Principles and Techniques of Data Science

Data 100

Kanu Grover   Bella Crouch

Table of contents

Welcome

About the Course Notes

This text was developed for the Spring 2023 Edition of the UC Berkeley course Data 100: Principles and Techniques of Data Science.

As this project is in development during the Spring 2023 semester, the course notes may be in flux. We appreciate your understanding. If you spot any errors or would like to suggest any changes, please email us. **Email**: data100.instructors@berkeley.edu

1 Introduction

> **Note**
>
> - Understand the stages of the data science lifecycle.

Data science is an interdisciplinary field with a variety of applications. The field is rapidly evolving; many of the key technical underpinnings in modern-day data science have been popularized during the early 21^{st} century.

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. To do so, 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 will 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 gives a clear point to know when to finish 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, its 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, 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 to 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 collected 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 reveal these 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 by our findings, or our initial exploration may have brought up new questions that require a 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 untrue 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 list of requirements. In our journey exploring the lifecycle, we'll cover both the underlying theory and technologies used in data science, and we hope you'll build an appreciation for the field.

With that, let's begin by introducing one of the most important tools in exploratory data analysis: `pandas`.

2 Pandas I

> **Note**
>
> - Build familiarity with basic `pandas` syntax
> - Learn the methods of selecting and filtering data from a DataFrame.
> - Understand the differences between DataFrames and Series

Data scientists work with data stored in a variety of formats. The primary focus of this class is in understanding tabular data – one of the most widely used formats in data science. This note introduces DataFrames, which are among the most popular representations of tabular data. We'll also introduce `pandas`, the standard Python package for manipulating data in DataFrames.

2.1 Introduction to Exploratory Data Analysis

Imagine you collected, or have been given a box of data. What do you do next?

The first step is to clean your data. **Data cleaning** often corrects issues in the structure and formatting of data, including missing values and unit conversions.

Data scientists have coined the term **exploratory data analysis (EDA)** to describe the process of transforming raw data to insightful observations. EDA is an *open-ended* analysis of transforming, visualizing, and summarizing patterns in data. In order to conduct EDA, we first need to familiarize ourselves with `pandas` – an important programming tool.

2.2 Introduction to Pandas

`pandas` is a data analysis library to make data cleaning and analysis fast and convenient in Python.

The `pandas` library adopts many coding idioms from `NumPy`. The biggest difference is that `pandas` is designed for working with tabular data, one of the most common data formats (and the focus of Data 100).

Before writing any code, we must import `pandas` into our Python environment.

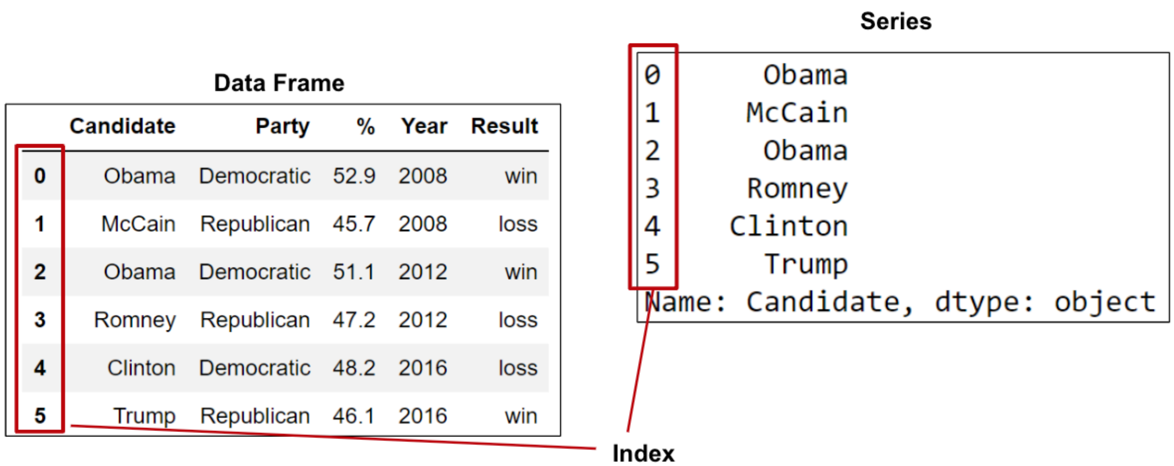
```
# `pd` is the conventional alias for Pandas, as `np` is for NumPy
import pandas as pd
```

2.3 Series, DataFrames, and Indices

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.



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 `Candidate` column. Both contain an **Index**, or a shared list of row labels (the integers from 0 to 5, inclusive).

2.3.1 Series

A Series represents a column of a DataFrame; more generally, it can be any 1-dimensional array-like object containing values of the same type with associated data labels, called its index.

```
import pandas as pd

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

```python
print(s)
```

```
0    -1
1    10
2     2
dtype: int64
```

```python
s.array # Data contained within the Series
```

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

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

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

By default, row indices in `pandas` are a sequential list of integers beginning from 0. Optionally, a list of desired indices can be passed to the `index` argument.

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

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

Indices can also be changed after initialization.

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

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

2.3.1.1 Selection in Series

Similar to an array, we can select a single value or a set of values from a Series. There are 3 primary methods of selecting data.

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

Let's define the following Series `ser`.

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

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

2.3.1.1.1 A Single Index Label

```
print(ser["a"]) # Notice how the return value is a single array element
```

```
4
```

2.3.1.1.2 A List of Index Labels

```
ser[["a", "c"]] # Notice how the return value is another Series
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

	0
a	4
c	0

2.3.1.1.3 A Filtering Condition

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

We first must apply a vectorized boolean operation to our Series that encodes the filter condition.

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

	0
a	True
b	False
c	False
d	True

Upon "indexing" in our Series with this condition, `pandas` selects only the rows with `True` values.

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

	0
a	4
d	6

2.3.2 DataFrames

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.

Here is an example of a DataFrame that contains election data.

```
import pandas as pd

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

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

Let's dissect the code above.

1. We first import the `pandas` library into our Python environment, using the alias `pd`. `import pandas as pd`
2. There are a number of ways to read data into a DataFrame. In Data 100, our data are typically stored in a CSV (comma-seperated 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("elections.csv")`

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 presedential candidate from some particular year. Each column represents a single attribute, or feature of the record.

In the example above, we constructed a DataFrame object using data from a CSV file. As we'll explore in the next section, we can create a DataFrame with data of our own.

2.3.2.1 Creating a DataFrame

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

1. Using a list and column names
2. From a dictionary
3. From a Series

2.3.2.1.1 Using a List and Column Names

Consider the following examples.

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

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

	Numbers
0	1
1	2
2	3

The first code cell creates a DataFrame with a single column `Numbers`, while the second creates a DataFrame with an additional column `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, "one"], [2, "two"]], columns = ["Number", "Description"])
df_list
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

	Number	Description
0	1	one
1	2	two

2.3.2.1.2 From a Dictionary

A second (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.

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

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

	Fruit	Price
0	Strawberry	5.49
1	Orange	3.99

2.3.2.1.3 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.

```python
# 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})
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

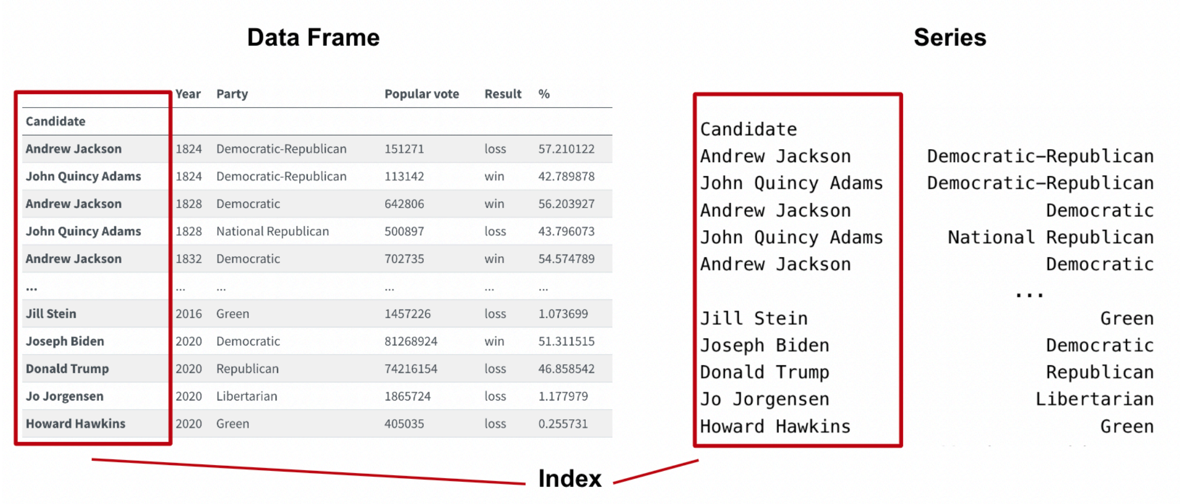
	A-column	B-column
r1	a1	b1
r2	a2	b2
r3	a3	b3

2.3.3 Indices

The major takeaway: we can think of a **DataFrame** as a collection of **Series** that all share the same **Index**.

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 presedential candidates. Selecting a new Series from this modified DataFrame yields the following.

```python
# This sets the index to the "Candidate" column
elections.set_index("Candidate", inplace=True)
```



To retrieve the indices of a DataFrame, simply use the `.index` attribute of the DataFrame class.

```
elections.index
```

```
Index(['Andrew Jackson', 'John Quincy Adams', 'Andrew Jackson',
       'John Quincy Adams', 'Andrew Jackson', 'Henry Clay', 'William Wirt',
       'Hugh Lawson White', 'Martin Van Buren', 'William Henry Harrison',
       ...
       'Darrell Castle', 'Donald Trump', 'Evan McMullin', 'Gary Johnson',
       'Hillary Clinton', 'Jill Stein', 'Joseph Biden', 'Donald Trump',
       'Jo Jorgensen', 'Howard Hawkins'],
      dtype='object', name='Candidate', length=182)
```

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

2.4 Slicing in DataFrames

Now that we've learned how to create 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**. We will do so with three primary methods of the DataFrame class:

1. `.loc`
2. `.iloc`
3. `[]`

2.4.1 Indexing with .loc

The `.loc` operator selects rows and columns in a DataFrame by their row and column label(s), respectively. The **row 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.

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

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

```
'Andrew Jackson'
```

To select *multiple* rows and columns, we can use Python slice notation. Here, we select both the first four rows and columns.

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

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

	Year	Party	Popular vote
0	1824	Democratic-Republican	151271
1	1824	Democratic-Republican	113142
2	1828	Democratic	642806
3	1828	National Republican	500897

Suppose that instead, we wanted *every* column value for the first four rows in the `elections` DataFrame. The shorthand `:` is useful for this.

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

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

	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

There are a couple of things we should note. 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.

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

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

Lastly, we can interchange list and slicing notation.

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

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

	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

2.4.2 Indexing with .iloc

Slicing with `.iloc` works similarly to `.loc`, although `.iloc` uses the integer positions of rows and columns rather the labels. 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 for the first presedential candidate in our `elections` DataFrame:

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

```
1824
```

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^{th} (or first) position of the `elections` DataFrame. Generally, this is true of any DataFrame where the row labels are incremented in ascending order from 0.

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]
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

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

Slicing is no longer inclusive in `.iloc` - it's *exclusive*. This is one of Pandas syntatical subtleties; you'll get used to 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]]
```

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/

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

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

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.

2.4.3 Indexing with []

The `[]` selection operator is the most baffling of all, yet the 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.

2.4.3.1 A slice of row numbers

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

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

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/
```

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

	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

2.4.3.2 A list of column labels

Suppose we now want the first four columns.

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

```
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages/IPython/core/
```

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

	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

2.4.3.3 A single column label

Lastly, if we only want 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 looks like a Series! In this course, we'll become very comfortable with `[]`, especially for selecting columns. In practice, `[]` is much more common than `.loc`.

2.5 Parting Note

The `pandas` library is enormous and contains many useful functions. Here is a link to documentation.

The introductory `pandas` lectures will cover important data structures and methods you should be fluent in. However, we want you to get familiar with the real world programming practice of …Googling! Answers to your questions can be found in documentation, Stack Overflow, etc.

With that, let's move on to Pandas II.