

# **Principles and Techniques of Data Science**

**Data 100**

Bella Crouch

Kanu Grover

Ishani Gupta

Minh Phan

Milad Shafaie

Matthew Shen

# Table of contents

<b>Welcome</b>	<b>3</b>
About the Course Notes . . . . .	3
<b>1 Introduction</b>	<b>4</b>
1.1 Data Science Lifecycle . . . . .	5
1.1.1 Ask a Question . . . . .	5
1.1.2 Obtain Data . . . . .	6
1.1.3 Understand the Data . . . . .	6
1.1.4 Understand the World . . . . .	7
1.2 Conclusion . . . . .	7
<b>2 Pandas I</b>	<b>9</b>
2.1 Tabular Data . . . . .	9
2.2 Series, DataFrames, and Indices . . . . .	11
2.2.1 Series . . . . .	12
2.2.2 DataFrames . . . . .	16
2.2.3 Indices . . . . .	21
2.3 DataFrame Attributes: Index, Columns, and Shape . . . . .	26
2.4 Slicing in DataFrames . . . . .	26
2.4.1 Extracting data with <code>.head</code> and <code>.tail</code> . . . . .	27
2.4.2 Label-based Extraction: Indexing with <code>.loc</code> . . . . .	28
2.4.3 Integer-based Extraction: Indexing with <code>.iloc</code> . . . . .	32
2.4.4 Context-dependent Extraction: Indexing with <code>[]</code> . . . . .	36
2.5 Parting Note . . . . .	41

# Welcome

## About the Course Notes

This text offers supplementary resources to accompany lectures presented in the Fall 2023 Edition of the UC Berkeley course Data 100: Principles and Techniques of Data Science.

New notes will be added each week to accompany live lectures. See the full calendar of lectures on the [course website](#).

If you spot any typos or would like to suggest any changes, please email us. **Email:** [data100.instructors@berkeley.edu](mailto:data100.instructors@berkeley.edu)

# 1 Introduction

## Learning Outcomes

- Acquaint yourself with the overarching goals of Data 100
- Understand the stages of the data science lifecycle

Data science is an interdisciplinary field with a variety of applications and offers great potential to address challenging societal issues. By building data science skills, you can empower yourself to participate in and drive conversations that shape your life and society as a whole, whether that be fighting against climate change, launching diversity initiatives, or more.

The field is rapidly evolving; many of the key technical underpinnings in modern-day data science have been popularized during the early 21<sup>st</sup> century.

It is fundamentally human-centered and facilitates decision-making by quantitatively balancing tradeoffs. To quantify things reliably, we must use and analyze data appropriately, apply critical thinking and skepticism at every step of the way, and consider how our decisions affect others.

Ultimately, data science is the application of data-centric, computational, and inferential thinking to:

- Understand the world (science).
- Solve problems (engineering).

A true mastery of data science requires a deep theoretical understanding and strong grasp of domain expertise. This course will help you build on the former – specifically, the foundation of your technical knowledge, allowing you to take data and produce useful insights on the world’s most challenging and ambiguous problems.

## Course Goals

- Prepare you for advanced Berkeley courses in data management, machine learning, and statistics
- Enable you to launch a career in data science
- Empower you to address real-world problems through computational and inferential thinking

### Some Topics We'll Cover

- Pandas and NumPy
- Exploratory Data Analysis
- Regular Expressions
- Visualization
- Sampling
- Model design and loss formulation
- Linear Regression
- Gradient Descent
- Logistic Regression
- And so much more!

To set you up for success, we've organized concepts in Data 100 around the **data science lifecycle**: an *iterative* process that encompasses the various statistical and computational building blocks of data science.

## 1.1 Data Science Lifecycle

The data science lifecycle is a high-level overview of the data science workflow. It's a cycle of stages that a data scientist should explore as they conduct a thorough analysis of a data-driven problem.

There are many variations of the key ideas present in the data science lifecycle. In Data 100, we visualize the stages of the lifecycle using a flow diagram. Notice how there are two entry points.

### 1.1.1 Ask a Question

Whether by curiosity or necessity, data scientists constantly ask questions. For example, in the business world, data scientists may be interested in predicting the profit generated by a certain investment. In the field of medicine, they may ask whether some patients are more likely than others to benefit from a treatment.

Posing questions is one of the primary ways the data science lifecycle begins. It helps to fully define the question. Here are some things you should ask yourself before framing a question.

- What do we want to know?
  - A question that is too ambiguous may lead to confusion.
- What problems are we trying to solve?

- The goal of asking a question should be clear in order to justify your efforts to stakeholders.
- What are the hypotheses we want to test?
  - This gives a clear perspective from which to analyze final results.
- What are the metrics for our success?
  - This establishes a clear point to know when to conclude the project.

### 1.1.2 Obtain Data

The second entry point to the lifecycle is by obtaining data. A careful analysis of any problem requires the use of data. Data may be readily available to us, or we may have to embark on a process to collect it. When doing so, it is crucial to ask the following:

- What data do we have, and what data do we need?
  - Define the units of the data (people, cities, points in time, etc.) and what features to measure.
- How will we sample more data?
  - Scrape the web, collect manually, run experiments, etc.
- Is our data representative of the population we want to study?
  - If our data is not representative of our population of interest, then we can come to incorrect conclusions.

Key procedures: *data acquisition*, *data cleaning*

### 1.1.3 Understand the Data

Raw data itself is not inherently useful. It's impossible to discern all the patterns and relationships between variables without carefully investigating them. Therefore, translating pure data into actionable insights is a key job of a data scientist. For example, we may choose to ask:

- How is our data organized and what does it contain?
  - Knowing what the data says about the world helps us better understand the world.
- Do we have relevant data?
  - If the data we have collected is not useful to the question at hand, then we must collect more data.

- What are the biases, anomalies, or other issues with the data?
  - These can lead to many false conclusions if ignored, so data scientists must always be aware of these issues.
- How do we transform the data to enable effective analysis?
  - Data is not always easy to interpret at first glance, so a data scientist should strive to reveal the hidden insights.

Key procedures: *exploratory data analysis, data visualization*.

### 1.1.4 Understand the World

After observing the patterns in our data, we can begin answering our question. This may require that we predict a quantity (machine learning), or measure the effect of some treatment (inference).

From here, we may choose to report our results, or possibly conduct more analysis. We may not be satisfied with our findings, or our initial exploration may have brought up new questions that require new data.

- What does the data say about the world?
  - Given our models, the data will lead us to certain conclusions about the real world.
- Does it answer our questions or accurately solve the problem?
  - If our model and data can not accomplish our goals, then we must reform our question, model, or both.
- How robust are our conclusions and can we trust the predictions?
  - Inaccurate models can lead to false conclusions.

Key procedures: *model creation, prediction, inference*.

## 1.2 Conclusion

The data science lifecycle is meant to be a set of general guidelines rather than a hard set of requirements. In our journey exploring the lifecycle, we'll cover both the underlying theory and technologies used in data science. By the end of the course, we hope that you start to see yourself as a data scientist.

With that, we'll begin by introducing one of the most important tools in exploratory data analysis: **pandas**.



## 2 Pandas I

### Learning Outcomes

- Build familiarity with `pandas` and `pandas` syntax.
- Learn key data structures: `DataFrame`, `Series`, and `Index`.
- Understand methods for extracting data: `.loc`, `.iloc`, and `[]`.

In this sequence of lectures, we will dive right into things by having you explore and manipulate real-world data. We'll first introduce `pandas`, a popular Python library for interacting with **tabular data**.

### 2.1 Tabular Data

Data scientists work with data stored in a variety of formats. The primary focus of this class is understanding *tabular data* — data that is stored in a table.

Tabular data is one of the most common systems that data scientists use to organize data. This is in large part due to the simplicity and flexibility of tables. Tables allow us to represent each **observation**, or instance of collecting data from an individual, as its own *row*. We can record each observation's distinct characteristics, or **features**, in separate *columns*.

To see this in action, we'll explore the `elections` dataset, which stores information about political candidates who ran for president of the United States in previous years.

```
import pandas as pd
pd.read_csv("data/elections.csv")
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of ``S`

	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
5	1832	Henry Clay	National Republican	484205	loss	37.603628
6	1832	William Wirt	Anti-Masonic	100715	loss	7.821583
7	1836	Hugh Lawson White	Whig	146109	loss	10.005985
8	1836	Martin Van Buren	Democratic	763291	win	52.272472
9	1836	William Henry Harrison	Whig	550816	loss	37.721543
10	1840	Martin Van Buren	Democratic	1128854	loss	46.948787
11	1840	William Henry Harrison	Whig	1275583	win	53.051213
12	1844	Henry Clay	Whig	1300004	loss	49.250523
13	1844	James Polk	Democratic	1339570	win	50.749477
14	1848	Lewis Cass	Democratic	1223460	loss	42.552229
15	1848	Martin Van Buren	Free Soil	291501	loss	10.138474
16	1848	Zachary Taylor	Whig	1360235	win	47.309296
17	1852	Franklin Pierce	Democratic	1605943	win	51.013168
18	1852	John P. Hale	Free Soil	155210	loss	4.930283
19	1852	Winfield Scott	Whig	1386942	loss	44.056548
20	1856	James Buchanan	Democratic	1835140	win	45.306080
21	1856	John C. Frémont	Republican	1342345	loss	33.139919
22	1856	Millard Fillmore	American	873053	loss	21.554001
23	1860	Abraham Lincoln	Republican	1855993	win	39.699408
24	1860	John Bell	Constitutional Union	590901	loss	12.639283
25	1860	John C. Breckinridge	Southern Democratic	848019	loss	18.138998
26	1860	Stephen A. Douglas	Northern Democratic	1380202	loss	29.522311
27	1864	Abraham Lincoln	National Union	2211317	win	54.951512
28	1864	George B. McClellan	Democratic	1812807	loss	45.048488
29	1868	Horatio Seymour	Democratic	2708744	loss	47.334695
30	1868	Ulysses Grant	Republican	3013790	win	52.665305
31	1872	Horace Greeley	Liberal Republican	2834761	loss	44.071406
32	1872	Ulysses Grant	Republican	3597439	win	55.928594
33	1876	Rutherford Hayes	Republican	4034142	win	48.471624
34	1876	Samuel J. Tilden	Democratic	4288546	loss	51.528376
35	1880	James B. Weaver	Greenback	308649	loss	3.352344
36	1880	James Garfield	Republican	4453337	win	48.369234
37	1880	Winfield Scott Hancock	Democratic	4444976	loss	48.278422
38	1884	Benjamin Butler	Anti-Monopoly	134294	loss	1.335838
39	1884	Grover Cleveland	Democratic	4914482	win	48.884933
40	1884	James G. Blaine	Republican	4856905	loss	48.312208
41	1884	John St. John	Prohibition	147482	loss	1.467021
42	1888	Alson Streeter	Union Labor	146602	loss	1.288861
43	1888	Benjamin Harrison	Republican	5443633	win	47.858041
44	1888	Clinton B. Fisk	Prohibition	249819	loss	2.196299
45	1888	Grover Cleveland	Democratic	5534488	loss	48.656799
46	1892	Benjamin Harrison	Republican	5176108	loss	42.984101
47	1892	Grover Cleveland	Democratic	5553898	win	46.121393
48	1892	James B. Weaver	Populist	1041028	loss	8.645038
49	1892	John Bidwell	Prohibition	270879	loss	2.249468
50	1896	John M. Palmer	National Democratic	134645	loss	0.969566
51	1896	Joshua Levering	Prohibition	131312	loss	0.945565

In the `elections` dataset, each row represents one instance of a candidate running for president in a particular year. For example, the first row represents Andrew Jackson running for president in the year 1824. Each column represents one characteristic piece of information about each presidential candidate. For example, the column named “Result” stores whether or not the candidate won the election.

Your work in Data 8 helped you grow very familiar with using and interpreting data stored in a tabular format. Back then, you used the `Table` class of the `datascience` library, a special programming library created specifically for Data 8 students.

In Data 100, we will be working with the programming library `pandas`, which is generally accepted in the data science community as the industry- and academia-standard tool for manipulating tabular data (as well as the inspiration for Petey, our panda bear mascot).

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 (our friends from Data 8).
- Perform vectorized computations to speed up our analysis (Lab 1).

## 2.2 Series, DataFrames, and Indices

To begin our work in `pandas`, we must first import the library into our Python environment. This will allow us to use `pandas` data structures and methods in our code.

```
# `pd` is the conventional alias for Pandas, as `np` is for NumPy
import pandas as pd
```

There are three fundamental data structures in `pandas`:

1. **Series**: 1D labeled array data; best thought of as columnar data.
2. **DataFrame**: 2D tabular data with rows and columns.
3. **Index**: A sequence of row/column labels.

`DataFrames`, `Series`, and `Indices` can be represented visually in the following diagram, which considers the first few rows of the `elections` dataset.

The elections DataFrame

	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

Index of the elections DataFrame

Index of the Result Series

A Series named Result

0	loss
1	win
2	win
3	loss
4	win

Name: Result, dtype: object

Notice how the **DataFrame** is a two-dimensional object — it contains both rows and columns. The **Series** above is a singular column of this **DataFrame**, namely the **Result** column. Both contain an **Index**, or a shared list of row labels (the integers from 0 to 4, inclusive).

### 2.2.1 Series

A **Series** represents a column of a **DataFrame**; more generally, it can be any 1-dimensional array-like object. It contains:

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

In the cell below, we create a **Series** named `s`.

```
s = pd.Series(["welcome", "to", "data 100"])
s
```

	0
0	welcome
1	to
2	data 100

```
s.values # Data values contained within the Series
```

```
array(['welcome', 'to', 'data 100'], dtype=object)
```

```
s.index # The Index of the Series
```

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

By default, the Index of a Series is a sequential list of integers beginning from 0. Optionally, a manually specified list of desired indices can be passed to the `index` argument.

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `Series`

	0
a	-1
b	10
c	2

```
s.index
```

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

Indices can also be changed after initialization.

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `Series`

	0
first	-1
second	10
third	2

```
s.index
```

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

### 2.2.1.1 Selection in Series

Much like when working with NumPy arrays, we can select a single value or a set of values from a **Series**. To do so, there are three primary methods:

1. A single label.
2. A list of labels.
3. A filtering condition.

To demonstrate this, let's define the Series `ser`.

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `S`

0	
a	4
b	-2
c	0
d	6

#### 2.2.1.1.1 A Single Label

```
ser["a"] # We return the value stored at the Index label "a"
```

4

#### 2.2.1.1.2 A List of Labels

```
ser[["a", "c"]] # We return a *Series* of the values stored at the Index labels "a" and "c"
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to\_latex` is expected to utilise the base implementation of `Series.to\_latex`

	0
a	4
c	0

#### 2.2.1.1.3 A Filtering Condition

Perhaps the most interesting (and useful) method of selecting data from a Series is by using a filtering condition.

First, we apply a boolean operation to the **Series**. This creates a **new Series of boolean values**.

```
ser > 0 # Filter condition: select all elements greater than 0
```

	0
a	True
b	False
c	False
d	True

We then use this boolean condition to index into our original **Series**. **pandas** will select only the entries in the original **Series** that satisfy the condition.

```
ser[ser > 0]
```

	0
a	4
d	6

## 2.2.2 DataFrames

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**.

In Data 8, you encountered the **Table** class of the **datascience** library, which represented tabular data. In Data 100, we'll be using the **DataFrame** class of the **pandas** library.

### 2.2.2.1 Creating a DataFrame

There are many ways to create a **DataFrame**. Here, we will cover the most popular approaches:

1. From a CSV file.
2. Using a list and column name(s).
3. From a dictionary.
4. From a **Series**.

More generally, the syntax for creating a **DataFrame** is: `pandas.DataFrame(data, index, columns)`.

#### 2.2.2.1.1 From a CSV file

In Data 100, our data are typically stored in a CSV (comma-separated values) file format. We can import a CSV file into a **DataFrame** by passing the data path as an argument to the following **pandas** function. `pd.read_csv("filename.csv")`

With our new understanding of **pandas** in hand, let's return to the **elections** dataset from before. Now, we can recognize that it is represented as a **pandas DataFrame**.

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to\_latex` is expected to utilise the base implementation of `S



	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
5	1832	Henry Clay	National Republican	484205	loss	37.603628
6	1832	William Wirt	Anti-Masonic	100715	loss	7.821583
7	1836	Hugh Lawson White	Whig	146109	loss	10.005985
8	1836	Martin Van Buren	Democratic	763291	win	52.272472
9	1836	William Henry Harrison	Whig	550816	loss	37.721543
10	1840	Martin Van Buren	Democratic	1128854	loss	46.948787
11	1840	William Henry Harrison	Whig	1275583	win	53.051213
12	1844	Henry Clay	Whig	1300004	loss	49.250523
13	1844	James Polk	Democratic	1339570	win	50.749477
14	1848	Lewis Cass	Democratic	1223460	loss	42.552229
15	1848	Martin Van Buren	Free Soil	291501	loss	10.138474
16	1848	Zachary Taylor	Whig	1360235	win	47.309296
17	1852	Franklin Pierce	Democratic	1605943	win	51.013168
18	1852	John P. Hale	Free Soil	155210	loss	4.930283
19	1852	Winfield Scott	Whig	1386942	loss	44.056548
20	1856	James Buchanan	Democratic	1835140	win	45.306080
21	1856	John C. Frémont	Republican	1342345	loss	33.139919
22	1856	Millard Fillmore	American	873053	loss	21.554001
23	1860	Abraham Lincoln	Republican	1855993	win	39.699408
24	1860	John Bell	Constitutional Union	590901	loss	12.639283
25	1860	John C. Breckinridge	Southern Democratic	848019	loss	18.138998
26	1860	Stephen A. Douglas	Northern Democratic	1380202	loss	29.522311
27	1864	Abraham Lincoln	National Union	2211317	win	54.951512
28	1864	George B. McClellan	Democratic	1812807	loss	45.048488
29	1868	Horatio Seymour	Democratic	2708744	loss	47.334695
30	1868	Ulysses Grant	Republican	3013790	win	52.665305
31	1872	Horace Greeley	Liberal Republican	2834761	loss	44.071406
32	1872	Ulysses Grant	Republican	3597439	win	55.928594
33	1876	Rutherford Hayes	Republican	4034142	win	48.471624
34	1876	Samuel J. Tilden	Democratic	4288546	loss	51.528376
35	1880	James B. Weaver	Greenback	308649	loss	3.352344
36	1880	James Garfield	Republican	4453337	win	48.369234
37	1880	Winfield Scott Hancock	Democratic	4444976	loss	48.278422
38	1884	Benjamin Butler	Anti-Monopoly	134294	loss	1.335838
39	1884	Grover Cleveland	Democratic	4914482	win	48.884933
40	1884	James G. Blaine	Republican	4856905	loss	48.312208
41	1884	John St. John	Prohibition	147482	loss	1.467021
42	1888	Alson Streeter	Union Labor	146602	loss	1.288861
43	1888	Benjamin Harrison	Republican	5443633	win	47.858041
44	1888	Clinton B. Fisk	Prohibition	249819	loss	2.196299
45	1888	Grover Cleveland	Democratic	5534488	loss	48.656799
46	1892	Benjamin Harrison	Republican	5176108	loss	42.984101
47	1892	Grover Cleveland	Democratic	5553898	win	46.121393
48	1892	James B. Weaver	Populist	1041028	loss	8.645038
49	1892	John Bidwell	Prohibition	270879	loss	2.249468
50	1896	John M. Palmer	National Democratic	134645	loss	0.969566
51	1896	Joshua Levering	Prohibition	131312	loss	0.945565

This code stores our `DataFrame` object in the `elections` variable. Upon inspection, our `elections` `DataFrame` has 182 rows and 6 columns (`Year`, `Candidate`, `Party`, `Popular Vote`, `Result`, `%`). Each row represents a single record — in our example, a presidential candidate from some particular year. Each column represents a single attribute or feature of the record.

#### 2.2.2.1.2 Using a List and Column Name(s)

We'll now explore creating a `DataFrame` with data of our own.

Consider the following examples. The first code cell creates a `DataFrame` with a single column `Numbers`. The second creates a `DataFrame` with the columns `Numbers` and `Description`. Notice how a 2D list of values is required to initialize the second `DataFrame` — each nested list represents a single row of data.

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `Series`

Numbers	
0	1
1	2
2	3

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `Series`

	Number	Description
0	1	one
1	2	two

### 2.2.2.1.3 From a Dictionary

A third (and more common) way to create a `DataFrame` is with a dictionary. The dictionary keys represent the column names, and the dictionary values represent the column values.

Below are two ways of implementing this approach. The first is based on specifying the columns of the `DataFrame`, whereas the second is based on specifying the rows of the `DataFrame`.

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `Series`

	Fruit	Price
0	Strawberry	5.49
1	Orange	3.99

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `Series`

	Fruit	Price
0	Strawberry	5.49
1	Orange	3.99

### 2.2.2.1.4 From a Series

Earlier, we explained how a `Series` was synonymous to a column in a `DataFrame`. It follows, then, that a `DataFrame` is equivalent to a collection of `Series`, which all share the same `Index`.

In fact, we can initialize a `DataFrame` by merging two or more `Series`.

```
# Notice how our indices, or row labels, are the same
```

```
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})
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to\_latex` is expected to utilise the base implementation of `Series.to\_latex`

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

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to\_latex` is expected to utilise the base implementation of `Series.to\_latex`

	0
r1	a1
r2	a2
r3	a3

```
s_a.to_frame()
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to\_latex` is expected to utilise the base implementation of `Series.to\_latex`

	0
r1	a1
r2	a2
r3	a3

### 2.2.3 Indices

On a more technical note, an `Index` doesn't have to be an integer, nor does it have to be unique. For example, we can set the index of the `elections` Dataframe to be the name of presidential candidates.

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of ``S`

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
Henry Clay	1832	National Republican	484205	loss	37.603628
William Wirt	1832	Anti-Masonic	100715	loss	7.821583
Hugh Lawson White	1836	Whig	146109	loss	10.005985
Martin Van Buren	1836	Democratic	763291	win	52.272472
William Henry Harrison	1836	Whig	550816	loss	37.721543
Martin Van Buren	1840	Democratic	1128854	loss	46.948787
William Henry Harrison	1840	Whig	1275583	win	53.051213
Henry Clay	1844	Whig	1300004	loss	49.250523
James Polk	1844	Democratic	1339570	win	50.749477
Lewis Cass	1848	Democratic	1223460	loss	42.552229
Martin Van Buren	1848	Free Soil	291501	loss	10.138474
Zachary Taylor	1848	Whig	1360235	win	47.309296
Franklin Pierce	1852	Democratic	1605943	win	51.013168
John P. Hale	1852	Free Soil	155210	loss	4.930283
Winfield Scott	1852	Whig	1386942	loss	44.056548
James Buchanan	1856	Democratic	1835140	win	45.306080
John C. Frémont	1856	Republican	1342345	loss	33.139919
Millard Fillmore	1856	American	873053	loss	21.554001
Abraham Lincoln	1860	Republican	1855993	win	39.699408
John Bell	1860	Constitutional Union	590901	loss	12.639283
John C. Breckinridge	1860	Southern Democratic	848019	loss	18.138998
Stephen A. Douglas	1860	Northern Democratic	1380202	loss	29.522311
Abraham Lincoln	1864	National Union	2211317	win	54.951512
George B. McClellan	1864	Democratic	1812807	loss	45.048488
Horatio Seymour	1868	Democratic	2708744	loss	47.334695
Ulysses Grant	1868	Republican	3013790	win	52.665305
Horace Greeley	1872	Liberal Republican	2834761	loss	44.071406
Ulysses Grant	1872	Republican	3597439	win	55.928594
Rutherford Hayes	1876	Republican	4034142	win	48.471624
Samuel J. Tilden	1876	Democratic	4288546	loss	51.528376
James B. Weaver	1880	Greenback	308649	loss	3.352344
James Garfield	1880	Republican	4453337	win	48.369234
Winfield Scott Hancock	1880	Democratic	4444976	loss	48.278422
Benjamin Butler	1884	Anti-Monopoly	134294	loss	1.335838
Grover Cleveland	1884	Democratic	4914482	win	48.884933
James G. Blaine	1884	Republican	4856905	loss	48.312208
John St. John	1884	Prohibition	147482	loss	1.467021
Alson Streeter	1888	Union Labor	146602	loss	1.288861
Benjamin Harrison	1888	Republican <sup>22</sup>	5443633	win	47.858041
Clinton B. Fisk	1888	Prohibition	249819	loss	2.196299
Grover Cleveland	1888	Democratic	5534488	loss	48.656799
Benjamin Harrison	1892	Republican	5176108	loss	42.984101
Grover Cleveland	1892	Democratic	5553898	win	46.121393
James B. Weaver	1892	Populist	1041028	loss	8.645038
John Bidwell	1892	Prohibition	270879	loss	2.249468
John M. Palmer	1896	National Democratic	124645	loss	0.969566

We can also select a new column and set it as the index of the DataFrame. For example, we can set the index of the `elections` Dataframe to represent the candidate's party.

```
elections.reset_index(inplace = True) # Resetting the index so we can set the Index again
# This sets the index to the "Party" column
elections.set_index("Party")
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex`` is expected to utilise the base implementation of ``S`

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
National Republican	Henry Clay	1832	484205	loss	37.603628
Anti-Masonic	William Wirt	1832	100715	loss	7.821583
Whig	Hugh Lawson White	1836	146109	loss	10.005985
Democratic	Martin Van Buren	1836	763291	win	52.272472
Whig	William Henry Harrison	1836	550816	loss	37.721543
Democratic	Martin Van Buren	1840	1128854	loss	46.948787
Whig	William Henry Harrison	1840	1275583	win	53.051213
Whig	Henry Clay	1844	1300004	loss	49.250523
Democratic	James Polk	1844	1339570	win	50.749477
Democratic	Lewis Cass	1848	1223460	loss	42.552229
Free Soil	Martin Van Buren	1848	291501	loss	10.138474
Whig	Zachary Taylor	1848	1360235	win	47.309296
Democratic	Franklin Pierce	1852	1605943	win	51.013168
Free Soil	John P. Hale	1852	155210	loss	4.930283
Whig	Winfield Scott	1852	1386942	loss	44.056548
Democratic	James Buchanan	1856	1835140	win	45.306080
Republican	John C. Frémont	1856	1342345	loss	33.139919
American	Millard Fillmore	1856	873053	loss	21.554001
Republican	Abraham Lincoln	1860	1855993	win	39.699408
Constitutional Union	John Bell	1860	590901	loss	12.639283
Southern Democratic	John C. Breckinridge	1860	848019	loss	18.138998
Northern Democratic	Stephen A. Douglas	1860	1380202	loss	29.522311
National Union	Abraham Lincoln	1864	2211317	win	54.951512
Democratic	George B. McClellan	1864	1812807	loss	45.048488
Democratic	Horatio Seymour	1868	2708744	loss	47.334695
Republican	Ulysses Grant	1868	3013790	win	52.665305
Liberal Republican	Horace Greeley	1872	2834761	loss	44.071406
Republican	Ulysses Grant	1872	3597439	win	55.928594
Republican	Rutherford Hayes	1876	4034142	win	48.471624
Democratic	Samuel J. Tilden	1876	4288546	loss	51.528376
Greenback	James B. Weaver	1880	308649	loss	3.352344
Republican	James Garfield	1880	4453337	win	48.369234
Democratic	Winfield Scott Hancock	1880	4444976	loss	48.278422
Anti-Monopoly	Benjamin Butler	1884	134294	loss	1.335838
Democratic	Grover Cleveland	1884	4914482	win	48.884933
Republican	James G. Blaine	1884	4856905	loss	48.312208
Prohibition	John St. John	1884	147482	loss	1.467021
Union Labor	Alson Streeter	1888	146602	loss	1.288861
Republican	Benjamin Harrison <sup>24</sup>	1888	5443633	win	47.858041
Prohibition	Clinton B. Fisk	1888	249819	loss	2.196299
Democratic	Grover Cleveland	1888	5534488	loss	48.656799
Republican	Benjamin Harrison	1892	5176108	loss	42.984101
Democratic	Grover Cleveland	1892	5553898	win	46.121393
Populist	James B. Weaver	1892	1041028	loss	8.645038
Prohibition	John Bidwell	1892	270879	loss	2.249468
National Democratic	John M. Palmer	1896	124645	loss	0.969566



And, if we'd like, we can revert the index back to the default list of integers.

```
# This resets the index to be the default list of integer
elections.reset_index(inplace=True)
elections.index
```

RangeIndex(start=0, stop=182, step=1)

It is also important to note that the row labels that constitute an index don't have to be unique. While index values can be unique and numeric, acting as a row number, they can also be 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

Here we see unique and numeric index values.

	Candidate	Party	%	Result
Year				
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

However, here the index values here are non-unique.

## 2.3 DataFrame Attributes: Index, Columns, and Shape

On the other hand, column names in a `DataFrame` are almost always unique. Looking back to the `elections` dataset, it wouldn't make sense to have two columns named "Candidate".

Sometimes, you'll want to extract these different values, in particular, the list of row and column labels.

For index/row labels, use `DataFrame.index`:

```
elections.set_index("Party", inplace = True)
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(['index', 'Candidate', 'Year', 'Popular vote', 'Result', '%'], dtype='object')
```

And for the shape of the `DataFrame`, we can use `DataFrame.shape`:

```
elections.shape
```

```
(182, 6)
```

## 2.4 Slicing in DataFrames

Now that we've learned more about `DataFrames`, let's dive deeper into their capabilities.

The API (Application Programming Interface) for the `DataFrame` class is enormous. In this section, we'll discuss several methods of the `DataFrame` API that allow us to extract subsets of data.

The simplest way to manipulate a `DataFrame` is to extract a subset of rows and columns, known as **slicing**.

Common ways we may want to extract data are grabbing:

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

We will do so with four primary methods of the `DataFrame` class:

1. `.head` and `.tail`
2. `.loc`
3. `.iloc`
4. `[]`

### 2.4.1 Extracting data with `.head` and `.tail`

The simplest scenario in which we want to extract data is when we simply want to select the first or last few rows of the `DataFrame`.

To extract the first `n` rows of a `DataFrame` `df`, we use the syntax `df.head(n)`.

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

# Extract the first 5 rows of the DataFrame
elections.head(5)
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of ``S`

	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

Similarly, calling `df.tail(n)` allows us to extract the last `n` rows of the DataFrame.

```
# Extract the last 5 rows of the DataFrame
elections.tail(5)
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of ``S`

	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

## 2.4.2 Label-based Extraction: Indexing with `.loc`

For the more complex task of extracting data with specific column or index labels, we can use `.loc`. The `.loc` accessor allows us to specify the **labels** of rows and columns we wish to extract. The **labels** (commonly referred to as the **indices**) are the bold text on the far *left* of a DataFrame, while the **column labels** are the column names found at the *top* of a DataFrame.

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

Row labels

Column labels

To grab data with `.loc`, we must specify the row and column label(s) where the data exists. The row labels are the first argument to the `.loc` function; the column labels are the second.

Arguments to `.loc` can be:

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

For example, to select a single value, we can select the row labeled 0 and the column labeled `Candidate` from the `elections` `DataFrame`.

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

```
'Andrew Jackson'
```

Keep in mind that passing in just one argument as a single value will produce a `Series`. Below, we've extracted a subset of the `"Popular vote"` column as a `Series`.

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions ``DataFrame.to_latex`` is expected to utilise the base implementation of ``Series.to_latex``

	Popular vote
87	15761254
25	848019
179	74216154

To select *multiple* rows and columns, we can use Python slice notation. Here, we select the rows from labels 0 to 3 and the columns from labels `"Year"` to `"Popular vote"`.

```
elections.loc[0:3, 'Year':'Popular vote']
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions ``DataFrame.to_latex`` is expected to utilise the base implementation of ``Series.to_latex``

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897

Suppose that instead, we want to extract *all* column values for the first four rows in the `elections` DataFrame. The shorthand `:` is useful for this.

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to\_latex` is expected to utilise the base implementation of `Series.to\_latex`

	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

We can use the same shorthand to extract all rows.

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to\_latex` is expected to utilise the base implementation of `Series.to\_latex`

	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
5	1832	Henry Clay	loss
6	1832	William Wirt	loss
7	1836	Hugh Lawson White	loss
8	1836	Martin Van Buren	win
9	1836	William Henry Harrison	loss
10	1840	Martin Van Buren	loss
11	1840	William Henry Harrison	win
12	1844	Henry Clay	loss
13	1844	James Polk	win
14	1848	Lewis Cass	loss
15	1848	Martin Van Buren	loss
16	1848	Zachary Taylor	win
17	1852	Franklin Pierce	win
18	1852	John P. Hale	loss
19	1852	Winfield Scott	loss
20	1856	James Buchanan	win
21	1856	John C. Frémont	loss
22	1856	Millard Fillmore	loss
23	1860	Abraham Lincoln	win
24	1860	John Bell	loss
25	1860	John C. Breckinridge	loss
26	1860	Stephen A. Douglas	loss
27	1864	Abraham Lincoln	win
28	1864	George B. McClellan	loss
29	1868	Horatio Seymour	loss
30	1868	Ulysses Grant	win
31	1872	Horace Greeley	loss
32	1872	Ulysses Grant	win
33	1876	Rutherford Hayes	win
34	1876	Samuel J. Tilden	loss
35	1880	James B. Weaver	loss
36	1880	James Garfield	win
37	1880	Winfield Scott Hancock	loss
38	1884	Benjamin Butler	loss
39	1884	Grover Cleveland	win
40	1884	James G. Blaine	loss
41	1884	John St. John	loss
42	1888	Alson Streeter	loss
43	1888	Benjamin Harrison	win
44	1888	Clinton B. Fisk	loss 31
45	1888	Grover Cleveland	loss
46	1892	Benjamin Harrison	loss
47	1892	Grover Cleveland	win
48	1892	James B. Weaver	loss
49	1892	John Bidwell	loss
50	1896	John M. Palmer	loss
51	1896	Joshua Levering	loss

There are a couple of things we should note. Firstly, unlike conventional Python, `pandas` allows us to slice string values (in our example, the column labels). Secondly, slicing with `.loc` is *inclusive*. Notice how our resulting `DataFrame` includes every row and column between and including the slice labels we specified.

Equivalently, we can use a list to obtain multiple rows and columns in our `elections` `DataFrame`.

```
elections.loc[[0, 1, 2, 3], ['Year', 'Candidate', 'Party', 'Popular vote']]
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to\_latex` is expected to utilise the base implementation of `Series.to\_latex`

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897

Lastly, we can interchange list and slicing notation.

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to\_latex` is expected to utilise the base implementation of `Series.to\_latex`

	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

### 2.4.3 Integer-based Extraction: Indexing with `.iloc`

Slicing with `.iloc` works similarly to `.loc`. However, `.iloc` uses the *index positions* of rows and columns rather than the labels (think to yourself: `loc` uses `lables`; `iloc` uses `indices`). The



arguments to the `.iloc` function also behave similarly — single values, lists, indices, and any combination of these are permitted.

Let's begin reproducing our results from above. We'll begin by selecting the first presidential candidate in our `elections` DataFrame:

```
# elections.loc[0, "Candidate"] - Previous approach
elections.iloc[0, 1]
```

```
'Andrew Jackson'
```

Notice how the first argument to both `.loc` and `.iloc` are the same. This is because the row with a label of 0 is conveniently in the 0<sup>th</sup> (equivalently, the first position) of the `elections` DataFrame. Generally, this is true of any DataFrame where the row labels are incremented in ascending order from 0.

And, as before, if we were to pass in only one single value argument, our result would be a `Series`.

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

	Candidate
1	John Quincy Adams
2	Andrew Jackson
3	John Quincy Adams

However, when we select the first four rows and columns using `.iloc`, we notice something.

```
# elections.loc[0:3, 'Year':'Popular vote'] - Previous approach
elections.iloc[0:4, 0:4]
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex`` is expected to utilise the base implementation of ``S`

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897

Slicing is no longer inclusive in `.iloc` — it's *exclusive*. In other words, the right end of a slice is not included when using `.iloc`. This is one of the subtleties of `pandas` syntax; you will get used to it with practice.

List behavior works just as expected.

```
#elections.loc[[0, 1, 2, 3], ['Year', 'Candidate', 'Party', 'Popular vote']] - Previous Ap  
elections.iloc[[0, 1, 2, 3], [0, 1, 2, 3]]
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of ``S`

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897

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

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

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of ``S`

	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
5	1832	Henry Clay	National Republican
6	1832	William Wirt	Anti-Masonic
7	1836	Hugh Lawson White	Whig
8	1836	Martin Van Buren	Democratic
9	1836	William Henry Harrison	Whig
10	1840	Martin Van Buren	Democratic
11	1840	William Henry Harrison	Whig
12	1844	Henry Clay	Whig
13	1844	James Polk	Democratic
14	1848	Lewis Cass	Democratic
15	1848	Martin Van Buren	Free Soil
16	1848	Zachary Taylor	Whig
17	1852	Franklin Pierce	Democratic
18	1852	John P. Hale	Free Soil
19	1852	Winfield Scott	Whig
20	1856	James Buchanan	Democratic
21	1856	John C. Frémont	Republican
22	1856	Millard Fillmore	American
23	1860	Abraham Lincoln	Republican
24	1860	John Bell	Constitutional Union
25	1860	John C. Breckinridge	Southern Democratic
26	1860	Stephen A. Douglas	Northern Democratic
27	1864	Abraham Lincoln	National Union
28	1864	George B. McClellan	Democratic
29	1868	Horatio Seymour	Democratic
30	1868	Ulysses Grant	Republican
31	1872	Horace Greeley	Liberal Republican
32	1872	Ulysses Grant	Republican
33	1876	Rutherford Hayes	Republican
34	1876	Samuel J. Tilden	Democratic
35	1880	James B. Weaver	Greenback
36	1880	James Garfield	Republican
37	1880	Winfield Scott Hancock	Democratic
38	1884	Benjamin Butler	Anti-Monopoly
39	1884	Grover Cleveland	Democratic
40	1884	James G. Blaine	Republican
41	1884	John St. John	Prohibition
42	1888	Alson Streeter	Union Labor
43	1888	Benjamin Harrison	Republican
44	1888	Clinton B. Fisk	Prohibition
45	1888	Grover Cleveland	Democratic
46	1892	Benjamin Harrison	Republican
47	1892	Grover Cleveland	Democratic
48	1892	James B. Weaver	Populist
49	1892	John Bidwell	Prohibition
50	1896	John M. Palmer	National Democratic
51	1896	Joshua Levering	Prohibition

This discussion begs the question: When should we use `.loc` vs. `.iloc`? In most cases, `.loc` is generally safer to use. You can imagine `.iloc` may return incorrect values when applied to a dataset where the ordering of data can change. However, `.iloc` can still be useful — for example, if you are looking at a `DataFrame` of sorted movie earnings and want to get the median earnings for a given year, you can use `.iloc` to index into the middle.

Overall, it is important to remember that:

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

## 2.4.4 Context-dependent Extraction: Indexing with `[]`

The `[]` selection operator is the most baffling of all, yet the most commonly used. It only takes a single argument, which may be one of the following:

1. A slice of row numbers.
2. A list of column labels.
3. A single-column label.

That is, `[]` is *context-dependent*. Let's see some examples.

### 2.4.4.1 A slice of row numbers

Say we wanted the first four rows of our `elections` `DataFrame`.

```
elections[0:4]
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to_latex`` is expected to utilise the base implementation of ``S`

	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

#### 2.4.4.2 A list of column labels

Suppose we now want the first four columns.

```
elections[["Year", "Candidate", "Party", "Popular vote"]]
```

/Users/Ishani/micromamba/lib/python3.9/site-packages/IPython/core/formatters.py:342: FutureWarning

In future versions `DataFrame.to\_latex` is expected to utilise the base implementation of `Series.to\_latex`

	Year	Candidate	Party	Popular vote
0	1824	Andrew Jackson	Democratic-Republican	151271
1	1824	John Quincy Adams	Democratic-Republican	113142
2	1828	Andrew Jackson	Democratic	642806
3	1828	John Quincy Adams	National Republican	500897
4	1832	Andrew Jackson	Democratic	702735
5	1832	Henry Clay	National Republican	484205
6	1832	William Wirt	Anti-Masonic	100715
7	1836	Hugh Lawson White	Whig	146109
8	1836	Martin Van Buren	Democratic	763291
9	1836	William Henry Harrison	Whig	550816
10	1840	Martin Van Buren	Democratic	1128854
11	1840	William Henry Harrison	Whig	1275583
12	1844	Henry Clay	Whig	1300004
13	1844	James Polk	Democratic	1339570
14	1848	Lewis Cass	Democratic	1223460
15	1848	Martin Van Buren	Free Soil	291501
16	1848	Zachary Taylor	Whig	1360235
17	1852	Franklin Pierce	Democratic	1605943
18	1852	John P. Hale	Free Soil	155210
19	1852	Winfield Scott	Whig	1386942
20	1856	James Buchanan	Democratic	1835140
21	1856	John C. Frémont	Republican	1342345
22	1856	Millard Fillmore	American	873053
23	1860	Abraham Lincoln	Republican	1855993
24	1860	John Bell	Constitutional Union	590901
25	1860	John C. Breckinridge	Southern Democratic	848019
26	1860	Stephen A. Douglas	Northern Democratic	1380202
27	1864	Abraham Lincoln	National Union	2211317
28	1864	George B. McClellan	Democratic	1812807
29	1868	Horatio Seymour	Democratic	2708744
30	1868	Ulysses Grant	Republican	3013790
31	1872	Horace Greeley	Liberal Republican	2834761
32	1872	Ulysses Grant	Republican	3597439
33	1876	Rutherford Hayes	Republican	4034142
34	1876	Samuel J. Tilden	Democratic	4288546
35	1880	James B. Weaver	Greenback	308649
36	1880	James Garfield	Republican	4453337
37	1880	Winfield Scott Hancock	Democratic	4444976
38	1884	Benjamin Butler	Anti-Monopoly	134294
39	1884	Grover Cleveland	Democratic	4914482
40	1884	James G. Blaine	Republican	4856905
41	1884	John St. John	Prohibition	147482
42	1888	Alson Streeter	Union Labor	146602
43	1888	Benjamin Harrison	Republican	5443633
44	1888	Clinton B. Fisk	Prohibition	249819
45	1888	Grover Cleveland	Democratic	5534488
46	1892	Benjamin Harrison	Republican	5176108
47	1892	Grover Cleveland	Democratic	5553898
48	1892	James B. Weaver	Populist	1041028
49	1892	John Bidwell	Prohibition	270879
50	1896	John M. Palmer	National Democratic	134645
51	1896	Joshua Levering	Prohibition	131312

#### 2.4.4.3 A single-column label

Lastly, `[]` allows us to extract only the `Candidate` column.

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

	Candidate
0	Andrew Jackson
1	John Quincy Adams
2	Andrew Jackson
3	John Quincy Adams
4	Andrew Jackson
5	Henry Clay
6	William Wirt
7	Hugh Lawson White
8	Martin Van Buren
9	William Henry Harrison
10	Martin Van Buren
11	William Henry Harrison
12	Henry Clay
13	James Polk
14	Lewis Cass
15	Martin Van Buren
16	Zachary Taylor
17	Franklin Pierce
18	John P. Hale
19	Winfield Scott
20	James Buchanan
21	John C. Frémont
22	Millard Fillmore
23	Abraham Lincoln
24	John Bell
25	John C. Breckinridge
26	Stephen A. Douglas
27	Abraham Lincoln
28	George B. McClellan
29	Horatio Seymour
30	Ulysses Grant
31	Horace Greeley
32	Ulysses Grant
33	Rutherford Hayes
34	Samuel J. Tilden
35	James B. Weaver
36	James Garfield
37	Winfield Scott Hancock
38	Benjamin Butler
39	Grover Cleveland
40	James G. Blaine
41	John St. John
42	Alson Streeter
43	Benjamin Harrison
44	Clinton B. Fisk
45	Grover Cleveland
46	Benjamin Harrison
47	Grover Cleveland
48	James B. Weaver
49	John Bidwell
50	John M. Palmer
51	Joshua Levering



The output is a **Series**! In this course, we'll become very comfortable with `[]`, especially for selecting columns. In practice, `[]` is much more common than `.loc`, especially since it is far more concise.

## 2.5 Parting Note

The **pandas** library is enormous and contains many useful functions. Here is a link to [documentation](#). We certainly don't expect you to memorize each and every method of the library.

The introductory Data 100 **pandas** lectures will provide a high-level view of the key data structures and methods that will form the foundation of your **pandas** knowledge. A goal of this course is to help you build your familiarity with the real-world programming practice of ...Googling! Answers to your questions can be found in documentation, Stack Overflow, etc. Being able to search for, read, and implement documentation is an important life skill for any data scientist.

With that, we will move on to Pandas II.