FOR PRESENTER USE ONLY

Cleaning and Combining Data With Pandas

Overview

In this lesson, students will learn how to use the Pandas library to clean data and combine multiple DataFrames.

Duration

120 minutes

Learning Objectives

In this lesson, students will:

- Use Pandas to handle missing or problematic data values.
- Identify appropriate cleaning strategies for specific types of data.
- Use groupby() and JOIN statements to combine data with Pandas.
- Create insights from data by splitting and combining data segments.

Pre-Class Materials and Preparation

For remote classrooms: Virtual breakout rooms and Slack may be needed to facilitate the partner exercise and discussions. As you plan for your lesson:

- Consider how you'll create pairs for the partner exercise (randomly, or with pre-assigned partners).
- Determine how (if at all) exercise timing may need to be adjusted.
- For helpful tips, keep an eye out for the **For remote classrooms** tag in the speaker notes.
- Prepare screenshots and answers to exercises in advance so that they can be easily shared in Slack during your lecture.

FOR PRESENTER USE ONLY

Suggested Agenda

Time	Activity
0:00–0:30	**Introduction + Handling Missing Data**
0:30–0:55	**Cleaning and Formatting Data**
0:55–1:05	**Break**
1:05–1:50	**Combining Data**
1:50–2:00	**Wrapping Up, Q&A, and Exit Ticket Completion**

Jupyter Notebook

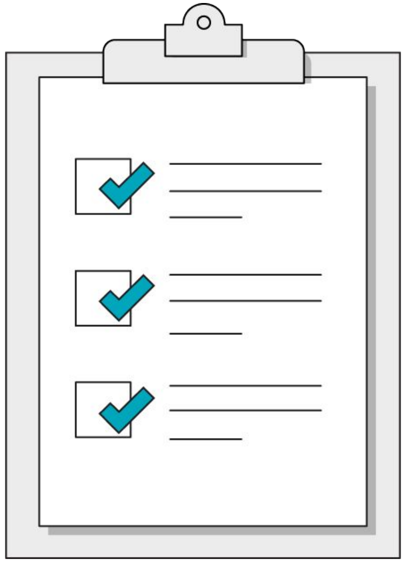
The exercises referred to in this lesson can be found in the Python Workbooks + Data folder.

Cleaning and Combining Data With Pandas

GENERAL ASSEMBLY

Our Learning Goals

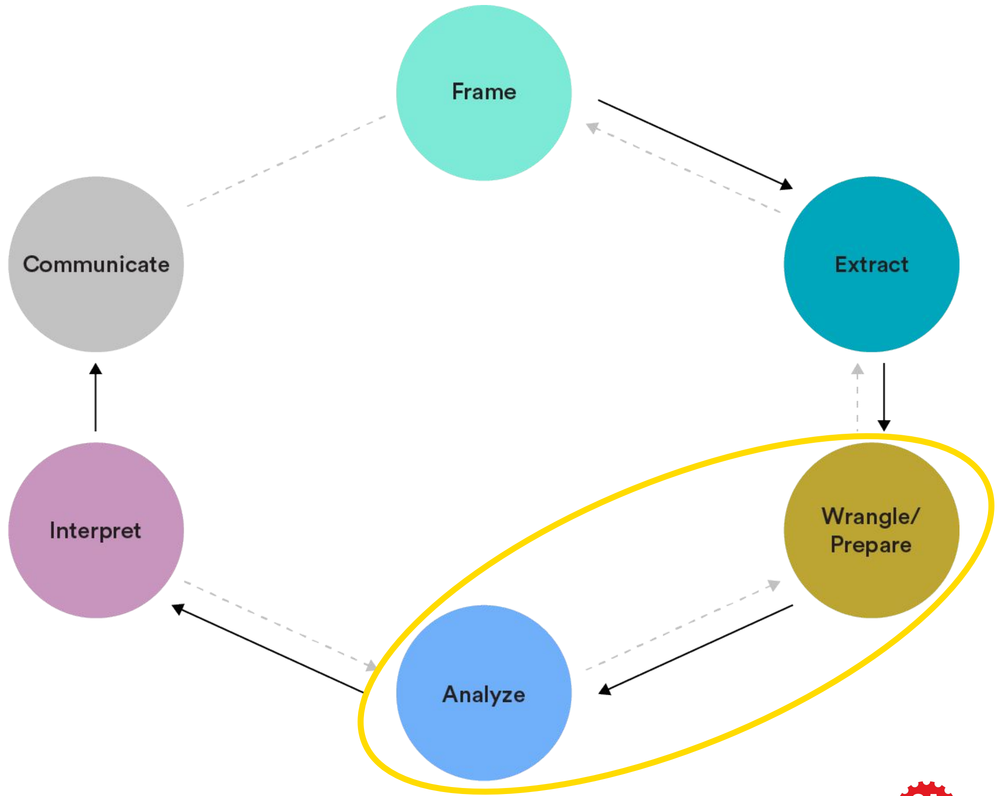
- Use Pandas to handle missing or problematic data values.
- Identify appropriate cleaning strategies for specific data types.
- Use groupby() and JOIN statements to combine data with Pandas.
- Create insights from data by splitting and combining data segments.



The Data Analytics Workflow

Wrangle/Prepare: Clean and prepare relevant data.

Analyze: Structure, comprehend, and visualize data.



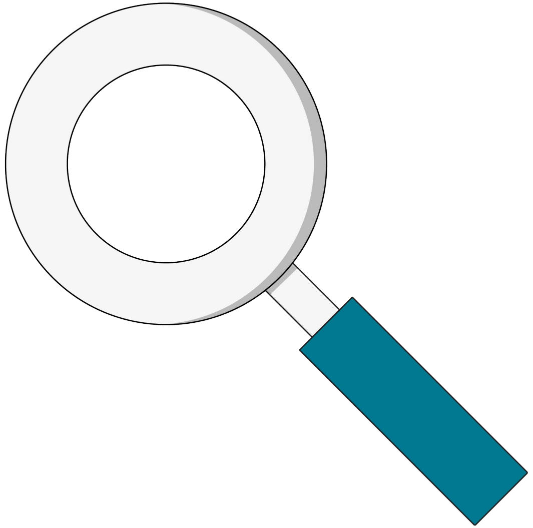
Cleaning and Combining Data With Pandas

Handling Missing Data

NULLs

A **NULL value** is **any missing value** in your data.

One common way of conceptualizing a NULL value is thinking of it as “**empty**” — not zero, not the word “NULL,” but simply *empty*.



Identifying Missing Data

The first step when dealing with NULLs is to to identify whether or not we're missing data at all.

We have a few ways to explore missing data, reminiscent of Boolean filters:

```
# isnull converts values into True if NULL, False otherwise

data_frame.isnull()

# we can then use .sum() to count up the number of NULL values

data_frame.isnull().sum()
```

Understanding Missing Data

Finding missing data is the easy part! Determining what to do next is more complicated.

Typically, we're most interested in knowing **why** we are missing data. Once we know the "**type of missingness**" (i.e., the source or cause of missing data), we can proceed effectively. This is essential to deciding whether to delete incomplete values or fill them in and, if so, with what.

Four Primary Strategies for Handling NULLs

1. Using external references, **find the true value of the missing data** and fill it in.
2. Make educated guesses about what the values could be; **impute (i.e., fill in) missing values** with the mean, median, or some other number.
3. **Ignore them.**
4. **Delete rows containing NULL values.**

Proceed with caution! These can rely on dangerous assumptions and are usually not good approaches.

Options for Missing Values

	Rationale	Example
Remove them	For some analyses, rows without a specific column are entirely useless.	`df["column"].dropna(inplace=False)` `df.dropna(subset=["column_a", "column_b"])`
Fill with arbitrary value	Useful if you want to keep incomplete records but still mark them with a specific value.	`df["column"].fillna("Unknown")` `df.fillna({"column_a":0, "column_b":100})`
Fill with computed value	For some statistical analysis, it may make sense to fill in the mean or median of the complete records.	`df["column"].fillna(df["column"].mean())`

More Options for Data Cleaning

	Rationale	Example

Replace specific values	You can replace a value or regex pattern with something else, which can help standardize data sets	`df.replace("Illinois", "IL")`
		`# replaces all appearances of "Illinois"with the abbreviation "IL"`

Fill with interpolated values	Pandas will automatically calculate missing values based on linear calculations	`# given [0,1,NA,3] as a series...`
		`df['sales_total'].interpolate()`
		`# results in [0,1,2,3]`

Guided Walk-Through:

10.1 Identifying Missing Data

20 minutes

Let's use the “Orders” table from Superstore to work with missing data in Section 10.1 of the workbook.

Cleaning and Combining Data With Pandas

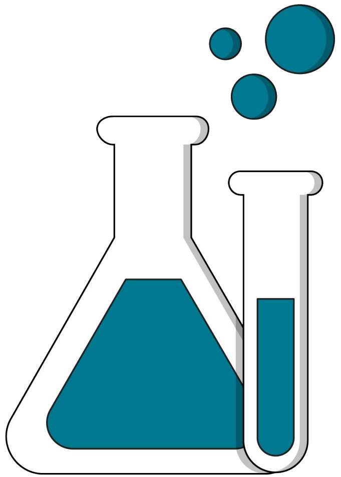
Cleaning and Formatting Data

Data Cleaning

Data cleaning is the process of assembling data into a **usable format for analysis**.

Common data cleaning actions include:

- **Reformatting dates** so that Python recognizes them.
- **Extracting day/hour/month/year from a date** to aggregate by those categories.
- **Removing duplicate values** or rows.
- **Combining data sources** into one table.
- **Concatenating or separating** data.



Modifying Series Within DataFrames

Many of our cleaning operations involve applying an operation to a Series:

```
data_frame[‘column_a’] = data_frame[‘column_a’].to_numeric()
```

This can also be used to create new columns based on existing data:

```
data_frame[‘Fahrenheit’] = data_frame[‘Celsius’] * (9/5) + 32
```

Handling Dates and Times

Some of the most challenging, frequently ill-formatted types of data are dates and times. Fortunately, Pandas is on top of it with the **`.to_datetime()`** method:
`df['Column'].to_datetime(inplace=True)`

Once a Series has been given a datetime data type, we can use access methods to extract specific time properties, like day or hour.

We can also use the Pandas **`Timestamp()`** method to convert data into timestamps:
`pd.Timestamp(date_string_or_number)`

Anatomy of a TimeStamp

```
moment_in_time = pd.TimeStamp("19890602T07:43:55")
```

moment_in_time.year	1989
moment_in_time.month	6
moment_in_time.day	2
moment_in_time.hour	7
moment_in_time.minute	43
moment_in_time.second	55

`apply()`

There is also the `apply()` method, which allows us use any function or method:

```
def strip_dollar_sign(str):
    return str.replace("$", "")

df[‘column_a’] = df[‘column_a’].apply(strip_dollar_sign)
```

If you need to apply a function to each row by row, set **`axis=1`** as an argument.

Addressing Duplicates

Fortunately, the issue of duplicate data is a mere Pandas method away from being solved! We can use the `.drop_duplicates()` method:

```
df.drop_duplicates()
```

If we want to drop duplicates based on certain columns, we can do that, too:

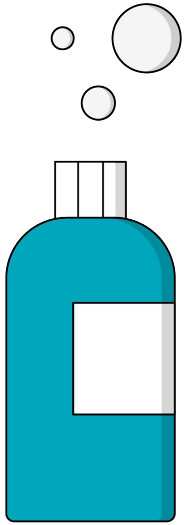
```
df.drop_duplicates(subset=[‘column_a’, ‘column_b’])
```

Partner Exercise:

10.2 Cleaning and Formatting Data

15 minutes

Let's use the Superstore data set for Section 10.2 to practice applying cleaning functions.



Cleaning and Combining Data With Pandas

Combining Data

Aggregating With `groupby()`

In Pandas, **`groupby()`** statements allow us to segment our population to a specific subset and draw calculations based on those segments.

```
data_frame.groupby([‘column_a’]).count()
```

We can think about a **`groupby()`** statement in three steps:

- **Split:** Separate our DataFrame into groups according to a specific attribute.
- **Apply:** Apply some function to the groups, like **`sum`**, **`count`**, or **`max`**.
- **Combine:** Put our DataFrame back together.

Taking Aggregate Measures

We can use the `.agg()` method to get multiple aggregate values:

```
df.groupby(‘col_a’)[‘col_b’].agg([‘count’, ‘mean’, ‘min’, ‘max’])
```

This command...

1. Takes our DataFrame, **`df`**...
2. Groups it by the values in **`col_a`**...
3. And calculates the count, mean, minimum, and maximum of **`col_b`**.

We can also **`groupby()`** multiple columns to drill down further:

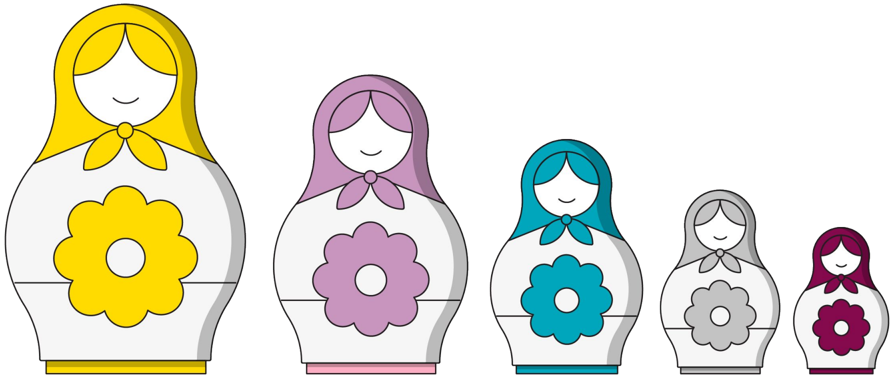
```
df.groupby(['first_column', 'second_column'])
```

Solo Exercise:

10.3 `groupby()` Insights

15 minutes

Use **`groupby()`** to segment the data from Superstore in Section 10.3.



Concatenate

We can combine two DataFrames together with **`df.concat()`**, which gives us the option to stack the DataFrames or add it as new columns.

```
df1 = pd.DataFrame([[‘a’, 1], [‘b’, 2]], columns=[‘letter’, ‘number’])
df2 = pd.DataFrame([[‘c’, 3], [‘d’, 4]], columns=[‘letter’, ‘number’])

df_with_more_rows = pd.concat([df1, df2])
df_with_more_columns = pd.concat([df1, df2], axis=1)
```

What Is JOINing?

JOINing is the process of **combining DataFrames according to specific values**.

Traditionally, this would be done with SQL.

JOINing allows us to:

- **Reduce the size** of a database.
- **Increase the speed** at which data is queried and returned.
- **Reduce the redundancy** of the data stored in the database.

JOINs

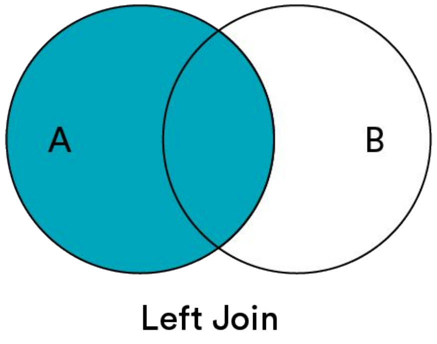
A **JOIN** relies on multiple data sets that share a common unique identifier, or “**key**.”

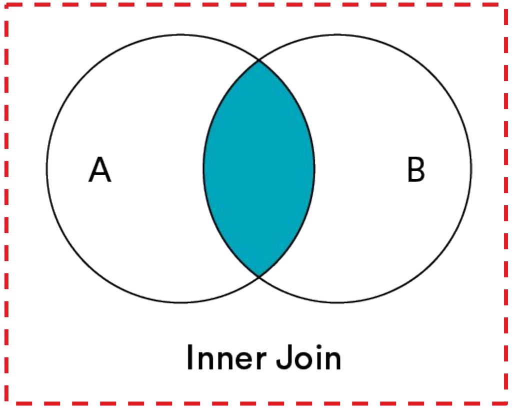
drivers		
id	name	vehicle_id
1	Janet	3
2	Terrell	4
3	Yoko	4
4	Lee	5

vehicles	
id	vehicle_name
1	Explorer
2	Civic
3	Corolla
4	Impala

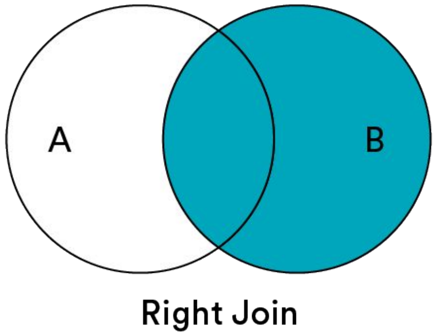
id	name	vehicle_id	vehicle_name
1	Janet	3	Corolla
2	Terrell	4	Impala
3	Yoko	4	Impala

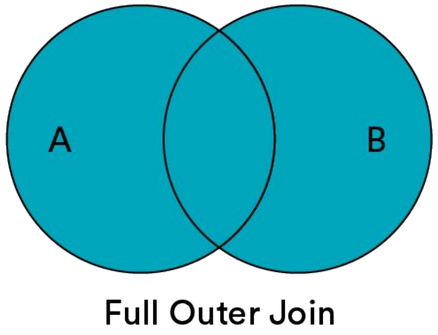
Types of JOINs





INNER JOIN is the same thing as JOIN.





Using merge

The robust method for JOINing in **`Pandas`** is **`merge()`,** which accepts several parameters:

```
pd.merge(left_df, right_df, how, left_on, right_on)
```

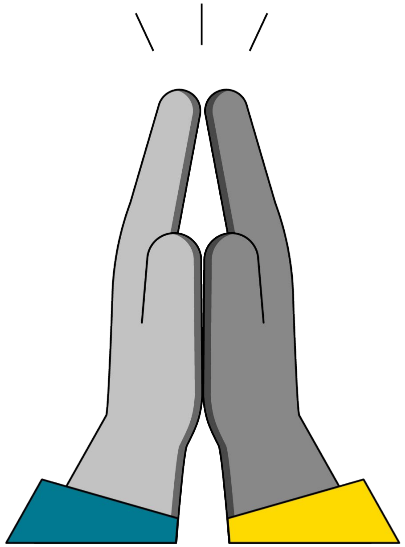
As you may have guessed, the first two parameters are the DataFrames to JOIN. The third parameter describes the type of JOIN, typically “left.” The last two parameters provide the column name for the shared column, or foreign key, that will be used to combine the two DataFrames.

Group Exercise:

10.4 JOINing Data Sets

15 minutes

Let’s practice JOINing stock data and the Superstore data in Section 10.4 of the workbook.



Cleaning and Combining Data With Pandas

Wrapping Up

Recap

In today's class, we...

- Used Pandas to handle missing or problematic data values.
- Identified appropriate cleaning strategies for specific types of data.
- Used groupby() and JOIN statements to combine data with Pandas.
- Created insights from data by splitting and combining data segments.

Looking Ahead

On your own:

- Work through the Python progress assessment on myGA (due at the end of the unit).
- Be sure to have instructor approval for your capstone project.
- Join someone else's project or invite others to join yours!

Next Class:

Python Unit Lab

Don’t Forget: Exit Tickets!