer position** (`.iloc`) or its **label** (`.loc`)

What if we want to extract rows that satisfy a given *condition*?

- `.loc` and `[ ]` also accept boolean arrays as input.
- Rows corresponding to `True` are extracted; rows corresponding to `False` are not.

```
babynames_first_10_rows = babynames.loc[:9, :]
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134
5	CA	F	1910	Ruth	128
6	CA	F	1910	Evelyn	126
7	CA	F	1910	Alice	118
8	CA	F	1910	Virginia	101
9	CA	F	1910	Elizabeth	93

Boolean Array Input for `.loc` and `[ ]`

- `.loc` and `[ ]` also accept boolean arrays as input.
- Rows corresponding to `True` are extracted; rows corresponding to `False` are not.

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134
5	CA	F	1910	Ruth	128
6	CA	F	1910	Evelyn	126
7	CA	F	1910	Alice	118
8	CA	F	1910	Virginia	101
9	CA	F	1910	Elizabeth	93

```
babynames_first_10_rows[[True, False, True, False,
True, False, True, False, True, False]]
```

	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

Boolean Array Input

We can perform the same operation using `.loc`.

```
babynames_first_10_rows.loc[[True, False, True, False, True, False, True,
False, True, False], :]
```

	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

Boolean Array Input

Useful because boolean arrays can be generated by using logical operators on `Series`.

Length 407428 `Series` where every entry is either "True" or "False", where "True" occurs for every babyname with "Sex" = "F".

```
logical_operator = (babynames["Sex"] == "F")
0          True
1          True
2          True
3          True
4          True
          ...
407423    False
407424    False
407425    False
407426    False
407427    False
Name: Sex, Length: 407428, dtype: bool
```

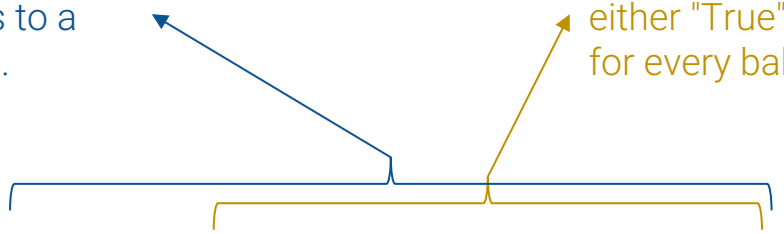
True in rows 0, 1, 2, …

Boolean Array Input

Useful because boolean arrays can be generated by using logical operators on `Series`.

Length 239537 `DataFrame` where every entry belongs to a babyname with "Sex" = "F".

Length 407428 `Series` where every entry is either "True" or "False", where "True" occurs for every babyname with "Sex" = "F".



```
babynames[(babynames["Sex"] == "F")]
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134
...	...	...	...	...	...
239532	CA	F	2022	Zemira	5
239533	CA	F	2022	Ziggy	5
239534	CA	F	2022	Zimal	5
239535	CA	F	2022	Zosia	5
239536	CA	F	2022	Zulay	5

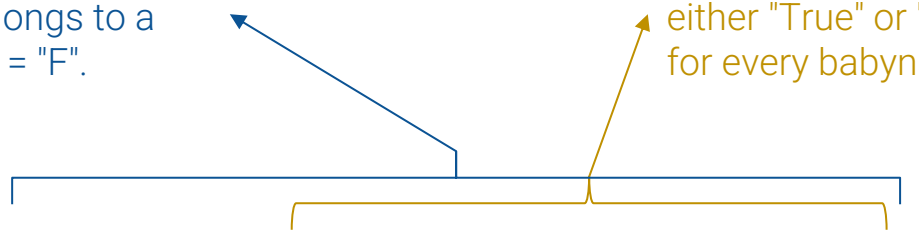
239537 rows × 5 columns

Boolean Array Input

Can also use **`.loc`**.

Length 239537 **`DataFrame`** where every entry belongs to a babyname with "Sex" = "F".

Length 407428 **`Series`** where every entry is either "True" or "False", where "True" occurs for every babyname with "Sex" = "F".



```
babynames.loc[babynames["Sex"] == "F", :]
```

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134
...	...	...	...	...	...
239532	CA	F	2022	Zemira	5
239533	CA	F	2022	Ziggy	5
239534	CA	F	2022	Zimal	5
239535	CA	F	2022	Zosia	5
239536	CA	F	2022	Zulay	5

239537 rows × 5 columns

Boolean Array Input

Boolean Series can be combined using various operators, allowing filtering of results by multiple criteria.

- The & operator allows us to apply logical_operator_1 **and** logical_operator_2
- The | operator allows us to apply logical_operator_1 **or** logical_operator_2

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

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134
...	...	...	...	...	...
342435	CA	M	1999	Yuuki	5
342436	CA	M	1999	Zakariya	5
342437	CA	M	1999	Zavier	5
342438	CA	M	1999	Zayn	5
342439	CA	M	1999	Zayne	5

342440 rows × 5 columns

Rows that have a Sex of "F" *or* are earlier than the year 2000 (or both!)

Bitwise Operators

`&` and `|` are examples of **bitwise operators**. They allow us to apply multiple logical conditions.

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

Symbol	Usage	Meaning
`~`	`~p`	Negation of `p`
`|`	`p | q`	`p` OR `q`
`&`	`p & q`	`p` AND `q`
`^`	`p ^ q`	`p` XOR `q` (exclusive or)

Alternatives to Direct Boolean Array Selection

Boolean array selection is a useful tool, but can lead to overly verbose code for complex conditions.

```
babynames[(babynames["Name"] == "Bella") |
          (babynames["Name"] == "Alex") |
          (babynames["Name"] == "Narges") |
          (babynames["Name"] == "Lisa")]
```

6289	CA	F	1923	Bella	5
7512	CA	F	1925	Bella	8
12368	CA	F	1932	Lisa	5
14741	CA	F	1936	Lisa	8
17084	CA	F	1939	Lisa	5
...	...	...	...	...	...
393248	CA	M	2018	Alex	495
396111	CA	M	2019	Alex	438
398983	CA	M	2020	Alex	379
401788	CA	M	2021	Alex	333
404663	CA	M	2022	Alex	344

317 rows × 5 columns

`pandas` provides **many** alternatives, for example:

- `.isin`
- `.str.startswith`
- `.groupby.filter` (we'll see this in Lecture 4)

Alternatives to Direct Boolean Array Selection

`pandas` provides **many** alternatives, for example:

- `.isin`
- `.str.startswith`
- `.groupby.filter` (see lecture 4)

```
names = ["Bella", "Alex", "Narges", "Lisa"]
babynames[babynames["Name"].isin(names)]
```

Returns a Boolean `Series` that is `True` when the corresponding name in `babynames` is Bella, Alex, Narges, or Lisa.

```
0         False
1         False
2         False
3         False
4         False
          ...
407423    False
407424    False
407425    False
407426    False
407427    False
Name: Name, Length: 407428, dtype: bool
```

Alternatives to Boolean Array Selection

`pandas` provides **many** alternatives, for example:

- `.isin`
- **`.str.startswith`**
- `.groupby.filter` (see lecture 4)

```
babynames[babynames["Name"].str.startswith("N")]
```

Returns a Boolean `Series` that is `True` when the corresponding name in `babynames` starts with "N".

```
0         False
1         False
2         False
3         False
4         False
          ...
407423    False
407424    False
407425    False
407426    False
407427    False
Name: Name, Length: 407428, dtype: bool
```

	State	Sex	Year	Name	Count
76	CA	F	1910	Norma	23
83	CA	F	1910	Nellie	20
127	CA	F	1910	Nina	11
198	CA	F	1910	Nora	6
310	CA	F	1911	Nellie	23
...	...	...	...	...	...
407319	CA	M	2022	Nilan	5
407320	CA	M	2022	Niles	5
407321	CA	M	2022	Nolen	5
407322	CA	M	2022	Noriel	5
407323	CA	M	2022	Norris	5

12229 rows × 5 columns

Adding, Removing, and Modifying Columns

Lecture 03

- Data extraction with `loc`, `iloc`, and `[]`
- Conditional selection
- **Adding, removing, and modifying columns**
- Useful utility functions
- Custom sorts
- Grouping

Syntax for Adding a Column

Adding a column is easy:

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

```
# 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
```

	State	Sex	Year	Name	Count	name_lengths
0	CA	F	1910	Mary	295	4
1	CA	F	1910	Helen	239	5
2	CA	F	1910	Dorothy	220	7
3	CA	F	1910	Margaret	163	8
4	CA	F	1910	Frances	134	7
...	...	...	...	...	...	...
407423	CA	M	2022	Zayvier	5	7
407424	CA	M	2022	Zia	5	3
407425	CA	M	2022	Zora	5	4
407426	CA	M	2022	Zuriel	5	6
407427	CA	M	2022	Zylo	5	4

407428 rows × 6 columns

Syntax for Modifying a Column

Modifying a column is very similar to adding a column.

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

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

	State	Sex	Year	Name	Count	name_lengths
0	CA	F	1910	Mary	295	3
1	CA	F	1910	Helen	239	4
2	CA	F	1910	Dorothy	220	6
3	CA	F	1910	Margaret	163	7
4	CA	F	1910	Frances	134	6
...	...	...	...	...	...	...
407423	CA	M	2022	Zayvier	5	6
407424	CA	M	2022	Zia	5	2
407425	CA	M	2022	Zora	5	3
407426	CA	M	2022	Zuriel	5	5
407427	CA	M	2022	Zylo	5	3

407428 rows × 6 columns

Syntax for Renaming a Column

Rename a column using the (creatively named) `.rename()` method.

- `.rename()` takes in a **dictionary** that maps old column names to their new ones.

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

	State	Sex	Year	Name	Count	Length
0	CA	F	1910	Mary	295	3
1	CA	F	1910	Helen	239	4
2	CA	F	1910	Dorothy	220	6
3	CA	F	1910	Margaret	163	7
4	CA	F	1910	Frances	134	6
...	...	...	...	...	...	...
407423	CA	M	2022	Zayvier	5	6
407424	CA	M	2022	Zia	5	2
407425	CA	M	2022	Zora	5	3
407426	CA	M	2022	Zuriel	5	5
407427	CA	M	2022	Zylo	5	3

407428 rows × 6 columns

Syntax for Dropping a Column (or Row)

Remove columns using the (also creatively named) `.drop` method.

- The `.drop()` method assumes you're dropping a row by default. Use `axis = "columns"` to drop a column instead.

```
babynames = babynames.drop("Length", axis = "columns")
```

	State	Sex	Year	Name	Count	Length
0	CA	F	1910	Mary	295	3
1	CA	F	1910	Helen	239	4
2	CA	F	1910	Dorothy	220	6
3	CA	F	1910	Margaret	163	7
4	CA	F	1910	Frances	134	6
...	...	...	...	...	...	...
407423	CA	M	2022	Zayvier	5	6
407424	CA	M	2022	Zia	5	2
407425	CA	M	2022	Zora	5	3
407426	CA	M	2022	Zuriel	5	5
407427	CA	M	2022	Zylo	5	3

407428 rows × 6 columns

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134
...	...	...	...	...	...
407423	CA	M	2022	Zayvier	5
407424	CA	M	2022	Zia	5
407425	CA	M	2022	Zora	5
407426	CA	M	2022	Zuriel	5
407427	CA	M	2022	Zylo	5

407428 rows × 5 columns

An Important Note: `DataFrame` Copies

Notice that we *re-assigned* **`babynames`** to an updated value on the previous slide.

```
babynames = babynames.drop("Length", axis = "columns")
```

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.

```
babynames.drop("Length", axis = "columns")
babynames
```

	State	Sex	Year	Name	Count	Length
0	CA	F	1910	Mary	295	3
1	CA	F	1910	Helen	239	4
2	CA	F	1910	Dorothy	220	6
3	CA	F	1910	Margaret	163	7
4	CA	F	1910	Frances	134	6
...	...	...	...	...	...	...

Our change was not applied!

Useful Utility Functions

Lecture 03

- Data extraction with `loc`, `iloc`, and `[]`
- Conditional selection
- Adding, removing, and modifying columns
- **Useful utility functions**
- Custom sorts
- Grouping

Pandas **`Series`** and **`DataFrames`** support a large number of operations, including mathematical operations, so long as the data is numerical.

```
yash_count = babynames[babynames["Name"] == "Yash"]["Count"]
```

```
np.mean(yash_count)
```

```
17.142857142857142
```

```
np.max(yash_count)
```

```
29
```

```
331824     8
334114     9
336390    11
338773    12
341387    10
343571    14
345767    24
348230    29
350889    24
353445    29
356221    25
358978    27
361831    29
364905    24
367867    23
370945    18
374055    14
376756    18
379660    18
383338     9
385903    12
388529    17
391485    16
394906    10
397874     9
400171    15
403092    13
406006    13
Name: Count, dtype: int64
```

Built-In `pandas` Methods

In addition to its rich syntax for indexing and support for other libraries (**`NumPy`**, native Python functions), **`pandas`** provides an enormous number of useful utility functions. Today, we'll discuss just a few:

- `size/shape`
- `describe`
- `sample`
- `value_counts`
- `unique`
- `sort_values`

The **`pandas`** library is rich in utility functions (we could spend the entire Course talking about them)! We encourage you to explore as you complete your assignments by Googling and reading documentation just as data scientists do.

`.shape` and `.size`

- `.shape` returns the shape of a `DataFrame` or `Series` in the form (number of rows, number of columns
- `.size` returns the total number of entries in a `DataFrame` or `Series` (number of rows times number of columns)

`babynames`

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134
...	...	...	...	...	...
407423	CA	M	2022	Zayvier	5
407424	CA	M	2022	Zia	5
407425	CA	M	2022	Zora	5
407426	CA	M	2022	Zuriel	5
407427	CA	M	2022	Zylo	5

407428 rows × 5 columns

```
babynames.shape
(407428, 5)
```

```
babynames.size
2037140
```

`.describe()`

- `.describe()` returns a "description" of a `DataFrame` or `Series` that lists summary statistics of the data

`babynames`

	State	Sex	Year	Name	Count
0	CA	F	1910	Mary	295
1	CA	F	1910	Helen	239
2	CA	F	1910	Dorothy	220
3	CA	F	1910	Margaret	163
4	CA	F	1910	Frances	134
...	...	...	...	...	...
407423	CA	M	2022	Zayvier	5
407424	CA	M	2022	Zia	5
407425	CA	M	2022	Zora	5
407426	CA	M	2022	Zuriel	5
407427	CA	M	2022	Zylo	5

407428 rows × 5 columns

`babynames.describe()`

	Year	Count
count	407428.000000	407428.000000
mean	1985.733609	79.543456
std	27.007660	293.698654
min	1910.000000	5.000000
25%	1969.000000	7.000000
50%	1992.000000	13.000000
75%	2008.000000	38.000000
max	2022.000000	8260.000000

`.describe()`

- A different set of statistics will be reported if `.describe()` is called on a `Series`.

```
babynames["Sex"].describe()
```

```
count     407428
unique         2
top            F
freq      239537
Name: Sex, dtype: object
```

`.sample()`

To sample a random selection of rows from a `DataFrame`, we use the `.sample()` method.

- By default, ***it is without replacement***. Use `replace=True` for **replacement**.
- Naturally, can be chained with other methods and operators (`iloc`, etc).

```
babynames.sample()
```

	State	Sex	Year	Name	Count
121141	CA	F	1992	Shanelle	28

```
babynames.sample(5).iloc[:, 2:]
```

	Year	Name	Count
44448	1961	Karyn	36
260410	1948	Carol	7
397541	2019	Arya	11
4767	1921	Sumiko	16
104369	1987	Thomas	11

```
babynames[babynames["Year"] == 2000]
         .sample(4, replace=True)
         .iloc[:, 2:]
```

	Year	Name	Count
151749	2000	Iridian	7
343560	2000	Maverick	14
149491	2000	Stacy	91
149212	2000	Angel	307

`.value_counts()`

The `Series.value_counts` method counts the number of occurrences of each unique value in a `Series` (it *counts* the number of times each *value* appears).

- Return value is also a `Series`.

```
babyname["Name"].value_counts()
```

```
Name
Jean         223
Francis      221
Guadalupe    218
Jessie       217
Marion       214
             ...
Renesme        1
Purity         1
Olanna         1
Nohea          1
Zayvier        1
Name: count, Length: 20437, dtype: int64
```

.unique()

The `Series.unique` method returns an array of every unique value in a `Series`.

```
babynames["Name"].unique()
```

```
array(['Mary', 'Helen', 'Dorothy', ..., 'Zae', 'Zai', 'Zayvier'],
      dtype=object)
```

`.sort_values()`

The `DataFrame.sort_values` and `Series.sort_values` methods 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.

```
babynames["Name"].sort_values()
```

```
366001      Aadan
384005      Aadan
369120      Aadan
398211    Aadarsh
370306      Aaden
            ...
220691      Zyrah
197529      Zyrah
217429      Zyrah
232167      Zyrah
404544      Zyrus
Name: Name, Length: 407428, dtype: object
```

`.sort_values()`

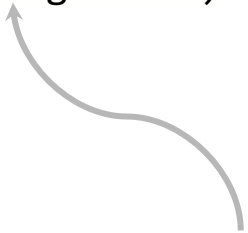
The `DataFrame.sort_values` and `Series.sort_values` methods sort a `DataFrame` (or `Series`).

- `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.**

```
babynames.sort_values(by = "Count", ascending=False)
```

	State	Sex	Year	Name	Count
268041	CA	M	1957	Michael	8260
267017	CA	M	1956	Michael	8258
317387	CA	M	1990	Michael	8246
281850	CA	M	1969	Michael	8245
283146	CA	M	1970	Michael	8196
...	...	...	...	...	...
317292	CA	M	1989	Olegario	5
317291	CA	M	1989	Norbert	5
317290	CA	M	1989	Niles	5
317289	CA	M	1989	Nikola	5
407427	CA	M	2022	Zylo	5

407428 rows × 5 columns



By default, rows are sorted in *ascending* order.

Custom Sorts

Lecture 03

- Data extraction with `loc`, `iloc`, and `[]`
- Conditional selection
- Adding, removing, and modifying columns
- Useful utility functions
- **Custom sorts**
- Grouping

Sorting By Length

Let’s try to solve the sorting problem with different approaches:

- We will create a temporary column, then sort on it.

Approach 1: Create a Temporary Column and Sort Based on the New Column

Sorting the `DataFrame` as usual:

```
# 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
babynames = babynames.sort_values(by = "name_lengths", ascending=False)
babynames.head(5)
```

	State	Sex	Year	Name	Count	name_lengths
334166	CA	M	1996	Franciscojavier	8	15
337301	CA	M	1997	Franciscojavier	5	15
339472	CA	M	1998	Franciscojavier	6	15
321792	CA	M	1991	Ryanchristopher	7	15
327358	CA	M	1993	Johnchristopher	5	15

Approach 2: Sorting Using the `key` Argument

```
babynames.sort_values("Name", key=lambda x: x.str.len(), ascending=False)
                 .head()
```

	State	Sex	Year	Name	Count
334166	CA	M	1996	Franciscojavier	8
327472	CA	M	1993	Ryanchristopher	5
337301	CA	M	1997	Franciscojavier	5
337477	CA	M	1997	Ryanchristopher	5
312543	CA	M	1987	Franciscojavier	5

Approach 3: Sorting Using the `map` Function

Suppose we want to sort by the number of occurrences of "dr" and “ea”s.

- Use the `Series.map` method.

```
def dr_ea_count(string):
    return string.count('dr') + string.count('ea')

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

	State	Sex	Year	Name	Count	dr_ea_count
115957	CA	F	1990	Deandrea	5	3
101976	CA	F	1986	Deandrea	6	3
131029	CA	F	1994	Leandrea	5	3
108731	CA	F	1988	Deandrea	5	3
308131	CA	M	1985	Deandrea	6	3

Grouping

Lecture 03

- Data extraction with `loc`, `iloc`, and `[]`
- Conditional selection
- Adding, removing, and modifying columns
- Useful utility functions
- Custom sorts
- **Grouping**

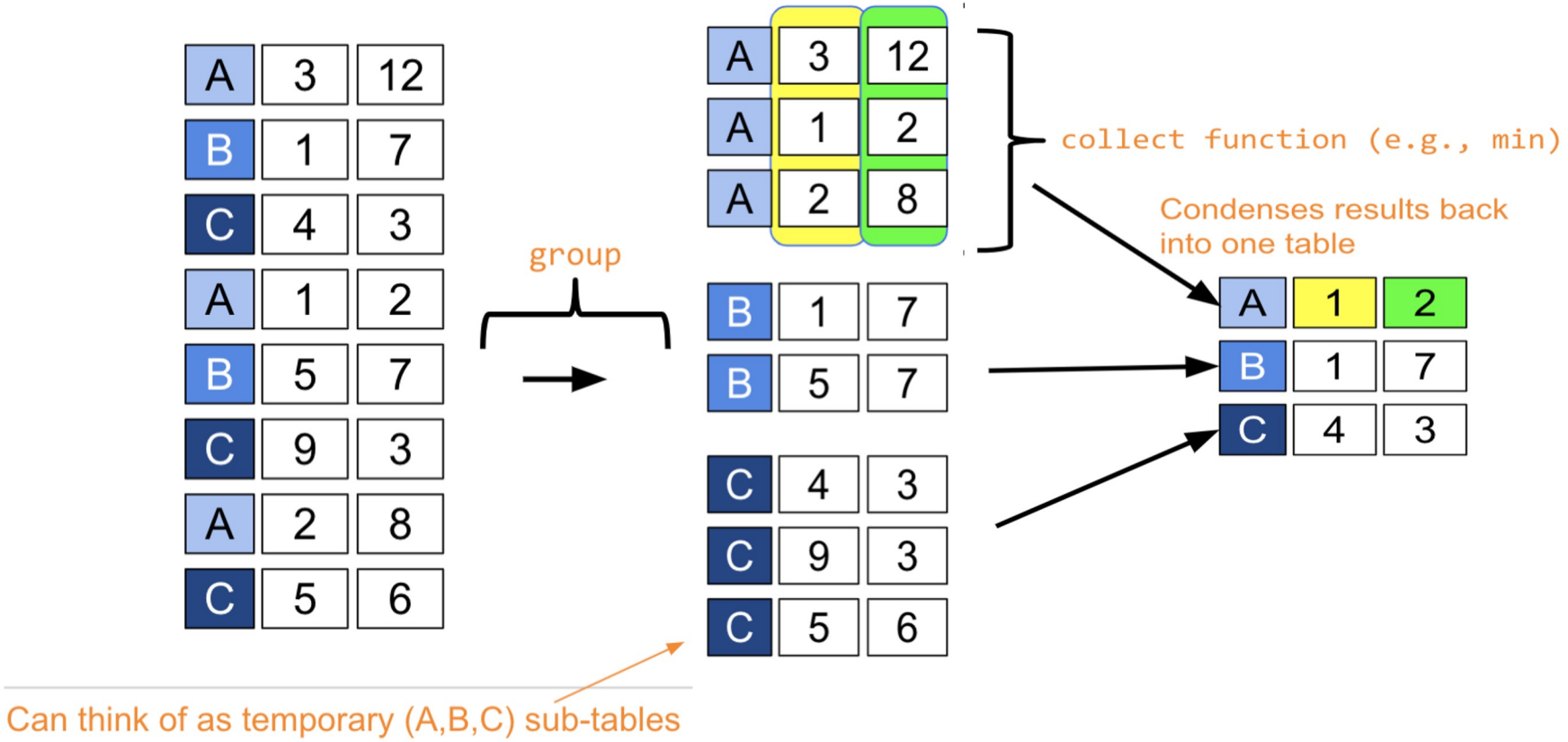
Why Group?

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 to 1) perform large operations, all at once and 2) summarize trends in a dataset.

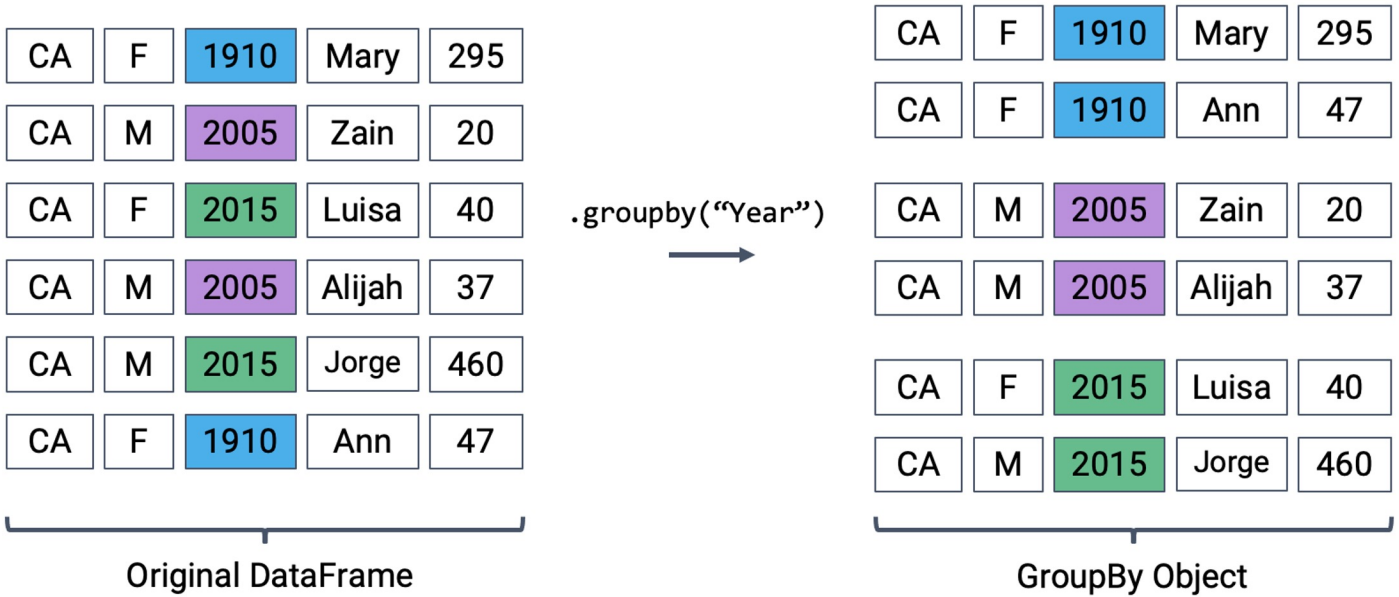
Visual Review of Grouping and Collection



`.groupby()`

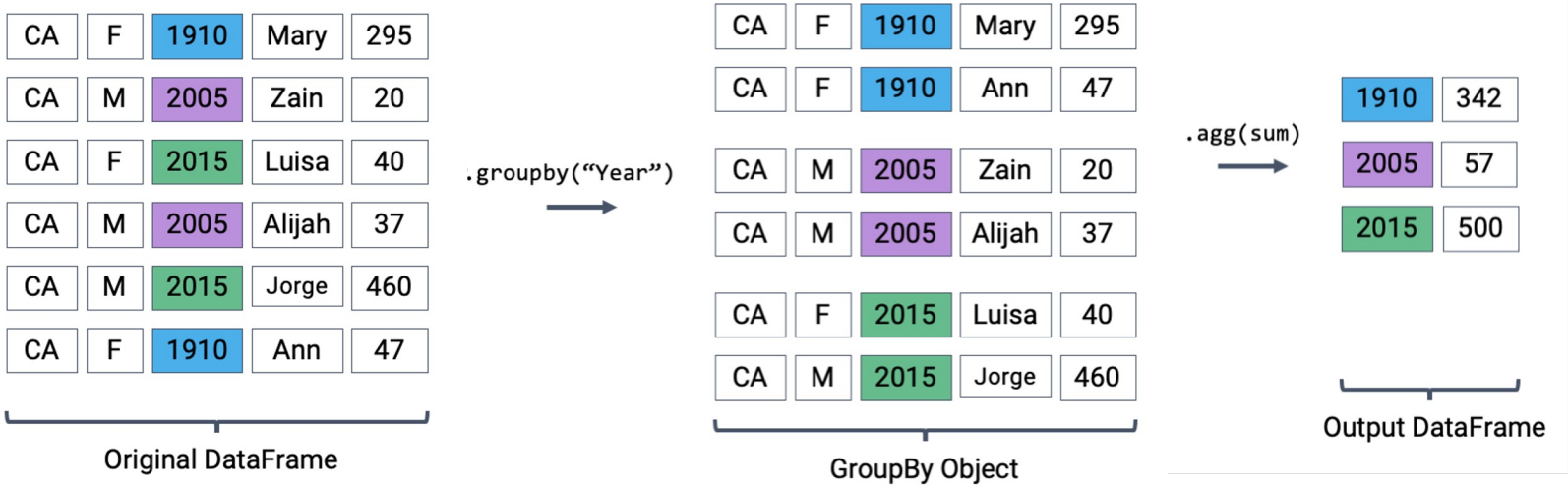
A `.groupby()` operation involves some combination of **splitting the object, applying a function**, and **combining the results**.

- Calling `.groupby()` generates `DataFrameGroupBy` objects → "mini" sub-DataFrames
- Each subframe contains all rows that correspond to the same group (here, a particular year)



`.groupby().agg()`

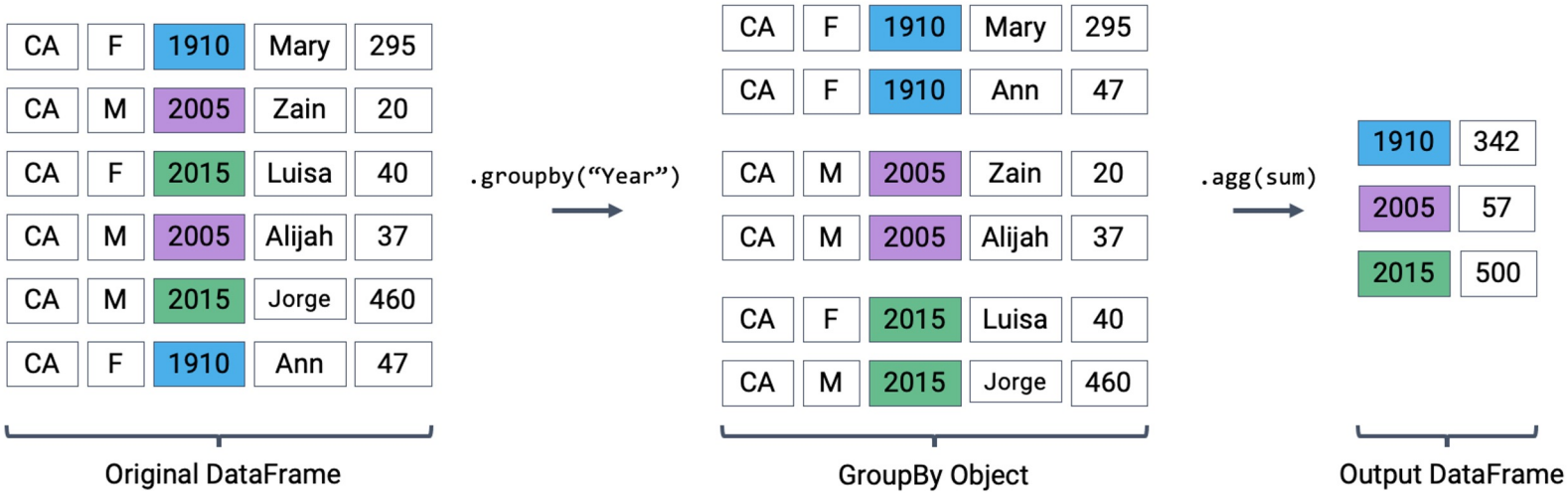
- We cannot work directly with **`DataFrameGroupBy`** objects! The diagram below is to help understand what goes on conceptually – in reality, we can't "see" the result of calling `.groupby`.
- Instead, we transform a **`DataFrameGroupBy`** object back into a DataFrame using `.agg`
  - `.agg` is how we apply an aggregation operation to the data.



Putting It All Together

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

- `babynames[["Year", "Count"]].groupby("Year").agg(sum)` computes the total number of babies born in each year.



Aggregation Functions

What goes inside of `.agg( )`?

- Any function that aggregates several values into one summary value
- Common examples:

In-Built Python Functions	NumPy Functions	In-Built `pandas` functions
`.agg(sum)`	`.agg(np.sum)`	`.agg("sum")`
`.agg(max)`	`.agg(np.max)`	`.agg("max")`
`.agg(min)`	`.agg(np.min)`	`.agg("min")`
	`.agg(np.mean)`	`.agg("mean")`
		`.agg("first")`
		`.agg("last")`

Some commonly-used aggregation functions can even be called directly, without the explicit use of `.agg( )`

```
babynames.groupby("Year").mean()
```

Putting Things Into Practice

Goal: Find the baby name with sex "F" that has fallen in popularity the most.

How do we define "fallen in popularity?"

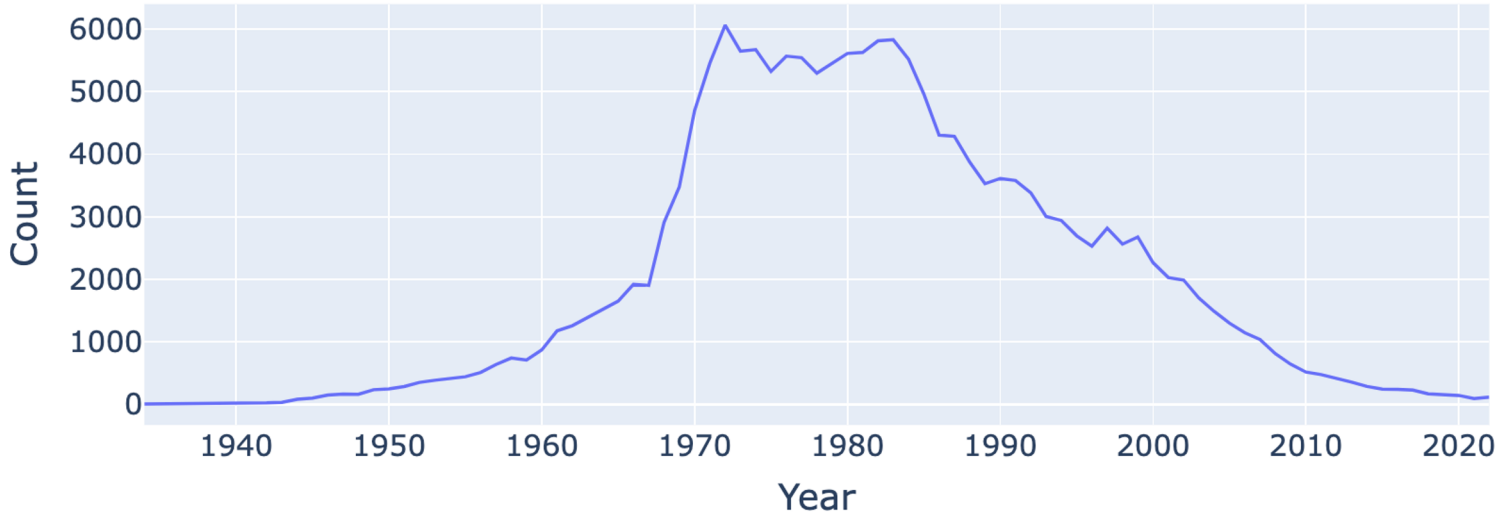
- Let’s create a metric: "ratio to peak" (RTP).
- The RTP is the ratio of babies born with a given name in 2022 to the *maximum* number of babies born with that name in *any* year.

Putting Things Into Practice

Goal: Find the baby name with sex "F" that has fallen in popularity the most.

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

Number of Jennifers Born in California Per Year



What Is "Popularity"?

Goal: Find the baby name with sex "F" that has fallen in popularity the most.

- Let's create a metric: "ratio to peak" (RTP).
- The RTP 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.

Example for "Jennifer":

- In 1972, we hit peak Jennifer. 6,065 Jennifers were born.
- In 2022, there were only 114 Jennifers.
- RTP is 114 / 6065 = 0.018796372629843364.

Calculating RTP

```
max_jenn = max(f_babynames[f_babynames["Name"] == "Jennifer"]["Count"])
```

```
6065
```

```
curr_jenn = f_babynames[f_babynames["Name"] == "Jennifer"]["Count"].iloc[-1]
```

```
114
```

Remember: `f_babynames` is sorted by year.
`.iloc[-1]` means "grab the latest year"

```
rtp = curr_jenn / max_jenn
```

```
0.018796372629843364
```

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

```
jenn_counts_ser = f_babynames[f_babynames["Name"] == "Jennifer"]["Count"]
ratio_to_peak(jenn_counts_ser)
```

```
0.018796372629843364
```

Calculating RTP Using `.groupby()`

`.groupby()` makes it easy to compute the RTP for all names at once!

```
rtp_table = f_babynames.groupby("Name")[["Year","Count"]].agg(ratio_to_peak)
```

Name	Year	Count
Aadhini	1.0	1.000000
Aadhira	1.0	0.500000
Aadhya	1.0	0.660000
Aadya	1.0	0.586207
Aahana	1.0	0.269231
...	...	...
Zyanya	1.0	0.466667
Zyla	1.0	1.000000
Zylah	1.0	1.000000
Zyra	1.0	1.000000
Zyrah	1.0	0.833333

13782 rows × 2 columns

Here, `collect = rtp`.

we'll implement it using `pandas`.

Answer

In the five rows shown, note the Year is 1.0 for every value.

Are there any rows for which Year is **not** 1.0?

A. Yes, names that appeared for the first time in 2022.
B. Yes, names that did not appear in 2022.
C. Yes, names whose peak Count was in 2022.
D. No, every row has a Year value of 1.0.

```
rtp_table = (
    f_babynames
    .groupby("Name")[["Year","Count"]]
    .agg(ratio_to_peak)
)
```

Name	Year	Count
Aadhini	1.0	1.000000
Aadhira	1.0	0.500000
Aadhya	1.0	0.660000
Aadya	1.0	0.586207
Aahana	1.0	0.269231
...	...	...
Zyanya	1.0	0.466667
Zyla	1.0	1.000000
Zylah	1.0	1.000000
Zyra	1.0	1.000000
Zyrah	1.0	0.833333

13782 rows × 2 columns

Answer

In the five rows shown, note the Year is 1.0 for every value.

Are there any rows for which Year is **not** 1.0?

A. Yes, names that appeared for the first time in 2022.
B. Yes, names that did not appear in 2022.
C. Yes, names whose peak Count was in 2022.
D. No, every row has a Year value of 1.0.

```
rtp_table = (
    f_babynames
    .groupby("Name")[["Year","Count"]]
    .agg(ratio_to_peak)
)
```

Name	Year	Count
Aadhini	1.0	1.000000
Aadhira	1.0	0.500000
Aadhya	1.0	0.660000
Aadya	1.0	0.586207
Aahana	1.0	0.269231
...	...	...
Zyanya	1.0	0.466667
Zyla	1.0	1.000000
Zylah	1.0	1.000000
Zyra	1.0	1.000000
Zyrah	1.0	0.833333

13782 rows × 2 columns

A Note on Nuisance Columns

At least as of the time of this slide creation executing our agg call results in a `TypeError`.

```
f_babynames.groupby("Name").agg(ratio_to_peak)
```

```
Cell In[110], line 5, in ratio_to_peak(series)
      1 def ratio_to_peak(series):
      2     """
      3     Compute the RTP for a Series containing the counts per year for a single name
      4     """
----> 5     return series.iloc[-1] / np.max(series)

TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

A Note on Nuisance Columns

Below, we explicitly select the column(s) we want to apply our aggregation function to **BEFORE** calling **agg**. This avoids the warning (and can prevent unintentional loss of data).

```
rtp_table = f_babynames.groupby("Name")[["Count"]].agg(ratio_to_peak)
```

	Count
Name	
Aadhini	1.000000
Aadhira	0.500000
Aadhya	0.660000
Aadya	0.586207
Aahana	0.269231
...	...
Zyanya	0.466667
Zyla	1.000000
Zylah	1.000000
Zyra	1.000000
Zyrah	0.833333

13782 rows × 1 columns

Renaming Columns After Grouping

By default, `.groupby` will not rename any aggregated columns (the column is still named "Count", even though it now represents the RTP.

For better readability, we may wish to rename “Count” to “Count RTP”

```
rtp_table = female_babynames.groupby("Name")[["Count"]].agg(ratio_to_peak)
rtp_table = rtp_table.rename(columns = {"Count": "Count RTP"})
```

Name	Count
Aadhini	1.000000
Aadhira	0.500000
Aadhya	0.660000
Aadya	0.586207
Aahana	0.269231
...	...

Name	Count RTP
Aadhini	1.000000
Aadhira	0.500000
Aadhya	0.660000
Aadya	0.586207
Aahana	0.269231
...	...

Some Data Science Payoff

By sorting `rtp_table` we can see the names whose popularity has decreased the most.

```
rtp_table.sort_values("Count RTP")
```

Name	Count RTP
Debra	0.001260
Debbie	0.002815
Carol	0.003180
Tammy	0.003249
Susan	0.003305
...	...
Fidelia	1.000000
Naveyah	1.000000
Finlee	1.000000
Roseline	1.000000
Aadhini	1.000000

13782 rows × 1 columns

Some Data Science Payoff

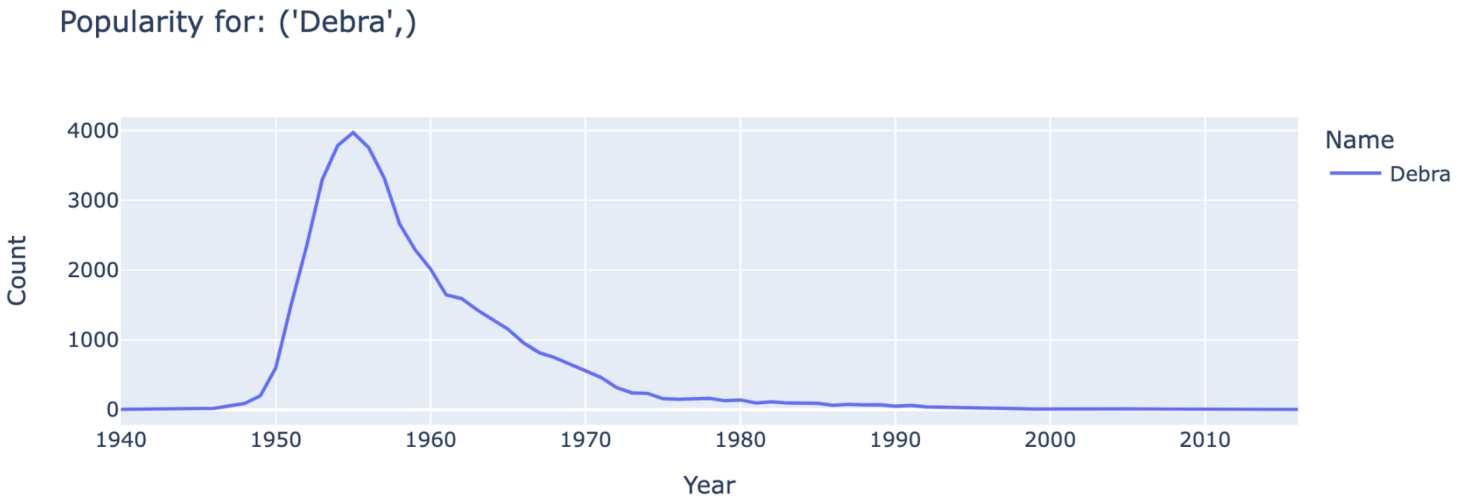
By sorting `rtp_table` we can see the names whose popularity has decreased the most.

```
rtp_table.sort_values("Count RTP")
```

Name	Count RTP
Debra	0.001260
Debbie	0.002815
Carol	0.003180
Tammy	0.003249
Susan	0.003305
...	...
Fidelia	1.000000
Naveyah	1.000000
Finlee	1.000000
Roseline	1.000000
Aadhini	1.000000

13782 rows × 1 columns

```
px.line(f_babynames[f_babyname["Name"] == "Debra")],
                        x = "Year", y = "Count")
```



We'll learn about plotting in week 3.

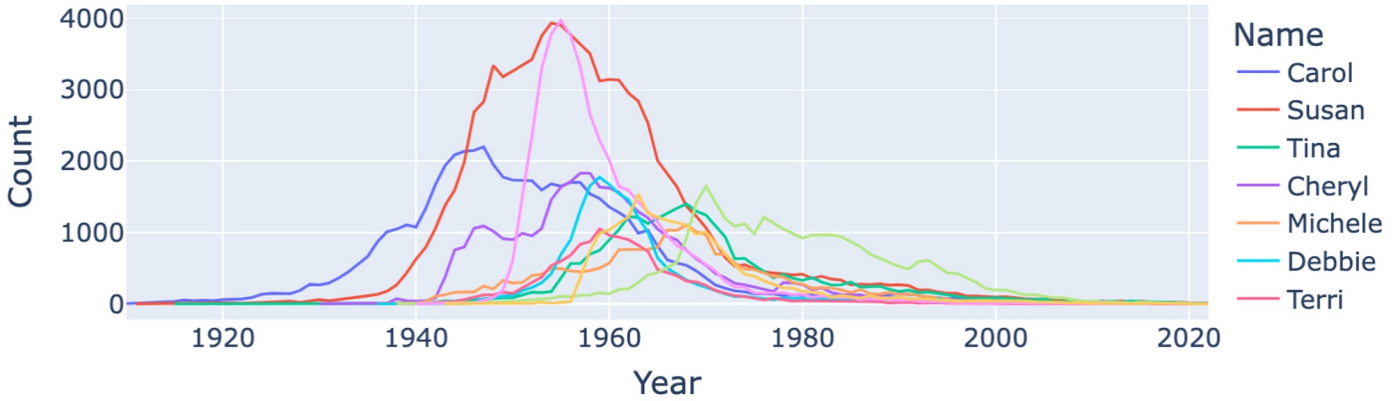
Some Data Science Payoff

We can get the list of the top 10 names and then plot popularity with::

```
top10 = rtp_table.sort_values("Count RTP").head(10).index
```

```
ndex(['Debra', 'Debbie', 'Carol', 'Tammy', 'Susan', 'Cheryl', 'Shannon',
       'Tina', 'Michele', 'Terri'],
      dtype='object', name='Name')
```

```
px.line(f_babynames[f_babyname["Name"].isin(top10)],
                          x = "Year", y = "Count", color = "Name")
```



Let’s Start Work on Notebook