😊 Combining DataFrames

What is Combining DataFrames in Pandas?

Combining DataFrames in Pandas refers to the process of bringing together two or more DataFrames into a single structure for analysis. This is essential when working with datasets that are split across different sources, tables, or files. Pandas provides multiple methods for combining DataFrames depending on the structure of the data and the desired outcome.

Why Combine DataFrames?

- To integrate different datasets with related information
- To enrich data by adding more columns or rows
- To cleanly concatenate data over time or categories
- To perform joins or merges similar to those in SQL

Key Methods to Combine DataFrames

1. Concatenation

- Combines DataFrames either **vertically (row-wise)** or **horizontally (column-wise)**.
- Useful when datasets share the same columns (for vertical stacking) or same index (for horizontal stacking).

Key features:

- No alignment by column names or index values.
- Indexes are preserved unless reset manually.
- Can be done with ignore_index to renumber rows.

2. Appending

- A simplified form of concatenation for adding rows to a DataFrame.
- Functionally similar to vertical concatenation.
- Deprecated in newer versions of Pandas; concat is preferred.

3. Merging

- Performs database-style joins (similar to SQL joins).
- Combines rows based on common columns or index values.
- Types of joins supported:
    - **Inner join**: keeps only matching rows
    - **Outer join**: keeps all rows, fills missing with NaN
    - **Left join**: keeps all rows from the left DataFrame

    - **Right join**: keeps all rows from the right DataFrame
- Useful for relational datasets with shared keys or identifiers.

4. Joining

- A convenience method for combining two DataFrames using their indexes or a column.
- Primarily used for index-based merging.
- Similar to SQL joins but index-driven.

Summary of Differences

Method	Combines by	Join type flexibility	Preserves Index	Typical Use Case
concat	Axis (rows or columns)	No (just stacking)	Yes	Stack data with same columns or index
append	Rows	No	Yes	Simple row addition (deprecated)
merge	Columns or indexes	Yes (inner, outer, etc.)	Depends	Database-style relational joins
join	Index or columns	Yes	Yes	Combine based on index or one key column

When to Use What

- Use **concat** when you are stacking datasets together without concern for matching on column values.
- Use **merge** when you are bringing together datasets based on shared key columns or indexes.
- Use **join** when working with index-based datasets or simple key-based merges.

Combining DataFrames effectively is essential for building comprehensive and clean datasets that support accurate analysis and modeling. Let me know if you'd like a breakdown with visual diagrams or typical use case scenarios.

To begin with the Lab

1. We start by importing NumPy and Pandas in our Jupyter notebook.

```
[2]: import numpy as np
     import pandas as pd
```

2. The code creates two dictionaries, data_one and data_two, each containing columns with lists of string values.

3. These dictionaries are then converted into pandas DataFrames named one and two using pd.DataFrame().
4. The one DataFrame contains columns "A" and "B", while two contains columns "C" and "D".
5. This setup is commonly used to organize structured data in tabular form for further analysis or manipulation using pandas.

```
[4]: data_one = {'A': ['A0', 'A1', 'A2', 'A3'],'B': ['B0', 'B1', 'B2', 'B3']}

[6]: data_two = {'C': ['C0', 'C1', 'C2', 'C3'], 'D': ['D0', 'D1', 'D2', 'D3']}

[8]: one = pd.DataFrame(data_one)

[10]: two = pd.DataFrame(data_two)

[12]: one
```

[12]:

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

```
[14]: two
```

[14]:

	C	D
0	C0	D0
1	C1	D1
2	C2	D2
3	C3	D3

6. The code axis0 = pd.concat([one, two], axis=0) combines the two DataFrames one and two vertically (row-wise).
7. The pd.concat() function is used to concatenate Pandas objects. By setting axis=0, the function stacks the rows of two below those of one.
8. Since one and two have different columns, the result will include all columns from both, with NaN values filling in where a row doesn't have a corresponding column.
9. This is useful for combining data from different sources while preserving all available information.

[16]:
```
axis0 = pd.concat([one,two],axis=0)
```

[18]:
```
axis0
```

[18]:

	A	B	C	D
0	A0	B0	NaN	NaN
1	A1	B1	NaN	NaN
2	A2	B2	NaN	NaN
3	A3	B3	NaN	NaN
0	NaN	NaN	C0	D0
1	NaN	NaN	C1	D1
2	NaN	NaN	C2	D2
3	NaN	NaN	C3	D3

10. The code axis1 = pd.concat([one, two], axis=1) combines the DataFrames one and two horizontally (column-wise).
11. The pd.concat() function with axis=1 joins the columns of both DataFrames side by side, aligning them by their index (row labels).
12. Since both one and two have the same number of rows (4), their rows align perfectly, and the result is a single DataFrame with columns A, B, C, and D.
13. This method is useful for combining different features (columns) for the same set of observations (rows)

[20]:
```
axis1 = pd.concat([one,two],axis=1)
```

[22]:
```
axis1
```

[22]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

14. The code two.columns = one.columns renames the columns of the two DataFrame to match those of one (i.e., columns "A" and "B").
15. Then, pd.concat([one, two]) concatenates the two DataFrames vertically (default axis=0) since their column names now match.
16. The result is a single DataFrame with rows from both one and two, stacked on top of each other under the same columns.
17. This is useful when combining similar datasets that originally had different column names.

[24]:
```
two.columns = one.columns
```

[26]:
```
pd.concat([one,two])
```

[26]:

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3
0	C0	D0
1	C1	D1
2	C2	D2
3	C3	D3

18. The code creates two DataFrames: registrations and logins.
    - registrations contains registration data with columns reg_id (registration ID) and name, listing people who registered (Andrew, Bobo, Claire, David).
    - logins contains login data with columns log_id (login ID) and name, listing people who logged in (Xavier, Andrew, Yolanda, Bobo).
19. These DataFrames are typically used for comparison or merging to analyze overlaps or differences between registered users and users who actually logged in.

[28]:
```
registrations = pd.DataFrame({'reg_id':[1,2,3,4],'name':['Andrew','Bobo','Claire','David']})
logins = pd.DataFrame({'log_id':[1,2,3,4],'name':['Xavier','Andrew','Yolanda','Bobo']})
```

[30]:
```
registrations
```

[30]:

	reg_id	name
0	1	Andrew
1	2	Bobo
2	3	Claire
3	4	David

[32]:
```
logins
```

[32]:

	log_id	name
0	1	Xavier
1	2	Andrew
2	3	Yolanda
3	4	Bobo

20. pd.merge() combines two DataFrames into one based on shared columns or indexes, similar to SQL JOIN operations. You specify how to join (inner, outer, left, right) and the columns/indexes to use as keys. Using the help function, you can see more about it.

```
[34]: help(pd.merge)
```

```
Help on function merge in module pandas.core.reshape.merge:

merge(left: 'DataFrame | Series', right: 'DataFrame | Series', how: 'MergeHow' = 'inner', on: 'IndexLabel | None' = None, left_on: 'IndexLabel | None' =
None, right_on: 'IndexLabel | None' = None, left_index: 'bool' = False, right_index: 'bool' = False, sort: 'bool' = False, suffixes: 'Suffixes' = ('_x',
'_y'), copy: 'bool | None' = None, indicator: 'str | bool' = False, validate: 'str | None' = None) -> 'DataFrame'
    Merge DataFrame or named Series objects with a database-style join.

    A named Series object is treated as a DataFrame with a single named column.

    The join is done on columns or indexes. If joining columns on
    columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes
    on indexes or indexes on a column or columns, the index will be passed on.
    When performing a cross merge, no column specifications to merge on are
    allowed.

    .. warning::

        If both key columns contain rows where the key is a null value, those
```

21. The code performs an **inner merge** (or inner join) between the registrations and logins DataFrames using pd.merge().
    - pd.merge(registrations, logins, how='inner', on='name') joins the two DataFrames on the common "name" column, keeping only the rows where names appear in both DataFrames (like Andrew and Bobo).
    - pd.merge(registrations, logins, how='inner') does the same, and since "name" is the only common column, pandas automatically uses it as the key for the merge.
22. The merge() function is used to combine DataFrames based on common columns or indices, similar to SQL joins.

```
[36]: # Notice pd.merge doesn't take in a list like concat
pd.merge(registrations,logins,how='inner',on='name')
```

[36]:

	reg_id	name	log_id
0	1	Andrew	2
1	2	Bobo	4

```
[38]: # Pandas smart enough to figure out key column (on parameter) if only one column name matches up
pd.merge(registrations,logins,how='inner')
```

[38]:

	reg_id	name	log_id
0	1	Andrew	2
1	2	Bobo	4

23. The code pd.merge(registrations, logins, how='left') performs a **left join** between the registrations and logins DataFrames.
24. This means all rows from the registrations DataFrame are kept, and matching rows from logins are added based on the common column "name".

```
[42]: pd.merge(registrations,logins,how='left')
```

[42]:

	reg_id	name	log_id
0	1	Andrew	2.0
1	2	Bobo	4.0
2	3	Claire	NaN
3	4	David	NaN

25. The code pd.merge(registrations, logins, how='right') performs a **right join** between the registrations and logins DataFrames.
26. It keeps all rows from the logins DataFrame and matches rows from registrations based on the common "name" column.

```
[44]: pd.merge(registrations,logins,how='right')
```

[44]:

	reg_id	name	log_id
0	NaN	Xavier	1
1	1.0	Andrew	2
2	NaN	Yolanda	3
3	2.0	Bobo	4

27. The code pd.merge(registrations, logins, how='outer') performs a **full outer join** between the registrations and logins DataFrames.
28. It returns all rows from both DataFrames, matching them where the "name" values are the same.

```
[46]: pd.merge(registrations,logins,how='outer')
```

[46]:

	reg_id	name	log_id
0	1.0	Andrew	2.0
1	2.0	Bobo	4.0
2	3.0	Claire	NaN
3	4.0	David	NaN
4	NaN	Xavier	1.0
5	NaN	Yolanda	3.0

29. We are using combinations of left_on, right_on, left_index, right_index to merge a column or index with each other. The code demonstrates merging DataFrames using indices.
30. First, the `registrations` DataFrame is set to use the "name" column as its index with `registrations.set_index("name")`.
31. Then, `pd.merge(registrations, logins, left_index=True, right_on='name')` merges the two DataFrames using the "name" index from `registrations` and the "name" column from `logins`.
32. Similarly, `pd.merge(logins, registrations, right_index=True, left_on='name')` merges by using the "name" column from `logins` and the index from `registrations`.
33. These methods allow merging using index or column-based keys.

[48]:
```
registrations
```

[48]:

	reg_id	name
0	1	Andrew
1	2	Bobo
2	3	Claire
3	4	David

[50]:
```
logins
```

[50]:

	log_id	name
0	1	Xavier
1	2	Andrew
2	3	Yolanda
3	4	Bobo

[52]:
```
registrations = registrations.set_index("name")
```

[54]:
```
registrations
```

[54]:

	reg_id
name	
Andrew	1
Bobo	2
Claire	3
David	4

[56]:
```
pd.merge(registrations,logins,left_index=True,right_on='name')
```

[56]:

	reg_id	log_id	name
1	1	2	Andrew
3	2	4	Bobo

[58]:
```
pd.merge(logins,registrations,right_index=True,left_on='name')
```

[58]:

	log_id	name	reg_id
1	2	Andrew	1
3	4	Bobo	2

34. The code resets the index of the registrations DataFrame with registrations.reset_index(), turning the index back into a regular column.
35. Then, the columns are renamed to 'reg_name' and 'reg_id'.
36. pd.merge(registrations, logins, left_on='reg_name', right_on='name') performs a merge using the 'reg_name' column from registrations and the 'name' column from logins.
37. Finally, drop('reg_name', axis=1) removes the 'reg_name' column from the merged result.
38. These steps help combine and clean the DataFrames based on matching names while eliminating unnecessary columns.

[60]:
```
registrations = registrations.reset_index()
```

[62]:
```
registrations
```

[62]:

	name	reg_id
0	Andrew	1
1	Bobo	2
2	Claire	3
3	David	4

[64]:
```
logins
```

[64]:

	log_id	name
0	1	Xavier
1	2	Andrew
2	3	Yolanda
3	4	Bobo

[66]:
```
registrations.columns = ['reg_name','reg_id']
```

[68]:
```
registrations
```

[68]:

	reg_name	reg_id
0	Andrew	1
1	Bobo	2
2	Claire	3
3	David	4

[72]:
```
pd.merge(registrations,logins,left_on='reg_name',right_on='name')
```

[72]:

	reg_name	reg_id	log_id	name
0	Andrew	1	2	Andrew
1	Bobo	2	4	Bobo

[74]:
```
pd.merge(registrations,logins,left_on='reg_name',right_on='name').drop('reg_name',axis=1)
```

[74]:

	reg_id	log_id	name
0	1	2	Andrew
1	2	4	Bobo

39. The code renames the columns of registrations and logins to ensure they have the same column names: 'name' and 'id'.
40. Then, pd.merge(registrations, logins, on='name') merges the two DataFrames based on the common 'name' column, automatically adding suffixes like _x for columns from the left (registrations) and _y for columns from the right (logins).
41. The second merge, pd.merge(registrations, logins, on='name', suffixes=('_reg', '_log')), customizes the suffixes to '_reg' for registrations and '_log' for logins, avoiding column name conflicts.

[76]:
```
registrations.columns = ['name','id']
```

[78]:
```
logins.columns = ['id','name']
```

[80]:
```
registrations
```

[80]:

	name	id
0	Andrew	1
1	Bobo	2
2	Claire	3
3	David	4

[82]:
```
logins
```

[82]:

	id	name
0	1	Xavier
1	2	Andrew
2	3	Yolanda
3	4	Bobo

[84]:
```
# _x is for left
# _y is for right
pd.merge(registrations,logins,on='name')
```

[84]:

	name	id_x	id_y
0	Andrew	1	2
1	Bobo	2	4

[86]:
```
pd.merge(registrations,logins,on='name',suffixes=('_reg','_log'))
```

[86]:

	name	id_reg	id_log
0	Andrew	1	2
1	Bobo	2	4