Principles and Techniques of Data Science

Data 100

Kanu Grover Bella Crouch

Table of contents

Welcome **4**
About the Course Notes . . . 4

1 Introduction **5**
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 . . . 8

2 Pandas I **9**
2.1 Introduction to Exploratory Data Analysis . . . 9
2.2 Introduction to Pandas . . . 9
2.3 Series, DataFrames, and Indices . . . 10
2.3.1 Series . . . 10
2.3.2 DataFrames . . . 13
2.3.3 Indices . . . 16
2.4 Slicing in DataFrames . . . 17
2.4.1 Indexing with .loc . . . 17
2.4.2 Indexing with .iloc . . . 19
2.4.3 Indexing with [] . . . 20
2.5 Parting Note . . . 21

3 Pandas II **22**
3.1 Conditional Selection . . . 23
3.2 Handy Utility Functions . . . 27
3.2.1 `Numpy` . . . 27
3.2.2 `.shape` and `.size` . . . 28
3.2.3 `.describe()` . . . 28
3.2.4 `.sample()` . . . 29
3.2.5 `.value_counts()` . . . 30
3.2.6 `.unique()` . . . 30
3.2.7 `.sort_values()` . . . 30
3.3 Adding and Removing Columns . . . 32

3.4 Aggregating Data with GroupBy 34
3.4.1 Parting note 37

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 21st 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.

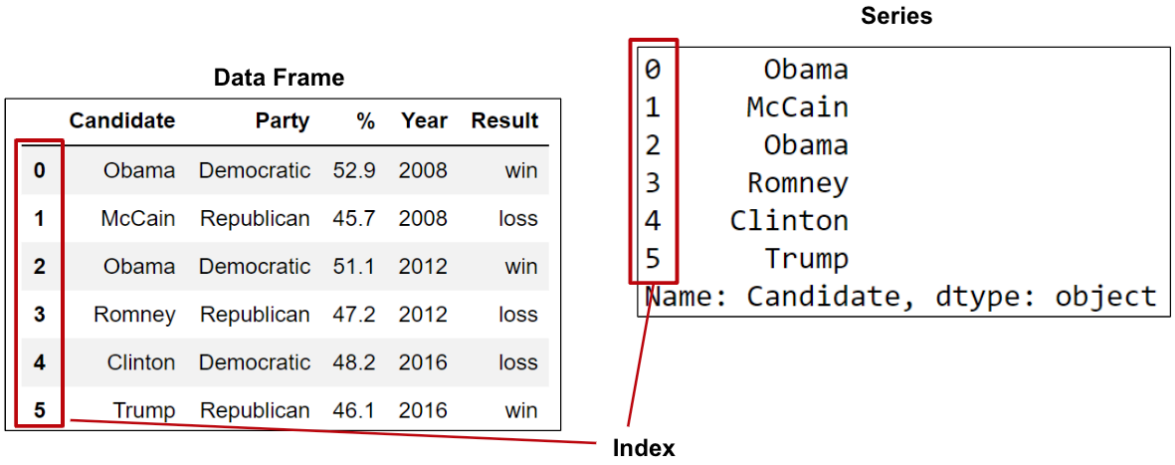
```
# `pd` is the conventional alias for Pandas, as `np` is for NumPy
import pandas as pd
```

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])
```

```
print(s)
```

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

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

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

```
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.

```
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.

```
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
```

	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.head()
```

	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

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
```

	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
```

	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
```

	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.

```
# 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})
```

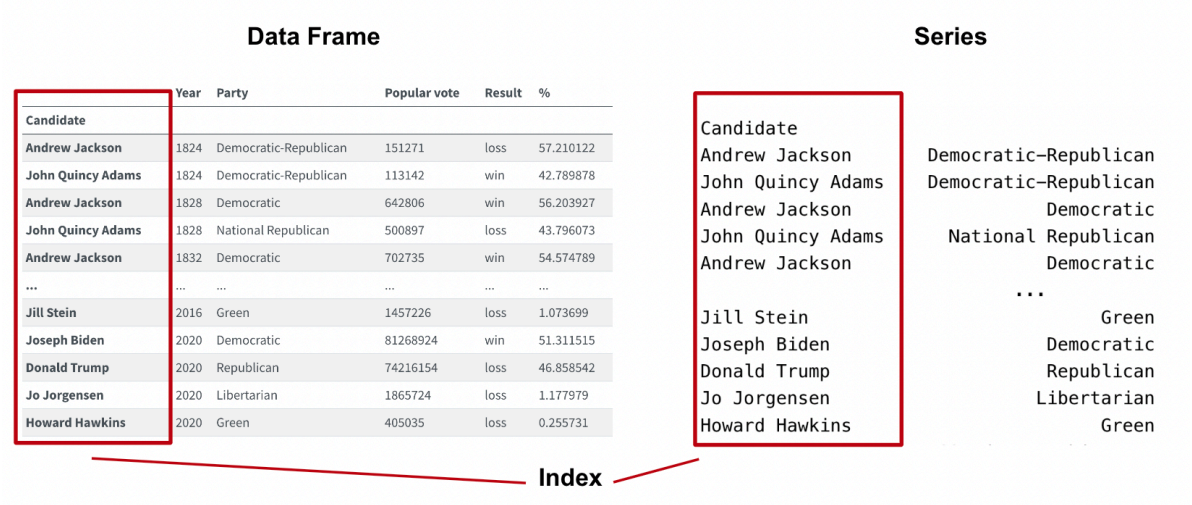
	A-column	B-column
r1	a1	b1
r2	a2	b2
r3	a3	b3

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.

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



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
...	...	...	...	...	...
Jill Stein	2016	Green	1457226	loss	1.073699
Joseph Biden	2020	Democratic	81268924	win	51.311515
Donald Trump	2020	Republican	74216154	loss	46.858542
Jo Jorgensen	2020	Libertarian	1865724	loss	1.177979
Howard Hawkins	2020	Green	405035	loss	0.255731

```
Candidate
Andrew Jackson          Democratic-Republican
John Quincy Adams       Democratic-Republican
Andrew Jackson                     Democratic
John Quincy Adams         National Republican
Andrew Jackson                     Democratic
                                  ...
Jill Stein                                Green
Joseph Biden                       Democratic
Donald Trump                       Republican
Jo Jorgensen                      Libertarian
Howard Hawkins                            Green
```

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

```
elections.head().index
```

```
Index(['Andrew Jackson', 'John Quincy Adams', 'Andrew Jackson',
```

```
 'John Quincy Adams', 'Andrew Jackson'],
dtype='object', name='Candidate')
```

```
# 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']
```

	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.

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

	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.

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

	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], :]
```

	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 similarily 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]
```

	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]]
```

	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]
```

	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"]].head()
```

	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

2.4.3.3 A single column label

Lastly, if we only want the `Candidate` column.

```
elections["Candidate"].head()
```

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

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.

3 Pandas II

> **Note**
>
> - Build familiarity with advanced `pandas` syntax
> - Extract data from a DataFrame using conditional selection
> - Recognize situations where aggregation is useful and identify the correct technique for performing an aggregation

Last time, we introduced the `pandas` library as a toolkit for processing data. We learned the DataFrame and Series data structures, familiarized ourselves with the basic syntax for manipulating tabular data, and began writing our first lines of `pandas` code.

In this lecture, we'll start to dive into some advanced `pandas` syntax. You may find it helpful to follow along with a notebook of your own as we walk through these new pieces of code.

We'll start by loading the `babynames` dataset.

```
import pandas as pd
import numpy as np
import urllib.request
import os.path
import zipfile

data_url = "https://www.ssa.gov/oact/babynames/state/namesbystate.zip"
local_filename = "babynamesbystate.zip"
if not os.path.exists(local_filename): # if the data exists don't download again
    with urllib.request.urlopen(data_url) as resp, open(local_filename, 'wb') as f:
        f.write(resp.read())

zf = zipfile.ZipFile(local_filename, 'r')

ca_name = 'CA.TXT'
field_names = ['State', 'Sex', 'Year', 'Name', 'Count']
with zf.open(ca_name) as fh:
    babynames = pd.read_csv(fh, header=None, names=field_names)
```

```
babynames.head()
```

	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

3.1 Conditional Selection

Conditional selection allows us to select a subset of rows in a DataFrame if they follow some specified condition.

To understand how to use conditional selection, we must look at another possible input of the `.loc` and `[]` methods – a boolean array, which is simply an array where each element is either `True` or `False`. This boolean array must have a length equal to the number of rows in the DataFrame. It will return all rows in the position of a corresponding True value in the array.

To see this in action, let's select all even-indexed rows in the first 10 rows of our DataFrame.

```
# Ask yourself: why is :9 is the correct slice to select the first 10 rows?
babynames_first_10_rows = babynames.loc[:9, :]

# Notice how we have exactly 10 elements in our boolean array argument
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

We can perform a similar operation using `.loc`.

```
babynames_first_10_rows.loc[[True, False, True, False, True, False, True, False, True, Fal
```

	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

These techniques worked well in this example, but you can imagine how tedious it might be to list out `True`s and `False`s for every row in a larger DataFrame. To make things easier, we can instead provide a logical condition as an input to `.loc` or `[]` that returns a boolean array with the necessary length.

For example, to return all names associated with `M` sex:

```
# First, use a logical condition to generate a boolean array
logical_operator = (babynames["Sex"] == "M")

# Then, use this boolean array to filter the DataFrame
babynames[logical_operator].head()
```

	State	Sex	Year	Name	Count
235791	CA	M	1910	John	237
235792	CA	M	1910	William	170
235793	CA	M	1910	James	159
235794	CA	M	1910	Robert	141
235795	CA	M	1910	George	138

Here, `logical_operator` evaluates to a Series of boolean values with length 400762, of which 235791 are `False`.

```
logical_operator.describe()
```

	Sex
count	400762
unique	2
top	False
freq	235791

Rows starting at row 235791 and ending at row 400761 evaluate to `True` and are thus returned in the DataFrame.

```
babynames[logical_operator].tail()
```

	State	Sex	Year	Name	Count
400757	CA	M	2021	Zyan	5
400758	CA	M	2021	Zyion	5
400759	CA	M	2021	Zyire	5
400760	CA	M	2021	Zylo	5
400761	CA	M	2021	Zyrus	5

Passing a Series as an argument to `babynames[]` has the same affect as using a boolean array. In fact, the `[]` selection operator can take a boolean Series, array, and list as arguments. These three are used interchangeably thoughout the course.

We can also use `.loc` to achieve similar results.

```
babynames.loc[babynames["Sex"] == "F"].head()
```

	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

Boolean conditions can be combined using various operators that allow us to filter results by multiple conditions. Some examples include the `&` (and) operator and the `|` (or) operator.

Note: When combining multiple conditions with logical operators, be sure to surround each condition with a set of parenthesis `()`. If you forget, your code will throw an error.

For example, if we want to return data on all females born in the 21st century, we can write:

```
babynames[(babynames["Sex"] == "F") & (babynames["Year"] > 2000)].head()
```

	State	Sex	Year	Name	Count
152816	CA	F	2001	Emily	2928
152817	CA	F	2001	Ashley	2717
152818	CA	F	2001	Samantha	2535
152819	CA	F	2001	Jessica	2244
152820	CA	F	2001	Alyssa	2059

Boolean array selection is a useful tool, but can lead to overly verbose code for complex conditions. `Pandas` provide many alternatives:

```
(
    babynames[(babynames["Name"] == "Bella") |
              (babynames["Name"] == "Alex") |
              (babynames["Name"] == "Ani") |
              (babynames["Name"] == "Lisa")]
).head()
# Note: The parentheses surrounding the code make it possible to break the code on to mult
```

	State	Sex	Year	Name	Count
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

The `.isin` function can be used to filter dataframes. The method helps in selecting rows with having a particular (or multiple) value in a particular column.

```
names = ["Bella", "Alex", "Ani", "Lisa"]
babynames[babynames["Name"].isin(names)].head()
```

	State	Sex	Year	Name	Count
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

The function `str.startswith` can be used to define a filter based on string values in a `Series` object.

```
babynames[babynames["Name"].str.startswith("N")].head()
```

	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

3.2 Handy Utility Functions

`pandas` contains an extensive library of functions that can help shorten the process of setting and getting information from its data structures. In the following section, we will give overviews of each of the main utility functions that will help us in Data 100.

- `Numpy` and built-in function support
- `.shape`
- `.size`
- `.describe()`
- `.sample()`
- `.value_counts()`
- `.unique()`
- `.sort_values()`

3.2.1 `Numpy`

```
bella_counts = babynames[babynames["Name"] == "Bella"]["Count"]
```

```
# Average number of babies named Bella each year
np.mean(bella_counts)
```

```
270.1860465116279
```

```
# Max number of babies named Bella born on a given year
max(bella_counts)
```

```
902
```

3.2.2 `.shape` and `.size`

`.shape` and `.size` are attributes of Series and DataFrames that measure the "amount" of data stored in the structure. Calling `.shape` returns a tuple containing the number of rows and columns present in the DataFrame or Series. `.size` is used to find the total number of elements in a structure, equivalent to the number of rows times the number of columns.

Many functions strictly require the dimensions of the arguments along certain axes to match. Calling these dimension-finding functions is much faster than counting all of the items by hand.

```
babynames.shape
```

```
(400762, 5)
```

```
babynames.size
```

```
2003810
```

3.2.3 `.describe()`

If many statistics are required from a DataFrame (minimum value, maximum value, mean value, etc.), then `.describe()` can be used to compute all of them at once.

```
babynames.describe()
```

	Year	Count
count	400762.000000	400762.000000
mean	1985.131287	79.953781
std	26.821004	295.414618
min	1910.000000	5.000000
25%	1968.000000	7.000000
50%	1991.000000	13.000000
75%	2007.000000	38.000000
max	2021.000000	8262.000000

A different set of statistics will be reported if `.describe()` is called on a Series.

```
babynames["Sex"].describe()
```

	Sex
count	400762
unique	2
top	F
freq	235791

3.2.4 `.sample()`

As we will see later in the semester, random processes are at the heart of many data science techniques (for example, train-test splits, bootstrapping, and cross-validation). `.sample()` lets us quickly select random entries (a row if called from a DataFrame, or a value if called from a Series).

```
babynames.sample()
```

	State	Sex	Year	Name	Count
270315	CA	M	1962	Burton	24

```
babynames.sample(5).iloc[:, 2:]
```

	Year	Name	Count
178530	2007	Tegan	15
12342	1932	Emiko	5
374411	2012	Arjen	5
14531	1936	Patty	25
263340	1956	Victor	464

```
babynames[babynames["Year"] == 2000].sample(4, replace = True).iloc[:, 2:]
```

	Year	Name	Count
150838	2000	Cloe	12
338972	2000	Osvaldo	139
340848	2000	Yael	6
339245	2000	Damion	43

3.2.5 `.value_counts()`

When we want to know the distribution of the items in a Series (for example, what items are most/least common), we use `.value-counts()` to get a breakdown of the unique *values* and their *counts*. In the example below, we can determine the name with the most years in which at least one person has taken that name by counting the number of times each name appears in the `"Name"` column of `babynames`.

```
babynames["Name"].head().value_counts()
```

	Name
Margaret	1
Mary	1
Dorothy	1
Helen	1
Frances	1

3.2.6 `.unique()`

If we have a Series with many repeated values, then `.unique()` can be used to identify only the *unique* values. Here we can get a list of all the names in `babynames`.

Exercise: what function can we call on the Series below to get the number of unique names?

```
babynames["Name"].unique()[:5]
```

```
array(['Mary', 'Helen', 'Dorothy', 'Margaret', 'Frances'], dtype=object)
```

3.2.7 `.sort_values()`

Ordering a DataFrame can be useful for isolating extreme values. For example, the first 5 entries of a row sorted in descending order (that is, from highest to lowest) are the largest 5 values. `.sort_values` allows us to order a DataFrame or Series by a specified rule. For DataFrames, we must specify the column by which we want to compare the rows and the function will return such rows. We can choose to either receive the rows in `ascending` order (default) or `descending` order.

```
babynames.sort_values(by = "Count", ascending=False).head()
```

	State	Sex	Year	Name	Count
263272	CA	M	1956	Michael	8262
264297	CA	M	1957	Michael	8250
313644	CA	M	1990	Michael	8247
278109	CA	M	1969	Michael	8244
279405	CA	M	1970	Michael	8197

We do not need to explicitly specify the column used for sorting when calling `.value_counts()` on a Series. We can still specify the ordering paradigm – that is, whether values are sorted in ascending or descending order.

```
babynames["Name"].sort_values(ascending=True).head()
```

	Name
380256	Aadan
362255	Aadan
365374	Aadan
394460	Aadarsh
366561	Aaden

3.2.7.1 Sorting With a Custom Key

Using `.sort_values` can be useful in many situations, but it many not cover all use cases. This is because `pandas` automatically sorts values in order according to numeric value (for number data) or alphabetical order (for string data). The following code finds the top 5 most popular names in California in 2021.

```
# Sort names by count in year 2021
# Recall that `.head(5)` displays the first five rows in the DataFrame
babynames[babynames["Year"] == 2021].sort_values("Count", ascending=False).head()
```

	State	Sex	Year	Name	Count
397909	CA	M	2021	Noah	2591
397910	CA	M	2021	Liam	2469
232145	CA	F	2021	Olivia	2395
232146	CA	F	2021	Emma	2171
397911	CA	M	2021	Mateo	2108

This offers us a lot of functionality, but what if we need to sort by some other metric? For example, what if we wanted to find the longest names in the DataFrame?

We can do this by specifying the `key` parameter of `.sort_values`. The `key` parameter is assigned to a function of our choice. This function is then applied to each value in the specified column. `pandas` will, finally, sort the DataFrame by the values outputted by the function.

```
# Here, a lambda function is applied to find the length of each value, `x`, in the "Name"
babynames.sort_values("Name", key=lambda x: x.str.len(), ascending=False).head(5)
```

	State	Sex	Year	Name	Count
313143	CA	M	1989	Franciscojavier	6
333732	CA	M	1997	Ryanchristopher	5
330421	CA	M	1996	Franciscojavier	8
323615	CA	M	1993	Johnchristopher	5
310235	CA	M	1988	Franciscojavier	10

3.3 Adding and Removing Columns

To add a new column to a DataFrame, we use a syntax similar to that used when accessing an existing column. Specify the name of the new column by writing `dataframe["new_column"]`, then assign this to a Series or Array containing the values that will populate this column.

```
# Add a column named "name_lengths" that includes the length of each name
babynames["name_lengths"] = babynames["Name"].str.len()
babynames.head(5)
```

	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

```
# Sort by the temporary column
babynames = babynames.sort_values(by = "name_lengths", ascending=False)
babynames.head()
```

	State	Sex	Year	Name	Count	name_lengths
313143	CA	M	1989	Franciscojavier	6	15
333732	CA	M	1997	Ryanchristopher	5	15
330421	CA	M	1996	Franciscojavier	8	15
323615	CA	M	1993	Johnchristopher	5	15
310235	CA	M	1988	Franciscojavier	10	15

In the example above, we made use of an in-built function given to us by the `str` accessor for getting the length of names. Then we used `name_length` column to sort the dataframe. What if we had wanted to generate the values in our new column using a function of our own making?

We can do this using the Series `.map` method. `.map` takes in a function as input, and will apply this function to each value of a Series.

For example, say we wanted to find the number of occurrences of the sequence "dr" or "ea" in each name.

```
# First, define a function to count the number of times "dr" or "ea" appear in each name
def dr_ea_count(string):
    return string.count("dr") + string.count("ea")

# Then, use `map` to apply `dr_ea_count` to each name in the "Name" column
babynames["dr_ea_count"] = babynames["Name"].map(dr_ea_count)

# Sort the DataFrame by the new "dr_ea_count" column so we can see our handiwork
babynames.sort_values(by = "dr_ea_count", ascending = False).head(5)
```

	State	Sex	Year	Name	Count	name_lengths	dr_ea_count
101969	CA	F	1986	Deandrea	6	8	3
304390	CA	M	1985	Deandrea	6	8	3
131022	CA	F	1994	Leandrea	5	8	3
115950	CA	F	1990	Deandrea	5	8	3
108723	CA	F	1988	Deandrea	5	8	3

If we want to remove a column or row of a DataFrame, we can call the `.drop` method. Use the `axis` parameter to specify whether a column or row should be dropped. Unless otherwise specified, `pandas` will assume that we are dropping a row by default.

```
# Drop our "dr_ea_count" and "length" columns from the DataFrame
babynames = babynames.drop(["dr_ea_count", "name_lengths"], axis="columns")
babynames.head(5)
```

	State	Sex	Year	Name	Count
313143	CA	M	1989	Franciscojavier	6
333732	CA	M	1997	Ryanchristopher	5
330421	CA	M	1996	Franciscojavier	8
323615	CA	M	1993	Johnchristopher	5
310235	CA	M	1988	Franciscojavier	10

Notice that we reassigned `babynames` to the result of `babynames.drop(...)`. This is a subtle, but important point: `pandas` table operations **do not occur in-place**. Calling `dataframe.drop(...)` will output a *copy* of `dataframe` with the row/column of interest removed, without modifying the original `dataframe` table.

In other words, if we simply call:

```
# This creates a copy of `babynames` and removes the row with label 3...
babynames.drop(3, axis="rows")

# ...but the original `babynames` is unchanged!
# Notice that the row with label 3 is still present
babynames.head(5)
```

	State	Sex	Year	Name	Count
313143	CA	M	1989	Franciscojavier	6
333732	CA	M	1997	Ryanchristopher	5
330421	CA	M	1996	Franciscojavier	8
323615	CA	M	1993	Johnchristopher	5
310235	CA	M	1988	Franciscojavier	10

3.4 Aggregating Data with GroupBy

Up until this point, we have been working with individual rows of DataFrames. As data scientists, we often wish to investigate trends across a larger *subset* of our data. For example, we may want to compute some summary statistic (the mean, median, sum, etc.) for a group of rows in our DataFrame. To do this, we'll use `pandas GroupBy` objects.

Let's say we wanted to aggregate all rows in `babynames` for a given year.

```
babynames.groupby("Year")
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001FD34F9E640>
```

What does this strange output mean? Calling `.groupby` has generated a `GroupBy` object. You can imagine this as a set of "mini" sub-DataFrames, where each subframe contains all of the rows from `babynames` that correspond to a particular year.

The diagram below shows a simplified view of `babynames` to help illustrate this idea.

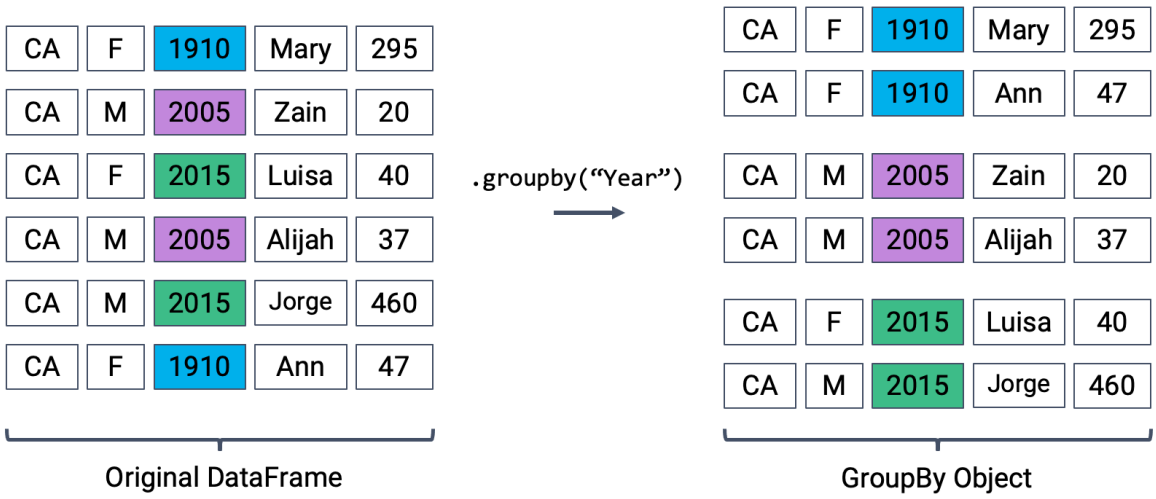


Figure 3.1: Creating a GroupBy object

We can't work with a `GroupBy` object directly – that is why you saw that strange output earlier, rather than a standard view of a DataFrame. To actually manipulate values within these "mini" DataFrames, we'll need to call an *aggregation method*. This is a method that tells `pandas` how to aggregate the values within the `GroupBy` object. Once the aggregation is applied, `pandas` will return a normal (now grouped) DataFrame.

The first aggregation method we'll consider is `.agg`. The `.agg` method takes in a function as its argument; this function is then applied to each column of a "mini" grouped DataFrame. We end up with a new DataFrame with one aggregated row per subframe. Let's see this in action by finding the `sum` of all counts for each year in `babynames` – this is equivalent to finding the number of babies born in each year.

```
babynames.groupby("Year").agg(sum).head(5)
```

Year	Count
1910	9163
1911	9983
1912	17946
1913	22094
1914	26926

We can relate this back to the diagram we used above. Remember that the diagram uses a simplified version of `babynames`, which is why we see smaller values for the summed counts.

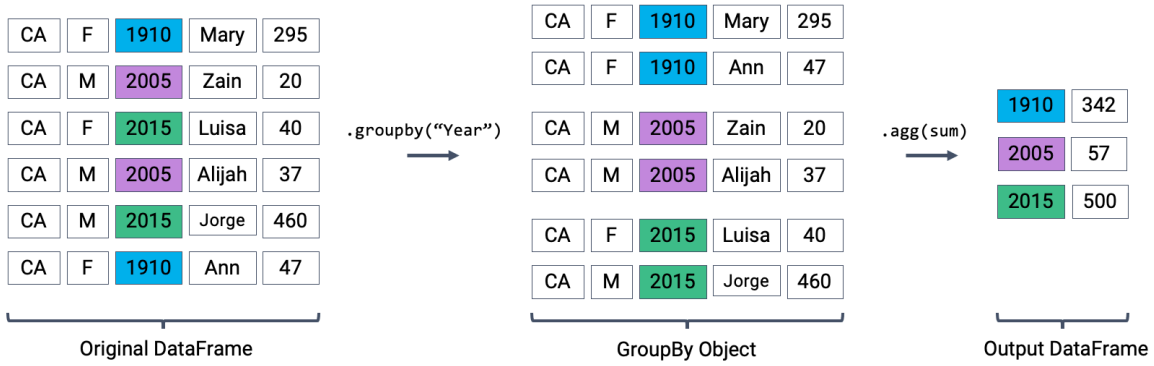


Figure 3.2: Performing an aggregation

Calling `.agg` has condensed each subframe back into a single row. This gives us our final output: a DataFrame that is now indexed by `"Year"`, with a single row for each unique year in the original `babynames` DataFrame.

You may be wondering: where did the `"State"`, `"Sex"`, and `"Name"` columns go? Logically, it doesn't make sense to `sum` the string data in these columns (how would we add "Mary" + "Ann"?). Because of this, `pandas` will simply omit these columns when it performs the aggregation on the DataFrame. Since this happens implicitly, without the user specifying that these columns should be ignored, it's easy to run into troubling situations where columns are removed without the programmer noticing. It is better coding practice to select *only* the columns we care about before performing the aggregation.

```
# Same result, but now we explicitly tell Pandas to only consider the "Count" column when
babynames.groupby("Year")[["Count"]].agg(sum).head(5)
```

Year	Count
1910	9163
1911	9983
1912	17946
1913	22094
1914	26926

3.4.1 Parting note

Manipulating `DataFrames` is a skill that is not mastered in just one day. Due to the flexibility of `pandas`, there are many different ways to get from a point A to a point B. We recommend trying multiple different ways to solve the same problem to gain even more practice and reach that point of mastery sooner.

Next, we will start digging deeper into the mechanics behind grouping data.