LECTURE 3

Pandas, Part I

Introduction to Pandas syntax, operators, and functions

Agenda

- Introduction to Exploratory Data Analysis
- Intro to Pandas
- Series, DataFrames, and Indices
- Slicing with loc, iloc, and []
- Demo

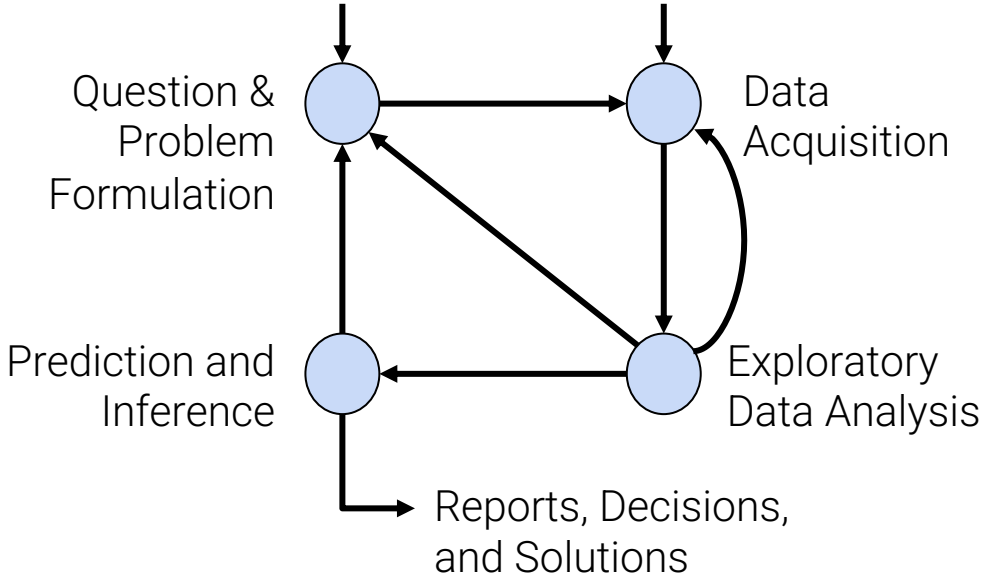
Get ready: lots of code incoming!

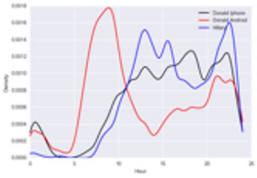
- Lecture: introduce high-level concepts
- Lab: practical experimentation

EDA and Data Wrangling

- **EDA and Data Wrangling**
- Intro to Pandas
- Series, DataFrames, and Indices
- Slicing with loc, iloc, and []
- Conditional selection
- Adding, removing, and modifying columns

Recall the Data Science Lifecycle



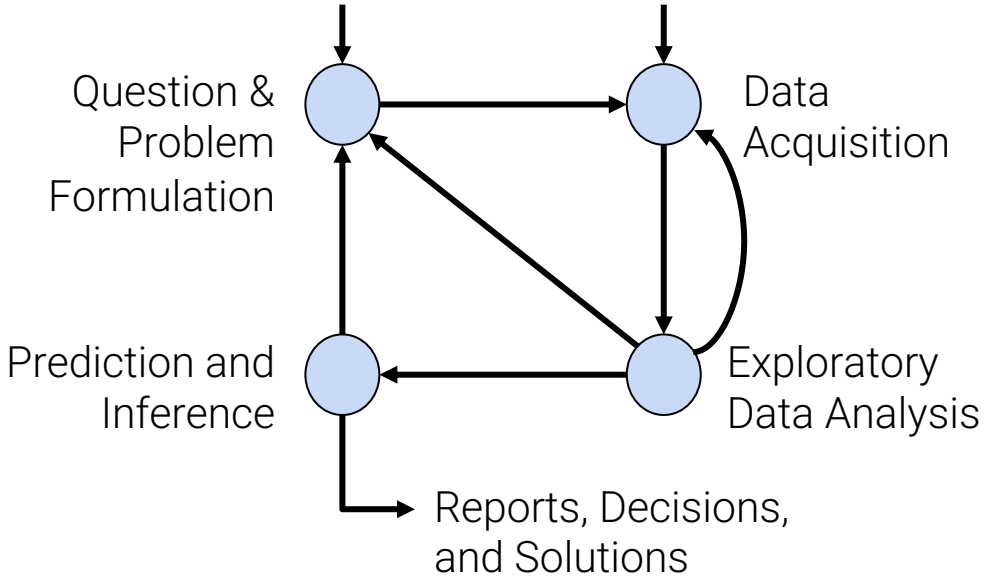


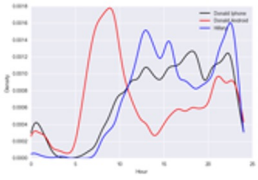
Congratulations!!!

You **have collected** or **have been given** a box of data.

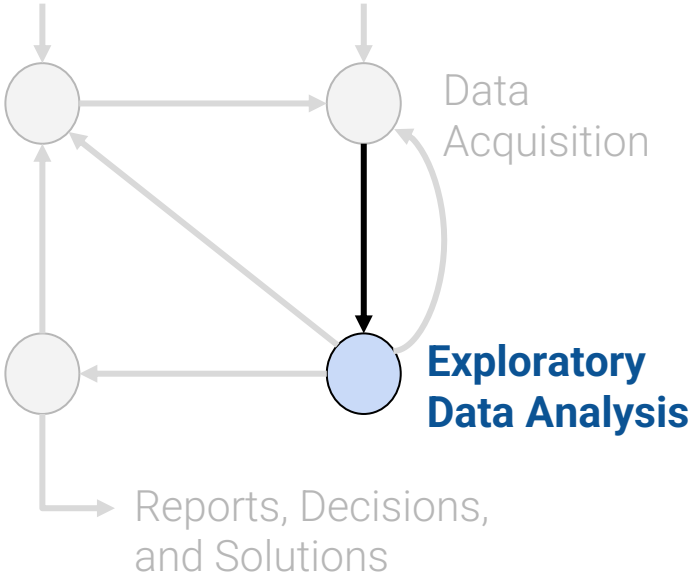
What do you do next?

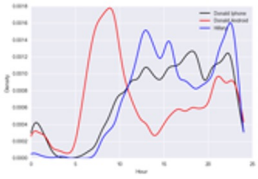
Recall the Data Science Lifecycle





Plan for First Few Weeks





(Weeks 1 and 2)

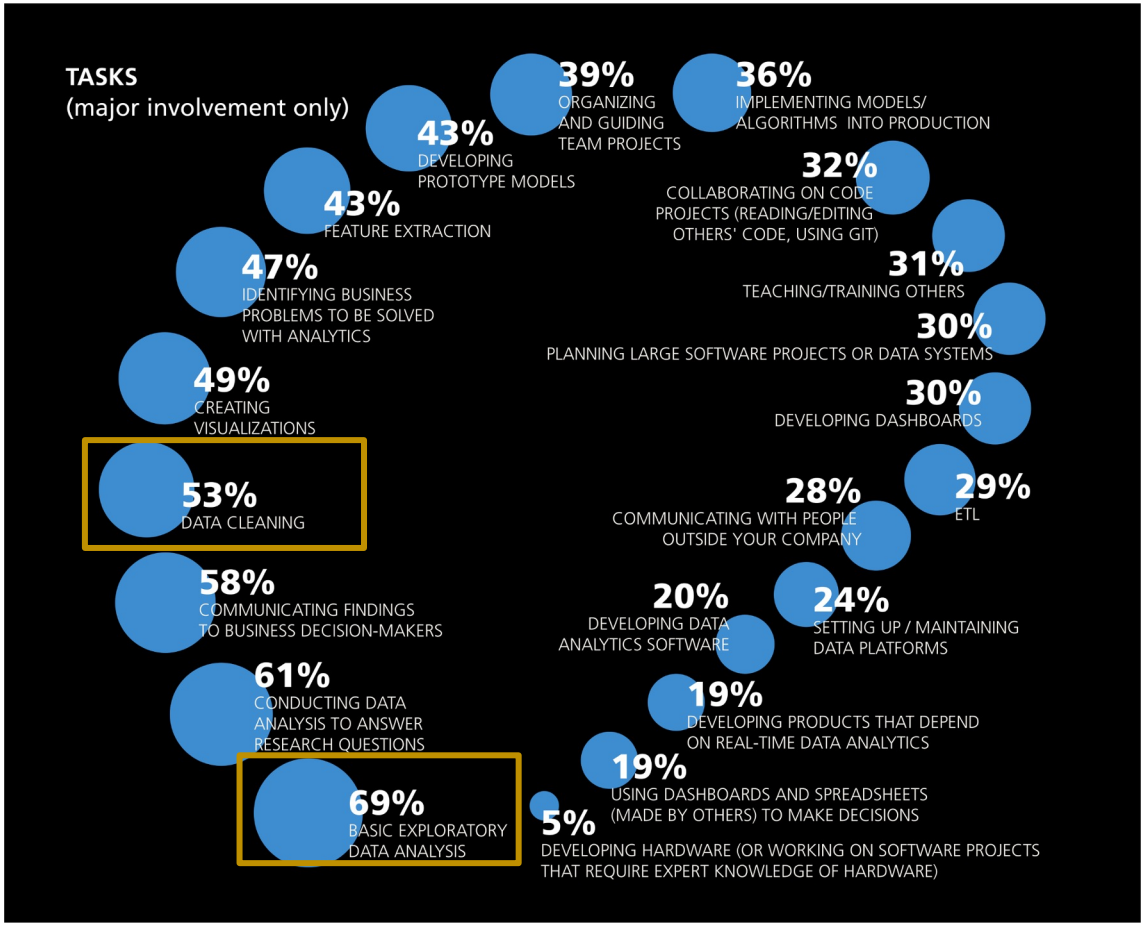
Exploring and Cleaning Tabular Data
pandas

(Weeks 2 and 3)

Data Science in Practice
EDA, Data Cleaning, Text processing (regular expressions)

Data science in industry

(from last time)



The major tasks that data scientists say they work on regularly.

Self-reported. Based on the results of the 2016 Data Science Salary Survey.

Intro to Pandas

- EDA and Data Wrangling
- **Intro to Pandas**
- Series, DataFrames, and Indices
- Slicing with loc, iloc, and []
- Conditional selection
- Adding, removing, and modifying columns

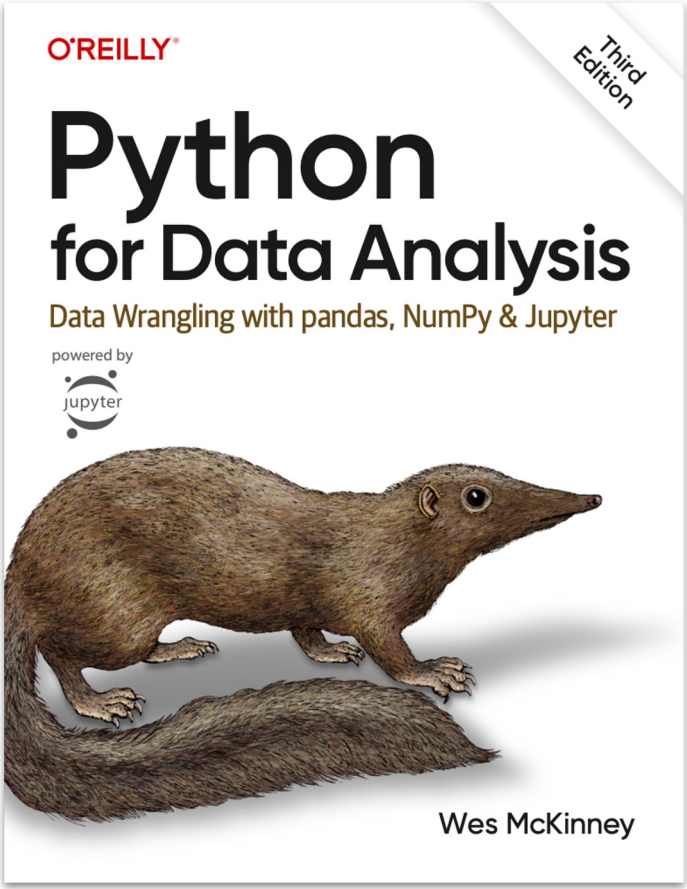
Introducing the Standard Python Data Science Tool: Pandas

The Python Data Analysis Library

- **Pandas** (derived from Panel Data) is a Data Analysis library to make data cleaning and analysis fast and convenient in Python.
- Pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data.
    - Numpy by contrast is best suited for working with homogenous numerical data.

Tabular data is one of the most common data formats.

You have free access to a fantastic book by the creator of Pandas!



Use O'Reilly Safari to read it for free:

https://www.oreilly.com/library-access

The "Open Edition" is freely available at
https://wesmckinney.com/book

Introducing the Standard Python Data Science Tool: pandas

Using **pandas**, we can:

- Arrange 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.

pandas is the standard tool across research and industry for working with tabular data.

Series, DataFrames, and Indices

- EDA and Data Wrangling
- Intro to Pandas
- **Series, DataFrames, and Indices**
- Slicing with loc, iloc, and []
- Conditional selection
- Adding, removing, and modifying columns

Data Scientists Love Tabular Data

"Tabular data" = data in a table.

Typically:

	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

A **row** represents one observation (here, a single person running for president in a particular year).

A **column** represents some characteristic, or feature, of that observation (here, the political party of that person).

To process tabular data, we'll use an industry-standard library called `pandas`.

DataFrames

In the "language" of pandas, we call a table a **`DataFrame`**.

We think of **`DataFrame`**s as collections of named columns, called **`Series`**.

	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

A `DataFrame`

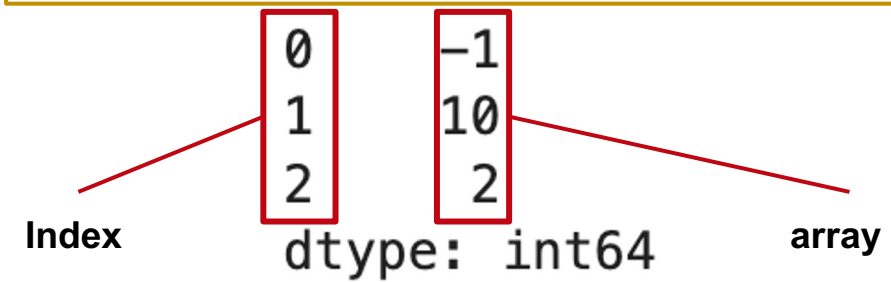
```
0          Andrew Jackson
1        John Quincy Adams
2          Andrew Jackson
3        John Quincy Adams
4          Andrew Jackson
               ...
177           Jill Stein
178         Joseph Biden
179         Donald Trump
180         Jo Jorgensen
181        Howard Hawkins
Name: Candidate, Length: 182, dtype: object
```

A `Series` named "Candidate"

Series

- A Series is a 1-dimensional **array-like object** containing a sequence of values of the same type and an associated array of data labels, called its **index**.

```
s = pd.Series([-1, 10, 2])
```




```
s.array
```

```
<PandasArray>
[-1, 10, 2]
Length: 3, dtype: int64
```

```
s.index
```

```
RangeIndex(start=0, stop=3, step=1)
```

Series - Custom Index

- We can provide index labels for items in a Series by passing an index list.

```
s = pd.Series([-1, 10, 2], index = ["a", "b", "c"])
```

```
a    -1
b    10
c     2
dtype: int64
```

```
s.index
```

```
Index(['a', 'b', 'c'], dtype='object')
```

- A Series index can also be changed.

```
s.index = ["first", "second", "third"])
```

```
first     -1
second    10
third      2
dtype: int64
```

```
s.index
```

```
Index(['first', 'second', 'third'], dtype='object')
```

Selection in Series

- We can select a single value or a set of values in a Series using:
  - A single label
  - A list of labels
  - A filtering condition

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

```
a    4
b   -2
c    0
d    6
dtype: int64
```

Selection in Series

- We can select a single value or a set of values in a Series using:
  - **A single label**
  - A list of labels
  - A filtering condition

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

```
a    4
b   -2
c    0
d    6
dtype: int64
```

```
s["a"]
```

```
4
```

Selection in Series

- We can select a single value or a set of values in a Series using:
  - A single label
  - **A list of labels**
  - A filtering condition

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

```
a    4
b   -2
c    0
d    6
dtype: int64
```

```
s[["a", "c"]]
```

```
a    4
c    0
dtype: int64
```

Selection in Series

- We can select a single value or a set of values in a Series using:
  - A single label
  - A list of labels
  - **A filtering condition**

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

```
a    4
b   -2
c    0
d    6
dtype: int64
```

- We first must apply a vectorized boolean operation to our Series that encodes the filter condition.
- Upon “indexing” in our Series with this condition, pandas selects only the rows with True values.

```
s > 0
```

```
a     True
b    False
c    False
d     True
dtype: bool
```

```
s[s > 0]
```

```
a    4
d    6
dtype: int64
```

DataFrames of Series!

Typically, we will work with **`Series`** using the perspective that they are columns in a **`DataFrame`**.

We can think of a **`DataFrame`** as a collection of **`Series`** that all share the same **`Index`**.

```
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`**

Creating a `DataFrame`

The syntax of creating **`DataFrame`** is:

```
pandas.DataFrame(data, index, columns)
```

Many approaches exist for creating a **`DataFrame`**. Here, we will go over the most popular ones.

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

Creating a `DataFrame`

The syntax of creating **`DataFrame`** is:

```
pandas.DataFrame(data, index, columns)
```

Many approaches exist for creating a **`DataFrame`**. Here, we will go over the most popular ones.

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

```
elections = pd.read_csv("data/elections.csv")
```

	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`**

Creating a `DataFrame`

The syntax of creating **`DataFrame`** is:

```
pandas.DataFrame(data, index, columns)
```

Many approaches exist for creating a **`DataFrame`**. Here, we will go over the most popular ones.

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

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

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

The DataFrame **`elections`** with **`"Year"`** as **`Index`**

Creating a `DataFrame`

Many approaches exist for creating a **`DataFrame`**. Here, we will go over the most popular ones.

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

```
pd.DataFrame([1, 2, 3],
             columns=["Numbers"])
```

	Numbers
0	1
1	2
2	3

```
pd.DataFrame([[1, "one"], [2, "two"]],
             columns = ["Number", "Description"])
```

	Number	Description
0	1	one
1	2	two

Creating a `DataFrame`

Many approaches exist for creating a **`DataFrame`**. Here, we will go over the most popular ones.

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

Specify columns of the **`DataFrame`**

```
pd.DataFrame({"Fruit":["Strawberry", "Orange"],
              "Price": [5.49, 3.99]})
```

```
pd.DataFrame([{"Fruit":"Strawberry", "Price":5.49},
              {"Fruit":"Orange", "Price":3.99}])
```

	Fruit	Price
0	Strawberry	5.49
1	Orange	3.99

Specify rows of the **`DataFrame`**

Creating a `DataFrame`

Many approaches exist for creating a **`DataFrame`**. Here, we will go over the most popular ones.

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

	A-column	B-column
r1	a1	b1
r2	a2	b2
r3	a3	b3

```
pd.DataFrame(s_a)
```

```
s_a.to_frame()
```

	0
r1	a1
r2	a2
r3	a3

Indices Are Not Necessarily Row Numbers

An **`Index`** (a.k.a. row labels) can also:

- Be non-numeric.
- Have a name, e.g. "Candidate".

```
# Creating a DataFrame from a CSV file and specifying the Index column
elections = pd.read_csv("data/elections.csv", index_col = "Candidate")
```

Candidate	Year	Party	Popular vote	Result	%
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

Indices Are Not Necessarily Unique

The row labels that constitute an index do not have to be unique.

- Left: The **index** values are all unique and numeric, acting as a row number.
- Right: The **index** values are named and non-unique.

	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

Year	Candidate	Party	%	Result
2008	Obama	Democratic	52.9	win
2008	McCain	Republican	45.7	loss
2012	Obama	Democratic	51.1	win
2012	Romney	Republican	47.2	loss
2016	Clinton	Democratic	48.2	loss
2016	Trump	Republican	46.1	win

Modifying Indices

- We can select a new column and set it as the index of the `DataFrame`.

Example: Setting the index to the "Party" column.

```
elections.set_index("Party")
```

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

Resetting the `Index`

- We can change our mind and reset the `Index` back to the default list of integers.

```
elections.reset_index()
```

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

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

Column Names Are Usually Unique!

Column names in `pandas` are almost always unique.

- Example: Really shouldn't have two columns named "Candidate".

	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

Retrieving the `Index`, `Columns`, and `shape`

Sometimes you'll want to extract the list of row and column labels.

```
elections.set_index("Party")
```

For row labels, use `DataFrame.index`:

```
elections.index
```

```
Index(['Democratic-Republican', 'Democratic-Republican', 'Democratic',
       'National Republican', 'Democratic', 'National Republican',
       'Anti-Masonic', 'Whig', 'Democratic', 'Whig',
       ...
       'Constitution', 'Republican', 'Independent', 'Libertarian',
       'Democratic', 'Green', 'Democratic', 'Republican', 'Libertarian',
       'Green'],
      dtype='object', name='Party', length=182)
```

For column labels, use `DataFrame.columns`:

```
elections.columns
```

```
Index(['Candidate', 'Year', 'Popular vote', 'Result', '%'], dtype='object')
```

For shape of the `DataFrame` we use `DataFrame.shape`:

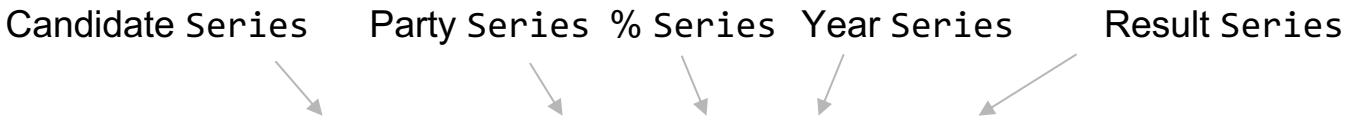
```
elections.shape
```

```
(182, 6)
```

The Relationship Between DataFrames, Series, and Indices

We can think of a **`DataFrame`** as a collection of **`Series`** that all share the same **`Index`**.

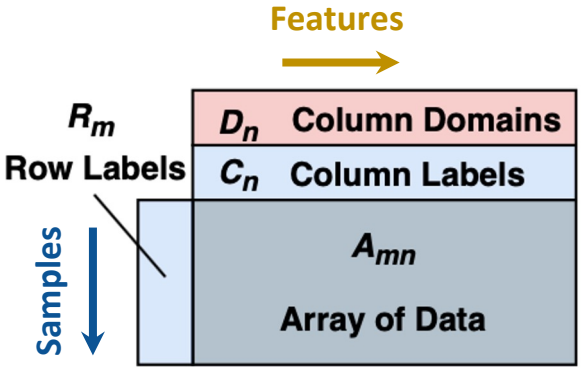
- Candidate, Party, %, Year, and Result **`Series`** all share an **`Index`** from 0 to 5.



	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

The World, a Statistician's View

A (statistical) population from which we draw **samples**.
Each sample has certain **features**.



	Year	Candidate	Party	Popular vote	Result	%
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

A generic DataFrame
(from https://arxiv.org/abs/2001.00888)

Here, our population is a census of all major party candidates since 1824.

The DataFrame API

The API for the `DataFrame` class is enormous.

- API: “Application Programming Interface”
- The API is the set of abstractions supported by the class.

Full documentation is at
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

- We will only consider a tiny portion of this API.

We want you to get familiar with the real world programming practice of… Searching!

- Answers to your questions are often found in the pandas documentation, stack overflow, etc.

With that warning, let’s dive in.

Slicing with loc, iloc, and [ ]

- EDA and Data Wrangling
- Intro to Pandas
- Series, DataFrames, and Indices
- **Slicing with loc, iloc, and [ ]**
- Conditional selection
- Adding, removing, and modifying columns

Extracting Data

One of the most basic tasks for manipulating a `DataFrame` is to extract rows and columns of interest. As we'll see, the large `pandas` API means there are many ways to do things.

Common ways we may want to extract data:

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

We'll find that all three of these methods are useful to us in data manipulation tasks.

`.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.

`elections`

	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

`elections.head(5)`

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

`elections.tail(5)`

	Candidate	Year	Party	Popular vote	Result	%
177	Jill Stein	2016	Green	1457226	loss	1.073699
178	Joseph Biden	2020	Democratic	81268924	win	51.311515
179	Donald Trump	2020	Republican	74216154	loss	46.858542
180	Jo Jorgensen	2020	Libertarian	1865724	loss	1.177979
181	Howard Hawkins	2020	Green	405035	loss	0.255731

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

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.

	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".

Integer-based Extraction: `.iloc`

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

- Example: Grab the 1st, 4th, 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`

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]
```

	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`

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.

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], 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`

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).

Context-dependent Extraction: [ ]

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: [ ]

[ ] 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: [ ]

`[ ]` 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

Context-dependent extraction: [ ]

`[ ]` 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`.

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!

Occasionally Useful Fact: Retrieving Row and Column Labels

Sometimes you'll want to extract the list of row and column labels.

For row labels, use `DataFrame.index`:

```
mottos.index
```

```
Index(['Alabama', 'Alaska', 'Arizona', 'Arkansas', 'California', 'Colorado',
       'Connecticut', 'Delaware', 'Florida', 'Georgia', 'Hawaii', 'Idaho',
       'Illinois', 'Indiana', 'Iowa', 'Kansas', 'Kentucky', 'Louisiana',
       'Maine', 'Maryland', 'Massachusetts', 'Michigan', 'Minnesota',
       'Mississippi', 'Missouri', 'Montana', 'Nebraska', 'Nevada',
       'New Hampshire', 'New Jersey', 'New Mexico', 'New York',
       'North Carolina', 'North Dakota', 'Ohio', 'Oklahoma', 'Oregon',
       'Pennsylvania', 'Rhode Island', 'South Carolina', 'South Dakota',
       'Tennessee', 'Texas', 'Utah', 'Vermont', 'Virginia', 'Washington',
       'West Virginia', 'Wisconsin', 'Wyoming'],
      dtype='object', name='State')
```

For column labels, use `DataFrame.columns`:

```
mottos.columns
```

```
Index(['Motto', 'Translation', 'Language', 'Date Adopted'], dtype='object')
```

Conditional selection

- EDA and Data Wrangling
- Intro to Pandas
- Series, DataFrames, and Indices
- Indexing 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 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
```

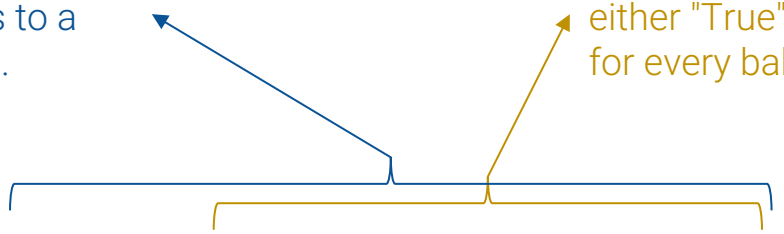
True in rows 0, 1, 2, …

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

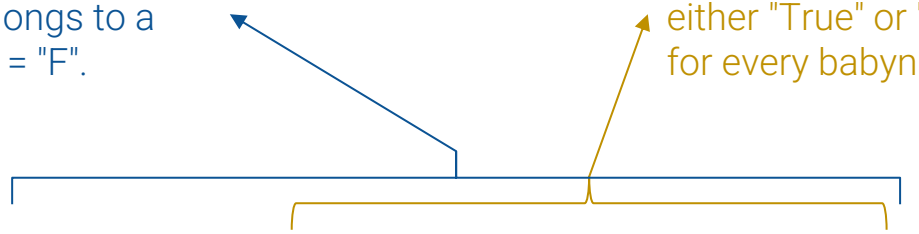
239537 rows × 5 columns

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

- EDA and Data Wrangling
- Intro to Pandas
- Series, DataFrames, and Indices
- Indexing 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
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!