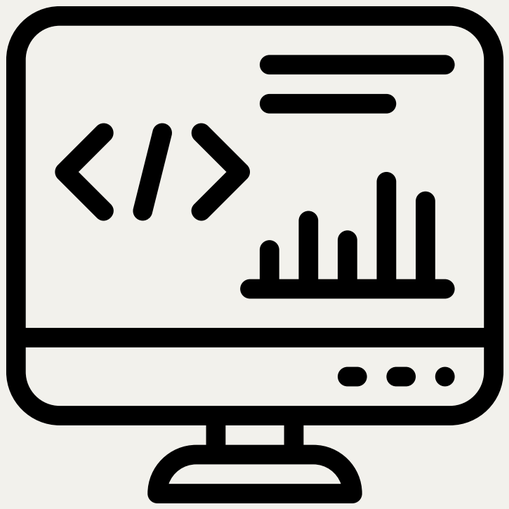


Pandas Review

لاتنسونا من دعواتكم

الله يوفقنا ويوفقكم وبإذن الله نتقابل في الصيفية 🤲🏻

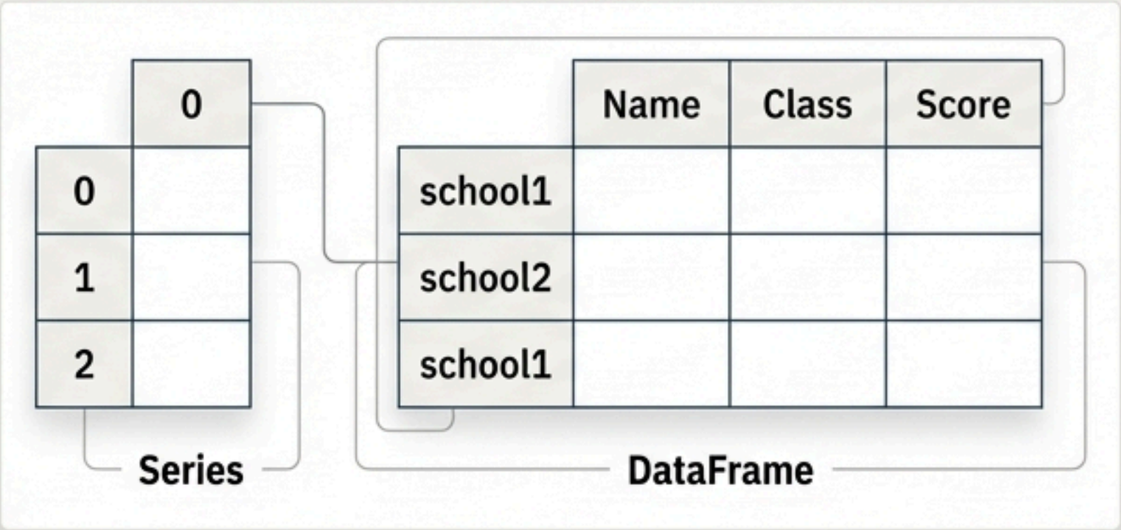
لو عندكم اي سؤال او اقتراح
هذي حسابتنا في X

Muath Sara Ice

Understanding the Core Components: Series and DataFrame

Pandas is built on two primary data structures:

- **Series**: A one-dimensional labeled array, similar to a column in a spreadsheet. It's a cross between a Python list and a dictionary, with an ordered index.
- **DataFrame**: A two-dimensional labeled data structure with columns of potentially different types. It is the heart of the Pandas library and can be thought of as a spreadsheet or a SQL table.



Method 1: From a list of Series

```
record1 = pd.Series({'Name': 'Alice', 'Class': 'Physics', 'Score': 85})
record2 = pd.Series({'Name': 'Jack', 'Class': 'Chemistry', 'Score': 82})
record3 = pd.Series({'Name': 'Helen', 'Class': 'Biology', 'Score': 90})

df = pd.DataFrame([record1, record2, record3],
                  index=['school1', 'school2', 'school1'])
```

Method 2: From a list of dictionaries

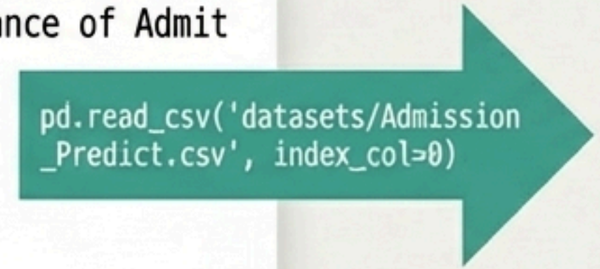
```
students = [{'Name': 'Alice', 'Class': 'Physics', 'Score': 85},
            {'Name': 'Jack', 'Class': 'Chemistry', 'Score': 82},
            {'Name': 'Helen', 'Class': 'Biology', 'Score': 90}]

df = pd.DataFrame(students,
                  index=['school1', 'school2', 'school1'])
```

Step 1: Loading and Inspecting Your Data

The most common data science workflow begins with reading data from an external file. Pandas provides the powerful `read_csv()` function to seamlessly load comma-separated value files into a DataFrame.

Raw CSV File

```
Serial No.,GRE Score,TOEFL Score,University
Rating,SOP,LOR ,CGPA,Research,Chance of Admit
1,337,118,4,4.5,4.5,9.65,1,0.92
2,324,107,4,4.0,4.5,8.87,1,0.76
3,316,104,3,3.0,3.5,8.00,1,0.72
...
```



Pandas DataFrame (df.head())

Serial No.	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
1	337	118	4	4.5	4.5	9.65	1	0.92
2	324	107	4	4.0	4.5	8.87	1	0.76
3	316	104	3	3.0	3.5	8.00	1	0.72
4	322	110	3	3.5	2.5	8.67	1	0.80
5	314	103	2	2.0	3.0	8.21	0	0.65

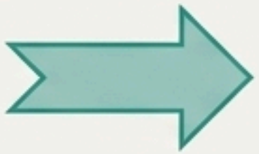
```
# Load the data, setting the first column as the index
df = pd.read_csv('datasets/Admission_Predict.csv', index_col=0)

# Display the top 5 rows
df.head()
```

Step 2: Cleaning Column Names for Consistency

Real-world data often has formatting issues, like extra whitespace in column names, which can cause errors. Robust cleaning is a critical step for reliable data access.
IBM Plex Sans

Before

```
df.columns
['GRE Score', 'TOEFL Score', …, 'LOR ', …]
```



After

```
after
['gre score', 'toefl score', …, 'lor', …]
```

Solution 2: Use a mapper function

```
# Cleans all column names at once
df_new = df.rename(mapper=str.strip, axis='columns')
```

Solution 3: Direct modification

```
# Clean columns with a list comprehension
cols = list(df.columns)
df.columns = [x.lower().strip() for x in cols]
df.head()
```

Step 3: Selecting Data with `.loc` and `.iloc`

`.loc`: Selects data based on **labels**.

`df.loc['school2']`

`.iloc`: Selects data based on **integer position**.

`df.iloc[0]`

	Name	Class	Score	Ranking
school1	Jam	AA	72	1
school2	Mark	AB	63	2
school1	John	AB	61	3
school3	Martis	AC	51	4
school4	Mary	AB	83	5

Selecting a Single Cell

`df.loc['school1', 'Name']`

Slicing Columns

`df.loc[:, ['Name', 'Score']]`

Warning: Avoid Chaining

Chained indexing like `df.loc['school1']['Name']` can be inefficient and work on a copy of the data, leading to unpredictable results. Prefer the combined selection: `df.loc['school1', 'Name']`.

Advanced Querying with Boolean Masking

1. Create the Mask

chance of admit
0.85
0.72
0.90
0.65
0.88

`df['chance of admit'] > 0.8` →

Mask
True
False
True
False
True

2. Apply the Mask

gre score	toefl score	lor	chance of admit
370	320	1.0	0.85
160	310	0.3	False
320	320	1.0	**True**
0.65	270	0.3	False
370	320	1.0	**True**

3. Filtered Result

gre score	toefl score	lor	chance of admit
370	320	1.0	0.85
370	300	1.0	0.90
370	320	1.0	0.88

```
# 1. Create the boolean mask
admit_mask = df['chance of admit'] > 0.8

# 2. Apply the mask to the DataFrame
high_chance_df = df[admit_mask]
```

Combining Masks

Use the `&` (and) and `|` (or) operators. **Crucially, each condition must be wrapped in parentheses.**

```
# Correct way to combine masks
df[(df['chance of admit'] > 0.8) & (df['gre score'] > 320)]
```

Step 4: Handling Missing Values

Missing data (often represented as `NaN`) is common. Pandas provides flexible tools to manage it.

- **Detecting**: `df.isnull()` creates a boolean mask showing `True` for missing values.
- **Dropping**: `df.dropna()` removes rows containing any missing values.
- **Filling**: `df.fillna(value)` replaces `NaN`s with a specified value.
  - `df.fillna(0)`: Fills with a constant.
  - `df.fillna(method='ffill')`: **Forward-fill** propagates the last valid observation forward. Ideal for sorted time-series data.

Forward Fill (`ffill`) in Action

Before Forward Fill

time	user	paused	volume
2023-10-27 10:00	A	True	75
2023-10-27 10:01	A	*NaN*	*NaN*
2023-10-27 10:02	B	False	60
2023-10-27 10:03	B	*NaN*	*NaN*
2023-10-27 10:04	B	*NaN*	80

```
df = df.fillna(method='ffill')
```

After Forward Fill

time	user	paused	volume
2023-10-27 10:00	A	True	75
2023-10-27 10:01	A	True	75
2023-10-27 10:02	B	False	60
2023-10-27 10:03	B	False	60
2023-10-27 10:04	B	False	80

```
# Sort the data to ensure correct propagation
df = df.set_index('time')
df = df.sort_index()

# Forward-fill the missing 'paused' and 'volume' values
df = df.fillna(method='ffill')
```

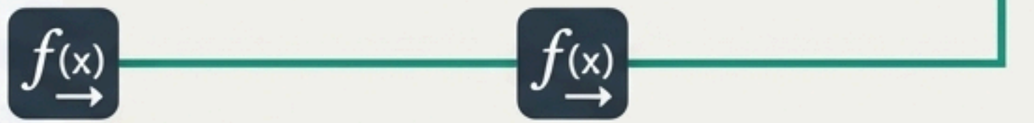
Step 5: Transforming Data and Creating New Features with `.apply()`

The `.apply()` function is a powerful tool for applying a custom function along an axis of a DataFrame (e.g., to each row). This is ideal for creating new columns based on the values of existing columns.

Example: Calculate min/max population for each county

```
def min_max_population(row):
    # Select the relevant population estimate columns
    data = row[['POPESTIMATE2010', 'POPESTIMATE2011',
                'POPESTIMATE2012', 'POPESTIMATE2013',
                'POPESTIMATE2014', 'POPESTIMATE2015']]

    # Create new columns in the row
    row['max_pop'] = np.max(data)
    row['min_pop'] = np.min(data)
    return row

# Apply the function to every row (axis='columns')
df = df.apply(min_max_population, axis='columns')
```

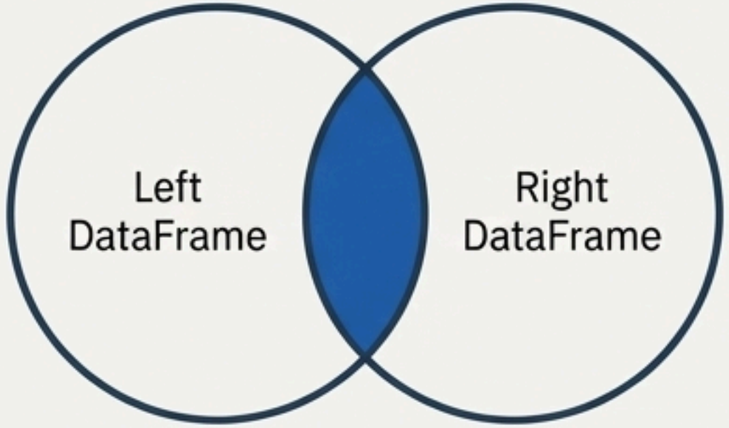
STNAME	CTYNAME	POPESTIMATE2010	POPESTIMATE2011	POPESTIMATE2012	POPESTIMATE2013	POPESTIMATE2014	POPESTIMATE2015	max_pop	min_pop
Alabama	Autauga County	54660	55253	55175	55038	55290	55347	55347	54660
Alabama	Baldwin County	183193	186659	190396	195126	199713	203709	203709	183193
Alabama	Barbour County	27341	27226	27159	26973	26815	26489	27341	26489
...	...	...	...	...	...	...	...	...	...



Combining Datasets: The Theory of Joins

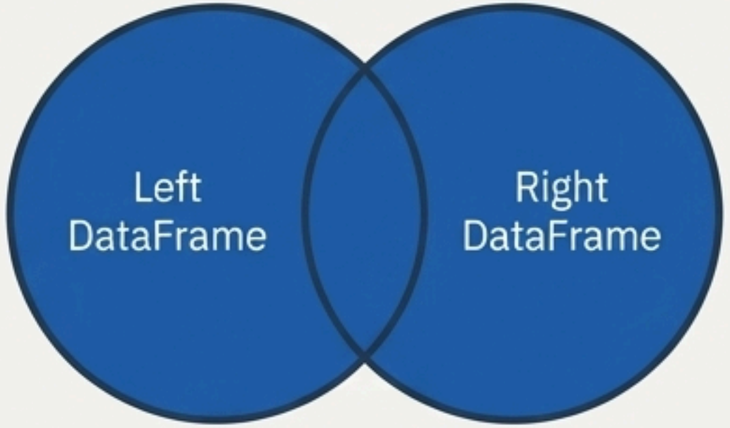
Often, your data is split across multiple tables. A “merge” or “join” combines them based on common keys. Relational theory defines four main join types, which can be visualized with Venn diagrams.

Inner Join (Intersection)



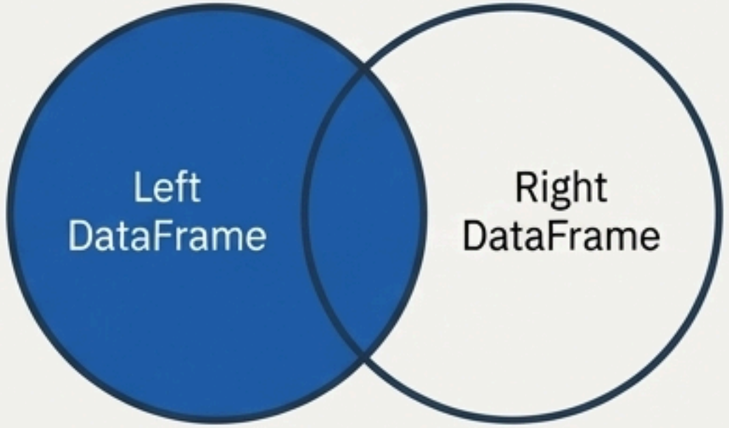
Keeps only the records that exist in **both** DataFrames.

Full Outer Join (Union)



Keeps **all** records from both DataFrames, filling in missing data with `NaN`.

Left Join



Keeps all records from the **left** DataFrame, and matches records from the right.

Practice: `pd.merge()` and `pd.concat()`

pd.merge()

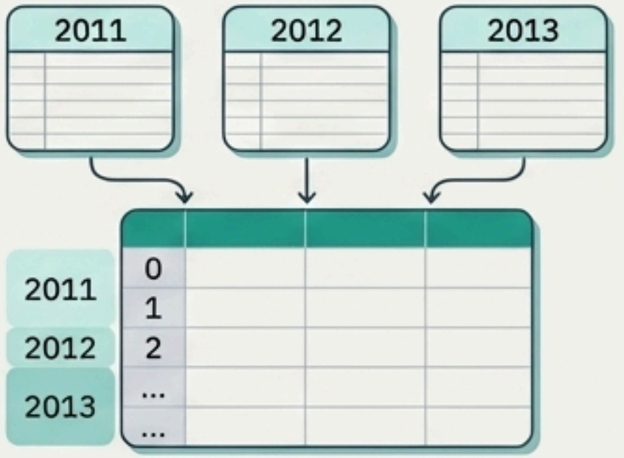
Pandas implements joins with `pd.merge()`.

```
# Create staff and student DataFrames
staff_df = pd.DataFrame([{'Name': 'Kelly', 'Role': 'Director of HR'},
                         {'Name': 'Sally', 'Role': 'Course liasion'},
                         {'Name': 'James', 'Role': 'Grader'}])
student_df = pd.DataFrame([{'Name': 'James', 'School': 'Business'},
                           {'Name': 'Mike', 'School': 'Law'},
                           {'Name': 'Sally', 'School': 'Engineering'}])

# Inner Join: Only people who are both staff and students
pd.merge(staff_df, student_df, how='inner', on='Name')

# Outer Join: All people from both lists
pd.merge(staff_df, student_df, how='outer', on='Name')

# Left Join: All staff, with student info if available
pd.merge(staff_df, student_df, how='left', on='Name')
```

	Name	Role	School
0	Kelly	Director of HR	NaN
1	Sally	Course liasion	Engineering
2	James	Grader	Business

pd.concat()

Use `pd.concat()` for vertical stacking. The keys parameter is useful for creating a multi-level index to track data origin.




```
# Concatenate scorecard data from multiple years
frames = [df_2011, df_2012, df_2013]
all_years_df = pd.concat(frames, keys=['2011', '2012', '2013'])
```

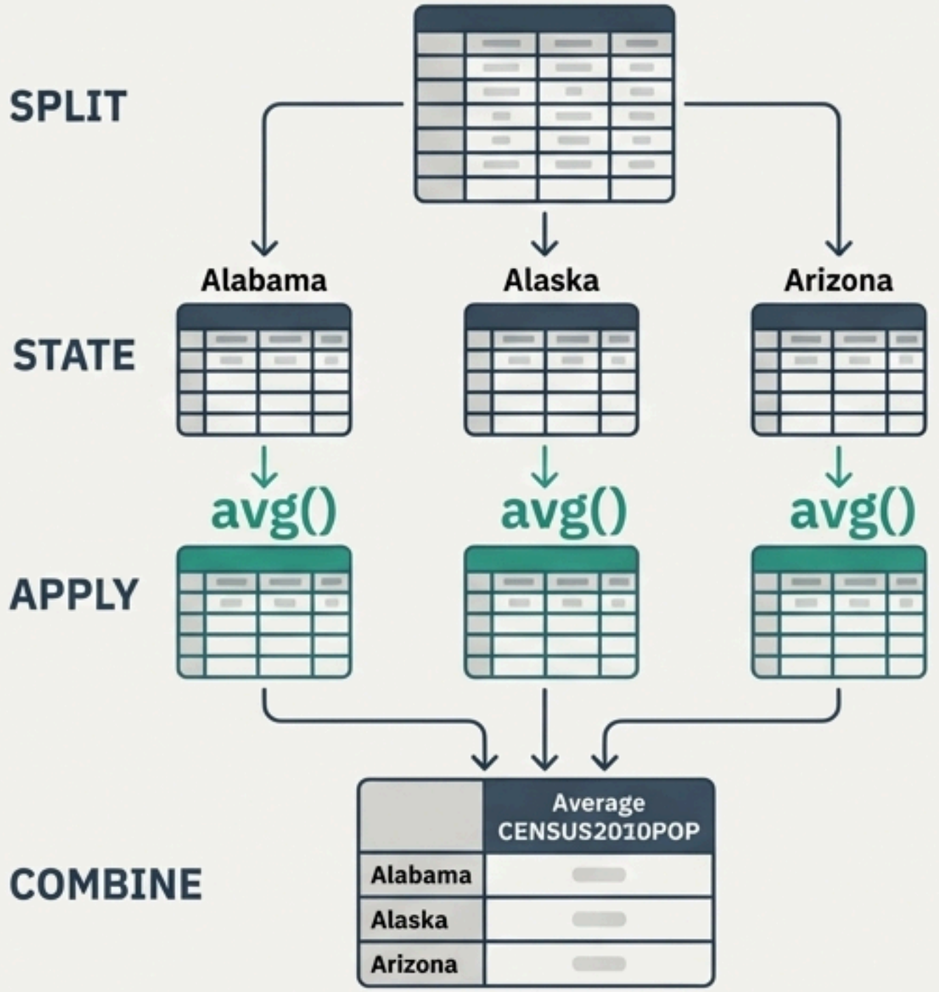
Step 6: Summarizing Data with the Split-Apply-Combine Strategy (IBM Bold)

The `groupby()` operation is a cornerstone of data analysis, following the **Split-Apply-Combine** pattern.

1. **Split**: The DataFrame is split into groups based on a key (e.g., all rows for 'Alabama', all rows for 'Alaska').
2. **Apply**: A function (e.g., `mean()`, `sum()`) is applied to each group independently.
3. **Combine**: The results are combined into a new, summary DataFrame.

Code Example: Find the average 2010 census population for counties within each state.

```
# Load and filter census data
df = pd.read_csv('datasets/census.csv')
df = df[df['SUMLEV']==50]

# Group by state, select column, and aggregate
df.groupby('STNAME')['CENSUS2010POP'].agg(np.average).head()
```



Reshaping and Summarizing with Pivot Tables

A `pivot_table` reshapes a DataFrame by specifying variables for rows, columns, and aggregated cell values.

Example: Compare the average university `score` across `country` and `Rank_Level` categories.

- `values`: The column to aggregate.
- `index`: The column for the new rows.
- `columns`: The column for the new columns.
- `aggfunc`: The function to apply.

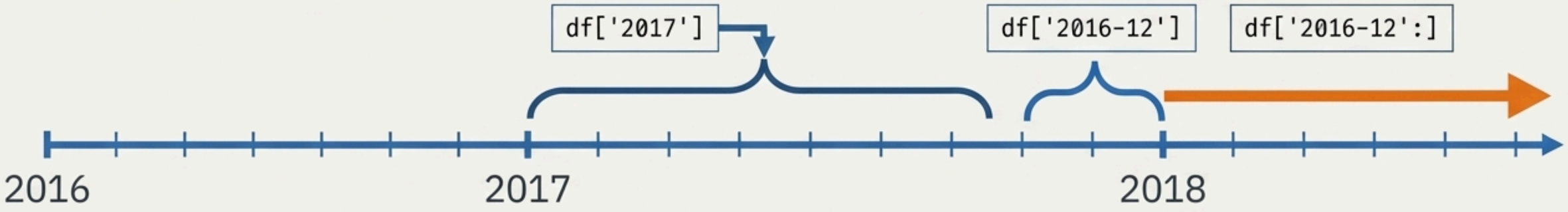
```
# First, create the 'Rank_Level' category
# (assuming create_category_function is defined)
df['Rank_Level'] = df['world_rank'].apply(create_category_function)

# Now, create the pivot table
df.pivot_table(values='score',
               index='country',
               columns='Rank_Level',
               aggfunc=[np.mean, np.max],
               margins=True).head()
```

	mean				max			
country	First Tier	Second Tier	Other	All	First Tier	Second Tier	Other	All
USA	90.5	80.2	65.4	78.7	100	95	88	100
UK	80.2	73.7	54.0	72.0	95	81	75	95
China	74.8	62.6	63.3	72.3	100	91	88	100
Australia	66.4	69.8	65.4	75.9	100	94	86	100
Germany	70.5	50.2	60.3	78.2	100	95	88	95
All	**90.5**	**80.2**	**65.4**	**78.7**	**100**	**88**	**100**	**100**

A Glimpse into Time Series Functionality

Pandas has robust, built-in functionality for **working with time series data**, including tools for converting, creating date ranges, and slicing.

- **Converting to Datetime:** pd.to_datetime() parses string formats into datetime objects.
- **Date Components:** Access components like .day, .month, and .year.
- **Slicing by Date:** Pandas supports intuitive slicing with partial date strings.




```
# Convert a column of strings to datetime objects

df['submission_time'] = pd.to_datetime(df['assignment1_submission'])
```

Writing ‘Pandorable’ Code: The Art of Method Chaining

“Pandorable” code refers to idioms that are concise, readable, and leverage the power of the Pandas library. **Method chaining** links operations in a single statement, avoiding intermediate variables and clarifying the transformation sequence.

Traditional Example (Step-by-Step)

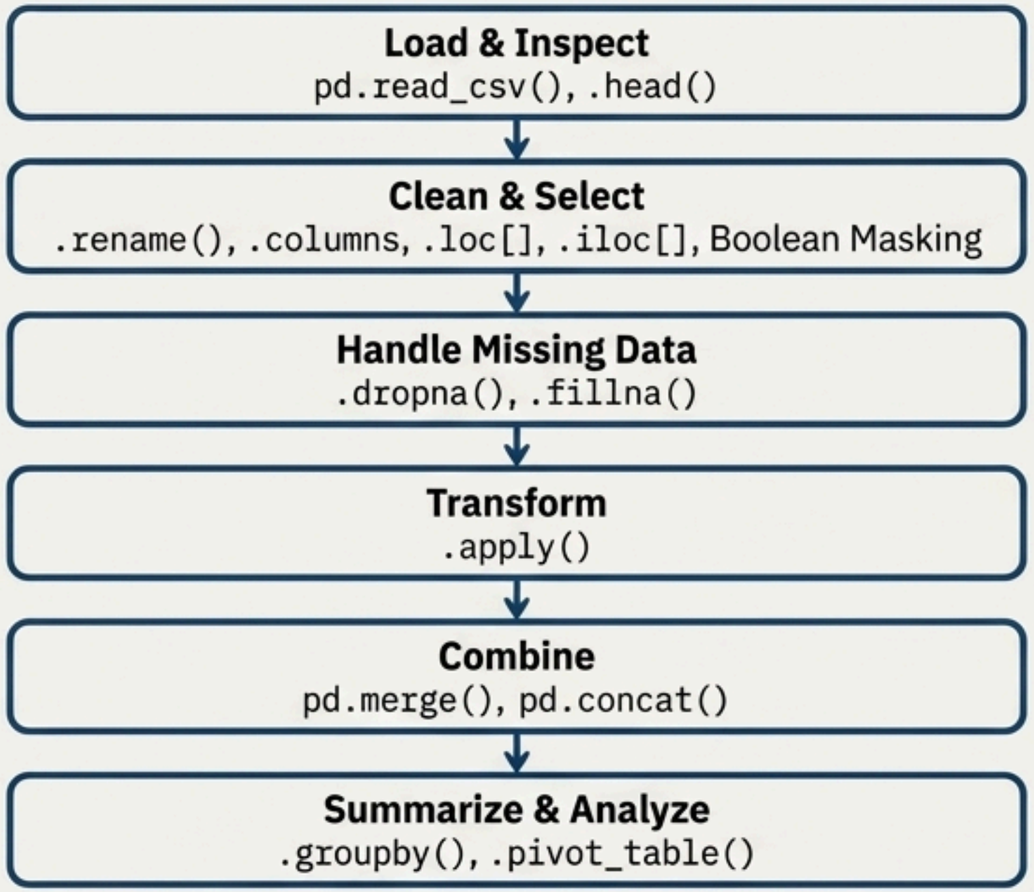
```
# Functionally identical, but uses
# intermediate steps and variables.
df = df[df['SUMLEV']==50]
df.set_index(['STNAME','CTYNAME'], inplace=True)
df.rename(columns={'ESTIMATESBASE2010':
                   'Estimates Base 2010'})
```

Pandorable Example (Method Chaining)

```
# A single, readable statement to filter,
# clean, index, and rename.
(df.where(df['SUMLEV']==50)
    .dropna()
    .set_index(['STNAME','CTYNAME'])
    .rename(columns={'ESTIMATESBASE2010':
                     'Estimates Base 2010'}))
```

The Data Scientist’s Toolkit: A Recap of the Workflow

We’ve journeyed through a complete data analysis workflow. The key is to think in terms of this structured process and apply the right function for each task.



Always strive for ‘pandorable’ code. Vectorized operations and idioms like method chaining are not just stylistic—they are significantly faster and more efficient than manual iteration.