slido

2130791

Join at slido.com
#2130791

ⓘ Click **Present with Slido** or install our Chrome extension to display joining instructions for participants while presenting.

LECTURE 3

Pandas, Part II

More on `pandas` (Selections and Utility Functions)

Data 100/Data 200, Spring 2025 @ UC Berkeley

Narges Norouzi and Josh Grossman

2130791

Goals for this Lecture

Lecture 03, Data 100 Spring 2025

Continue our tour of `pandas`

- Extracting data using `.iloc` and `[]`
- Extract data according to a condition
- Modify columns in a `DataFrame`
- Aggregate data

Last lecture: introducing tools

Today: "doing things"

2130791

Agenda

Lecture 03, Data 100 Spring 2025

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

2130791

Data Extraction with `iloc` and `[]`

Lecture 03, Data 100 Spring 2025

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

Integer-based Extraction: `.iloc`

2130791

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

- Example: Grab the 1st, 2nd, and 3rd columns of the `DataFrame`.

```
df.iloc[row_integers, column_integers]
```

The `.iloc` accessor allows us to specify the ***integers*** of rows and columns we wish to extract.

- Python convention: The first position has integer index 0.

Row integers

Column integers

	0	1	2	3	4	5
	Year	**Candidate**	**Party**	**Popular vote**	**Result**	**%**
0	**0** 1824	Andrew Jackson	Democratic-Republican	151271	loss	57.210122
1	**1** 1824	John Quincy Adams	Democratic-Republican	113142	win	42.789878
2	**2** 1828	Andrew Jackson	Democratic	642806	win	56.203927
3	**3** 1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	**4** 1832	Andrew Jackson	Democratic	702735	win	54.574789
	... ...	...	...	...	...	...

Integer-based Extraction: `.iloc`

2130791

Arguments to `.iloc` can be:

- A list.
- A slice (syntax is **exclusive** of the right hand side of the slice).
- A single value.

	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
...	...	...	...	...	...	...
182	2024	Donald Trump	Republican	77303568	win	49.808629
183	2024	Kamala Harris	Democratic	75019230	loss	48.336772
184	2024	Jill Stein	Green	861155	loss	0.554864
185	2024	Robert Kennedy	Independent	756383	loss	0.487357
186	2024	Chase Oliver	Libertarian Party	650130	loss	0.418895

Integer-based Extraction: `.iloc`

Arguments to `.iloc` can be:

- **A list.**
- A slice (syntax is **exclusive** of the right hand side of the slice).
- A single value.

```
elections.iloc[[1, 2, 3], [0, 1, 2]]
```

Select the rows at positions 1, 2, and 3.

	Year	Candidate	Party
1	1824	John Quincy Adams	Democratic-Republican
2	1828	Andrew Jackson	Democratic
3	1828	John Quincy Adams	National Republican

Select the columns at positions 0, 1, and 2.

Integer-based Extraction: `.iloc`

2130791

Arguments to `.iloc` can be:

- A list.
- **A slice** (syntax is **exclusive of the right hand side of the slice**).
- A single value.

```
elections.iloc[[1, 2, 3], 0:3]
```

	Year	Candidate	Party
1	1824	John Quincy Adams	Democratic-Republican
2	1828	Andrew Jackson	Democratic
3	1828	John Quincy Adams	National Republican

Select the rows at positions 1, 2, and 3.

Select *all* columns from integer 0 *to* integer 2.

Remember:
integer-based slicing is right-end exclusive!

Integer-based Extraction: `.iloc`

2130791

Just like `.loc`, we can use a colon with `.iloc` to extract all rows or all columns.

```
elections.iloc[:, 0:3]
```

	Year	Candidate	Result
0	1824	Andrew Jackson	loss
1	1824	John Quincy Adams	win
2	1828	Andrew Jackson	win
3	1828	John Quincy Adams	loss
4	1832	Andrew Jackson	win
...	...	...	...
182	2024	Donald Trump	win
183	2024	Kamala Harris	loss
184	2024	Jill Stein	loss
185	2024	Robert Kennedy	loss
186	2024	Chase Oliver	loss

Grab all rows of the columns at integers 0 to 2.

Integer-based Extraction: `.iloc`

2130791

Arguments to `.iloc` can be:

- A list.
- A slice (syntax is exclusive of the right hand side of the slice).
- **A single value.**

```
elections.iloc[[1, 2, 3], 1]
```

```
1    John Quincy Adams
2       Andrew Jackson
3    John Quincy Adams
Name: Candidate, dtype: object
```

As before, the result for a single value argument is a `Series`.

We have extracted row integers 1, 2, and 3 from the column at position 1.

```
elections.iloc[0, 1]
```

```
'Andrew Jackson'
```

We've extracted the string value with row position 0 and column position 1.

`.loc` vs `.iloc`

2130791

Remember:

- `.loc` performs label-based extraction
- `.iloc` performs integer-based extraction

When choosing between `.loc` and `.iloc`, you'll usually choose `.loc`.

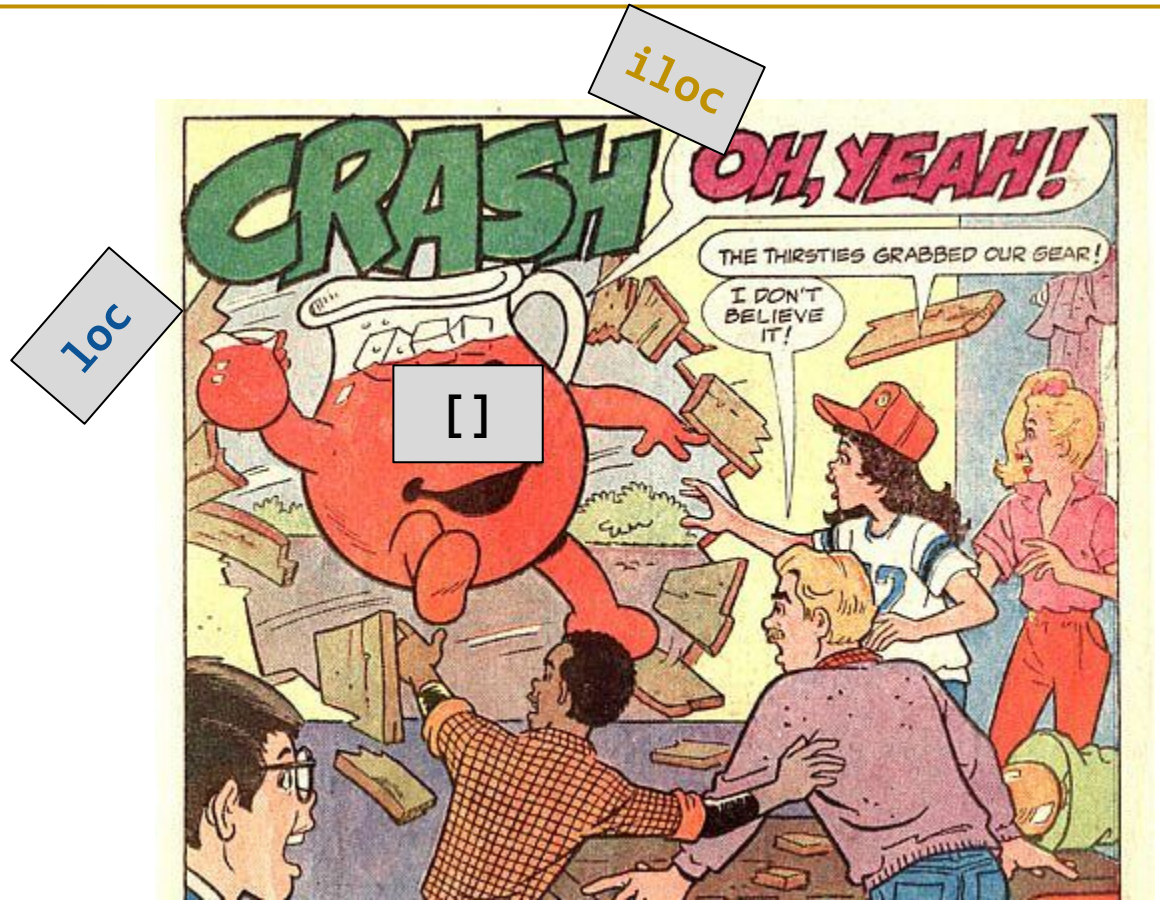
- Safer: If the order of data gets shuffled in a public database, your code still works.
- Readable: Easier to understand what `elections.loc[:, ["Year", "Candidate", "Result"]]` means than `elections.iloc[:, [0, 1, 4]]`

`.iloc` can still be useful.

- Example: If you have a `DataFrame` of movie earnings sorted by earnings, can use `.iloc` to get the median earnings for a given year (index into the middle).

… Just When It Was All Making Sense

2130791



Context-dependent Extraction: [ ]

2130791

Selection operators:

- `.loc` selects items by **label**. First argument is rows, second argument is columns.
- `.iloc` selects items by **integer**. First argument is rows, second argument is columns.
- `[]` only takes one argument, which may be:
  - A slice of **row numbers**.
  - A list of **column labels**.
  - A single **column label**.

That is, `[]` is context sensitive.

Let's see some examples.

Context-dependent Extraction: [ ]

2130791

`[ ]` only takes one argument, which may be:

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

```
elections[3:7]
```

	Year	Candidate	Party	Popular vote	Result	%
3	1828	John Quincy Adams	National Republican	500897	loss	43.796073
4	1832	Andrew Jackson	Democratic	702735	win	54.574789
5	1832	Henry Clay	National Republican	484205	loss	37.603628
6	1832	William Wirt	Anti-Masonic	100715	loss	7.821583

Context-dependent Extraction: [ ]

2130791

`[ ]` only takes one argument, which may be:

- A slice of row numbers.
- **A list of column labels.**
- A single column label.

```
elections[["Year", "Candidate", "Result"]]
```

	Year	Candidate	Result
0	1824	Andrew Jackson	loss
1	1824	John Quincy Adams	win
2	1828	Andrew Jackson	win
3	1828	John Quincy Adams	loss
4	1832	Andrew Jackson	win
...	...	...	...
182	2024	Donald Trump	win
183	2024	Kamala Harris	loss
184	2024	Jill Stein	loss
185	2024	Robert Kennedy	loss
186	2024	Chase Oliver	loss

Context-dependent Extraction: [ ]

2130791

`[ ]` only takes one argument, which may be:

- A slice of row numbers.
- A list of column labels.
- **A single column label.**

```
elections["Candidate"]
```

```
0          Andrew Jackson
1       John Quincy Adams
2          Andrew Jackson
3       John Quincy Adams
4          Andrew Jackson
              ...
182          Donald Trump
183         Kamala Harris
184            Jill Stein
185        Robert Kennedy
186          Chase Oliver
Name: Candidate, Length: 187, dtype: object
```

Extract the "Candidate" column as a `Series`.

Why Use [ ]?

2130791

In short: [ ] can be much more concise than `.loc` or `.iloc`

- Consider the case where we wish to extract the "Candidate" column. It is far simpler to write `elections["Candidate"]` than it is to write `elections.loc[:, "Candidate"]`

In practice, [ ] is often used over `.iloc` and `.loc` in data science work. Typing time adds up!

slido

2130791

Which of the following statements correctly return the value "blue fish" from the "weird" DataFrame?

Click **Present with Slido** or install our Chrome extension to activate this poll while presenting.

Conditional Selection

Lecture 03, Data 100 Spring 2025

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

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

2130791

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 `[ ]`

2130791

- `.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

2130791

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

2130791

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, …

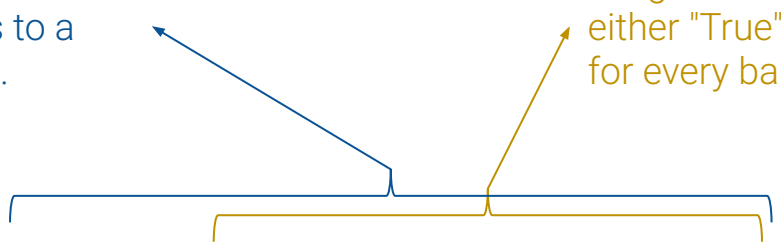
Boolean Array Input

2130791

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

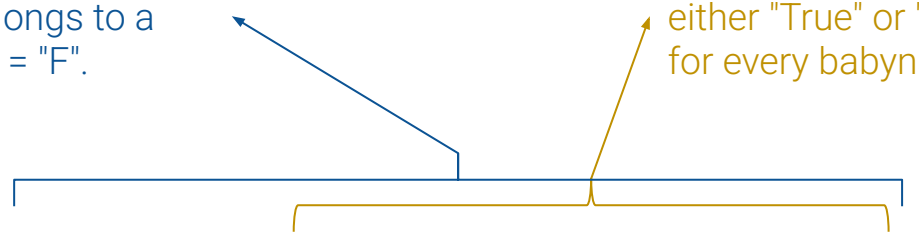
Boolean Array Input

2130791

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

2130791

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

2130791

`&` 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)

slido

2130791

Which of the following pandas statements returns a DataFrame of the first 3 baby names with Count > 250.

ⓘ Click **Present with Slido** or install our Chrome extension to activate this poll while presenting.

Alternatives to Direct Boolean Array Selection

2130791

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

2130791

`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

2130791

`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

Interlude

2130791

2130791

Adding, Removing, and Modifying Columns

Lecture 03, Data 100 Spring 2025

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

Syntax for Adding a Column

2130791

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

2130791

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

2130791

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)

2130791

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 100 Spring 2025

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

Pandas **`Series`** and **`DataFrames`** support a large number of operations, including mathematical operations, so long as the data is numerical. [Data 8 **`NumPy`** reference.]

```
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

2130791

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 summer 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`

2130791

- `.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()

2130791

- `.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()

2130791

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.

2130791

Lecture 3 ended here!

We will cover the rest in lecture 4

2130791

Custom Sorts

Lecture 03, Data 100 Spring 2025

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

Sorting By Length

2130791

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

- Assume that we want to sort entries based on the length of the name.
- Approach 1:
  - We will create a temporary column which holds the length of each name and then will sort on it.

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

2130791

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

2130791

```
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

2130791

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

2130791

LECTURE 3

Pandas, Part II

Content credit: Acknowledgments