

LECTURE 3

Pandas, Part II

More on Pandas (Conditional Selection, Utility Functions, Grouping, Aggregation)

CSCI 3022 CU Boulder

Maribeth Oscamou

Announcements

- No class Monday
- More office hours start next week
- HW 1 Coding Grades Posted to Gradescope (will be posted to Canvas next Wednesday with manual score)
- HW 2 will be released tonight - due next Thursday at 11:59pm
- nb2 session (NumPy): Tuesday 5pm-6pm Zoom

Goals for this Lecture

Continue our tour of **pandas**

- Extracting data using `[]` vs `loc` vs `iloc`
- Extract data according to a condition
- Modify columns in a **DataFrame**

Last lecture: introducing tools

Today: "doing things"

Data Extraction

- **Data extraction**
- Conditional selection
- Adding, removing, and modifying columns
-

Review: Context-dependent Extraction: []

[] 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



[] 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
...
177	2016	Jill Stein	loss
178	2020	Joseph Biden	win
179	2020	Donald Trump	loss
180	2020	Jo Jorgensen	loss
181	2020	Howard Hawkins	loss



[] 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
```

```
...
```

```
177      Jill Stein
178    Joseph Biden
179    Donald Trump
180    Jo Jorgensen
181    Howard Hawkins
```

```
Name: Candidate, Length: 182, dtype: object
```

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



Label-based Extraction: .loc

A more complex task: We want to extract data with specific column or index labels.

```
df.loc[row_labels, column_labels]
```

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

- We describe "labels" as the bolded text at the top and left of a **DataFrame**.

Row labels

	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

Column labels



Arguments to `.loc` can be:

- A list.
- A slice (syntax is inclusive 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
...
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

Label-based Extraction: .loc

Arguments to `.loc` can be:

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

```
elections.loc[[87, 25, 179], ["Year", "Candidate", "Result"]]
```

Select the rows with labels 87, 25, and 179.

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

Select the columns with labels "Year", "Candidate", and "Result".



Label-based Extraction: .loc

Arguments to `.loc` can be:

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

```
elections.loc[[87, 25, 179], "Popular vote": "%"]
```

Select the rows with labels 87, 25, and 179.



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

Select all columns *starting* from "Popular vote" *until* "%".




Label-based Extraction: .loc

To extract *all* rows or *all* columns, use a colon (:)

```
elections.loc[:, ["Year", "Candidate", "Result"]]
```

All rows for the columns with labels "Year", "Candidate", and "Result".

Ellipses (...) indicate more rows not shown. 

	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
...
177	2016	Jill Stein	loss
178	2020	Joseph Biden	win
179	2020	Donald Trump	loss
180	2020	Jo Jorgensen	loss
181	2020	Howard Hawkins	loss

```
elections.loc[[87, 25, 179], :]
```

All columns for the rows with labels 87, 25, 179.

	Candidate	Year	Party	Popular vote	Result	%
87	Herbert Hoover	1932	Republican	15761254	loss	39.830594
25	John C. Breckinridge	1860	Southern Democratic	848019	loss	18.138998
179	Donald Trump	2020	Republican	74216154	loss	46.858542



Label-based Extraction: .loc

Arguments to `.loc` can be:

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

```
elections.loc[[87, 25, 179], "Popular vote"]
```

```
87      15761254
```

```
25       848019
```

```
179     74216154
```

```
Name: Popular vote, dtype: int64
```

Wait, what? Why did everything get so ugly?

We've extracted a subset of the "Popular vote" column as a **Series**.

```
elections.loc[0, "Candidate"]
```

```
'Andrew Jackson'
```

We've extracted the string value with row label 0 and column label "Candidate".

Selection Operators Compared

Selection operators:

- `[]` only takes one argument, which may be:
 - A slice of **row numbers**.
 - A list of **column labels**.
 - A single **column label**.
- `.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.

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
...
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

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

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]
```

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 *all* columns from integer 0 to integer 2.

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



Integer-based Extraction: .iloc

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

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

	Year	Candidate	Party
0	1824	Andrew Jackson	Democratic-Republican
1	1824	John Quincy Adams	Democratic-Republican
2	1828	Andrew Jackson	Democratic
3	1828	John Quincy Adams	National Republican
4	1832	Andrew Jackson	Democratic
...
177	2016	Jill Stein	Green
178	2020	Joseph Biden	Democratic
179	2020	Donald Trump	Republican
180	2020	Jo Jorgensen	Libertarian
181	2020	Howard Hawkins	Green

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



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.

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).

Selection Operators Compared

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**.

Why Use []?

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!

Conditional Selection

- Data extraction with `loc`, `iloc`, and `[]`
- **Conditional selection**
- Adding, removing, and modifying columns
-

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 babynames 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, ...

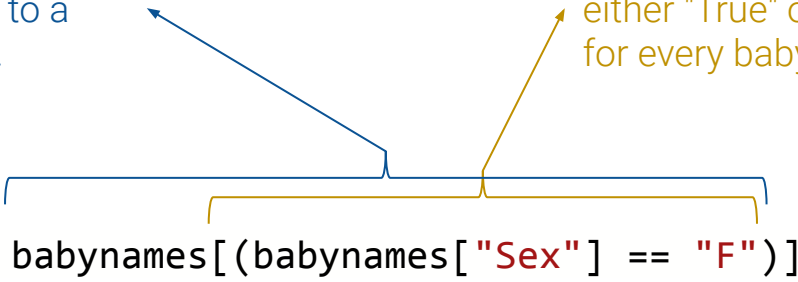


Boolean Array Input

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

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

Length 407428 **Series** where every entry is
either "True" or "False", where "True" occurs
for every babynames with "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 x 5 columns

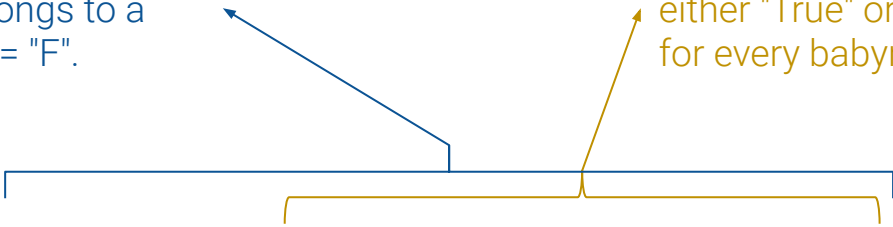


Boolean Array Input

Can also use `.loc`.

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

Length 407428 **Series** where every entry is
either "True" or "False", where "True" occurs
for every babynames 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 x 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
...
149050	CA	F	1999	Zareen	5
149051	CA	F	1999	Zeinab	5
149052	CA	F	1999	Zhane	5
149053	CA	F	1999	Zoha	5
149054	CA	F	1999	Zoila	5

Rows that have a Sex of "F" *and* are earlier than the year 2000

49055 rows x 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

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

342440 rows × 5 columns

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
<code>~</code>	<code>~p</code>	Negation of <code>p</code>
<code> </code>	<code>p q</code>	<code>p</code> OR <code>q</code>
<code>&</code>	<code>p & q</code>	<code>p</code> AND <code>q</code>
<code>^</code>	<code>p ^ q</code>	<code>p</code> XOR <code>q</code> (exclusive or)

Which of the following pandas statements returns a DataFrame with the same columns as babynames but only the rows of the first 3 baby names with Count > 250? (Select all that apply)

- A) `babynames[babynames["Count"] > 250].head(3)`
- B) `babynames.loc[babynames["Count"] > 250, :].iloc[0:2, :]`
- C) `babynames.loc[babynames["Count"] > 250, :].head(3)`
- D) `babynames.loc[babynames["Count"] > 250, :].iloc[0:3, :]`
- E) `babynames[babynames["Count"] > 250, :].head(3)`

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")]
```

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

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

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

Alternatives to Direct Boolean Array Selection

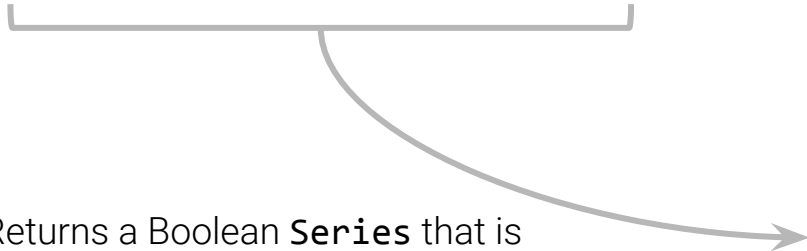
pandas provides **many** alternatives, for example:

- `.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

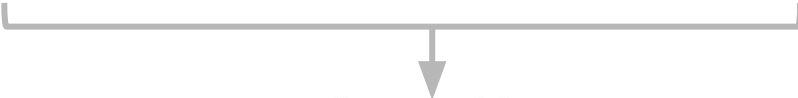
Name: Name, Length: 407428, dtype: bool

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")]
```



0	False
1	False
2	False
3	False
4	False

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

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 x 5 columns

Adding, Removing, and Modifying Columns

- Data extraction with `loc`, `iloc`, and `[]`
- Conditional selection
- **Adding, removing, and modifying columns**
-

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
babynames["name_lengths"] = babynames["Name"].str.len()

# Add a column named "name_lengths" that
# includes the length of each name
babynames["name_lengths"] = babynames["name_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 x 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"})
```



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.

	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 x 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



	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 x 6 columns

407428 rows x 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!



LECTURE 3

Pandas, Part II

Content credit: [Acknowledgments](#)

Extra Content: View and Copies in Pandas

When we subset a numpy array, the result is not always a new array; sometimes what NumPy returns is a view of the data in the original array.

Since pandas Series and DataFrames are backed by numpy arrays, it will probably come as no surprise that something similar sometimes happens in pandas. Unfortunately, while this behavior is relatively straightforward in numpy, in pandas there's just no getting around the fact that it's a hot mess.

The View/Copy Headache in pandas

In `numpy`, the rules for when you get views and when you don't are a little complicated, but they are consistent: certain behaviors (like simple indexing) will *always* return a view, and others (fancy indexing) will *never* return a view.

But in `pandas`, whether you get a view or not—and whether changes made to a view will propagate back to the original DataFrame—depends on the structure and data types in the original DataFrame.

https://www.practicaldatascience.org/html/views_and_copies_in_pandas.html

The Good News

To help address this issue, pandas has a built-in alert system that will **sometimes** warning you when you're in a situation that may cause problems, called the `SettingWithCopyWarning`, which you can see here:

```
[11]: df = pd.DataFrame({"a": np.arange(4), "b": ["w", "x", "y", "z"]})
      my_slice = df["a"]
      my_slice
```

```
[11]: 0    0
      1    1
      2    2
      3    3
      Name: a, dtype: int64
```

```
[12]: my_slice.iloc[1] = 2
```

```
/var/folders/fs/h_8_rwsn5hvg9mhp0txgc_s9v6191b/T/ipykernel_41268/1176285234.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
```

```
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#
my_slice.iloc[1] = 2
```

Any time you see a `SettingWithCopyWarning`, go up to where the possible view was created (in this case, `my_slice = df["a"]`) and add a `.copy()`:

```
[13]: my_slice = df["a"].copy()
      my_slice.iloc[1] = 2
```

The Bad News

The bad news is that the `SettingWithCopyWarning` will only flag one pattern where the copy-view problem crops up. Indeed, if you follow the link provided in the warning, you'll see it wasn't designed to address the copy-view problem *writ large*, but rather a more narrow behavior where the user tries to change a subset of a DataFrame incorrectly (we'll talk more about that in our coming readings). Indeed, you'll notice we didn't get a single `SettingWithCopyWarning` until the section where we started talking about that warning in particular (and I created an example designed to set it off).

So: if you see a `SettingWithCopyWarning` **do not** ignore it—find where you may have created a view or may have created a copy and add a `.copy()` so the error goes away. **But just because you don't see that warning doesn't mean you're in the clear!**

Which leads me to what I will admit is an infuriating piece of advice to have to offer: **if you take a subset for any purpose other than immediately analyzing, you should add `.copy()` to that subsetting.** Seriously. Just when in doubt, `.copy()`.

Best Practice 1: Avoid Chained Indexing

Chained indexing occurs when you use multiple indexing operations consecutively. For example, if you select a column and then select a row, this is chained indexing. It's important to avoid chained indexing because it can create views instead of copies, which can lead to unexpected behavior.

Here's an example of chained indexing:

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Chained indexing (avoid this!)
df_view = df['A'][0:2]

# This will raise a warning
df_view['A'] = [100, 200]
```

Output:

```
C:\Users\user\AppData\Local\Programs\Python\Python38\lib\site-packages\ipykernel_launcher.py:
A value is trying to be set on a copy of a slice from a DataFrame
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide
import sys
```

In this example, we use chained indexing to select the first two rows of the 'A' column. This creates a view of the DataFrame, which we then try to modify. However, we get a warning telling us that we are trying to modify a copy of the original DataFrame.

To avoid this, it's best to use `.loc` or `.iloc` to select rows and columns in a single operation.

Best Practice 2: Use .copy() When Needed

If you need to modify a DataFrame without affecting the original DataFrame, it's important to create a copy using the `.copy()` method. This will ensure that any changes you make to the copy will not affect the original DataFrame.

Here's an example:

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Create a copy of the DataFrame
df_copy = df.copy()

# Modify the copy
df_copy['A'] = [100, 200, 300]

# The original DataFrame is not affected
print(df)
print(df_copy)
```

Output:

```
   A  B
0  1  4
1  2  5
2  3  6

   A  B
0 100  4
1 200  5
2 300  6
```

In this example, we create a copy of the original DataFrame using the `.copy()` method. We then modify the copy by setting the 'A' column to a new value. The original DataFrame is not affected by this operation.

Best Practice 3: Use .iloc for Slice Assignment

If you need to modify a slice of a DataFrame, it's important to use `.iloc` to ensure that a copy is created. This will ensure that any changes you make to the slice will not affect the original DataFrame.

Here's an example:

```
import pandas as pd

# Create a DataFrame
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Modify a slice of the DataFrame
df.iloc[0:2]['A'] = [100, 200]

# The original DataFrame is not affected
print(df)
```

Output:

	A	B
0	1	4
1	2	5
2	3	6

In this example, we modify a slice of the DataFrame using `.iloc` to select the first two rows and the 'A' column. We then set the values of the 'A' column to a new value. However, the original DataFrame is not affected by this operation because a copy was created.