

# **Data Wrangling: Join, Combine, and Reshape**

**Prof. Gheith Abandah**

# Reference

- **Chapter 8**
- Wes McKinney, **Python for Data Analysis**: Data Wrangling with Pandas, NumPy, and IPython, O'Reilly Media, 2nd Edition, 2018.
  - Material: <https://github.com/wesm/pypop-book>

# Data Wrangling: Join, Combine, and Reshape

- **Data wrangling** is the process of gathering, selecting, and transforming data to answer an analytical question.
- In many applications, **data** may be **spread** across a number of files or databases or be arranged in a **form** that is **not easy to analyze**.
- This topic focuses on **tools** to help **combine, join, and rearrange** data.

# Outline

8.1 Hierarchical Indexing

8.2 Combining and Merging Datasets

8.3 Reshaping and Pivoting

# Outline

## 8.1 Hierarchical Indexing

## 8.2 Combining and Merging Datasets

## 8.3 Reshaping and Pivoting

- Introduction
- Reordering and Sorting Levels
- Summary Statistics by Level
- Indexing with a DataFrame's columns

# 8.1 Hierarchical Indexing

- **Hierarchical indexing** is having multiple index levels on an axis.
- Allows you to **work with higher dimensional data** in a lower dimensional form.
- **Example:** Tracking population (millions) using pandas **Series**.

```
pop = pd.Series([34, 37, 19, 20, 21, 25],  
                index=[['CA', 'CA', 'NY',  
                        'NY', 'TX', 'TX'],  
                       [2000, 2010, 2000,  
                        2010, 2000, 2010]])
```

CA	2000	34
	2010	37
NY	2000	19
	2010	19
TX	2000	21
	2010	25

# 8.1 Hierarchical Indexing

- Try accessing:

- **Index**
- **Elements**
- **Slices**

```
pop.index
```

```
pop['CA', 2000]
```

```
pop['CA']
```

```
pop['NY': 'TX']
```

```
pop.loc[['CA', 'TX']]
```

```
pop.loc[:, 2010]
```

- Try **unstack** and **stack**.

```
pop.unstack()
```

```
pop.unstack().stack()
```

# 8.1 Hierarchical Indexing

- With a **DataFrame**, both axis can have a **hierarchical index**.

```
popf = pd.DataFrame(  
    [[34, 18, 16], [37, 19, 16],  
     [19, 10, 9], [20, 10, 10],  
     [21, 11, 10], [25, 13, 12]],  
    index=[[ 'CA', 'CA', 'NY',  
            'NY', 'TX', 'TX'],  
          [2000, 2010, 2000,  
           2010, 2000, 2010]],
```

```
columns=[[ 'Pop', 'Pop', 'Pop'],  
         [ 'Total', 'Male', 'Female']])
```

		Pop		
		Total	Male	Female
CA	2000	34	18	16
	2010	37	19	16
NY	2000	19	10	9
	2010	20	10	10
TX	2000	21	11	10
	2010	25	13	12



# 8.1 Hierarchical Indexing

- The **hierarchical levels** can have **names** (as strings or any Python objects).

- **Try:**

- `popf['Pop']`
- `popf['Pop', 'Total']['CA']`
- `popf['Pop', 'Total']['CA', 2000]`
- `popf.loc['CA', 2000]`

```
popf.index.names = ['State', 'Year']  
popf.columns.names = ['Key', 'Details']
```

```
Key          Pop  
Details      Total Male Female  
State Year  
CA          2000      34   18   16  
           2010      37   19   16  
NY          2000      19   10    9  
           2010      20   10   10  
TX          2000      21   11   10  
           2010      25   13   12
```

# Reordering and Sorting Levels

- You can **rearrange** the **order** of the **levels** on an axis.
- The **swaplevel** takes two level numbers or names and returns a new object with the levels interchanged.

```
popf.swaplevel('State', 'Year')
```

Key		Pop		
Details		Total	Male	Female
Year	State			
2000	CA	34	18	16
2010	CA	37	19	16
2000	NY	19	10	9
2010	NY	20	10	10
2000	TX	21	11	10
2010	TX	25	13	12

# Reordering and Sorting Levels

- You can **sort** the data by the values in one **specific level**.
- **sort\_index** sorts the data using only the values in a single level.
- Sorting provides better **performance**.

```
popf.swaplevel(  
    0, 1).sort_index(level=0)
```

Key		Pop		
Details		Total	Male	Female
Year	State			
2000	CA	34	18	16
	NY	19	10	9
	TX	21	11	10
2010	CA	37	19	16
	NY	20	10	10
	TX	25	13	12

# Summary Statistics by Level

- Many descriptive and summary statistics have a **level option** in which you can specify the level you want **to aggregate** by on a particular axis.

```
popf.min(level='State')
Key      Pop
Details Total Male Female
State
CA       34   18   16
NY       19   10    9
TX       21   11   10
```

```
popf.min(level='Key', axis=1)
Key      Pop
State Year
CA      2000   16
        2010   16
NY      2000    9
        2010   10
TX      2000   10
        2010   12
```

# Indexing with a DataFrame's columns

- You can convert some DataFrame columns to index by **set\_index**.

before				
	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

after			
	c	d	
	a	b	
one	0	0	7
	1	1	6
	2	2	5
two	0	3	4
	1	4	3
	2	5	2
	3	6	1

```
frame2 = frame.set_index(['c', 'd'])
```

The opposite function is:  
`frame2.reset_index()`

# Outline

8.1 Hierarchical Indexing

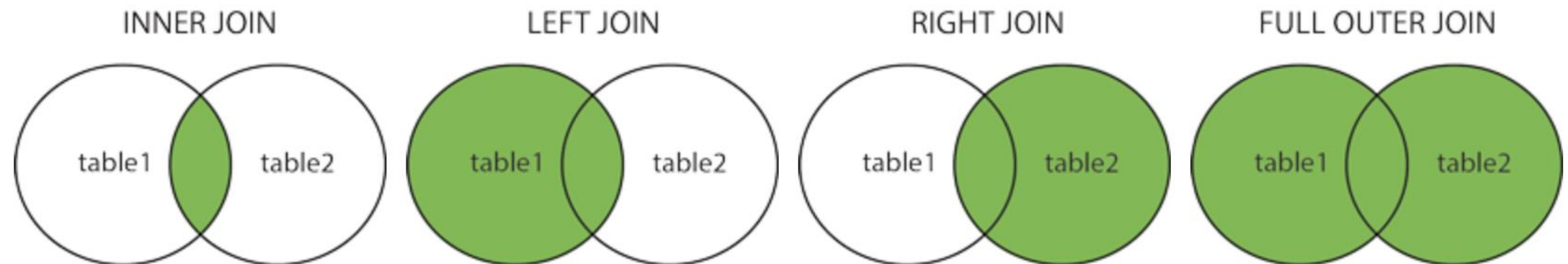
8.2 Combining and Merging  
Datasets

8.3 Reshaping and Pivoting

- Database-Style DataFrame Joins
- Merging on Index
- Concatenating Along an Axis
- Combining Data with Overlap

# Database-Style DataFrame Joins

- **Join** operations are **central** to **relational databases** (e.g., SQL-based).
- **Reference:** [https://www.w3schools.com/sql/sql\\_join.asp](https://www.w3schools.com/sql/sql_join.asp)



# Database-Style DataFrame Joins

- **Merge** or join operations **combine datasets** by linking rows using one or more keys.
- The **merge** function in pandas is the main entry point for using join algorithms on your data.
- **Many to one inner join** example:

df1		
	data1	key
0	0	b
1	1	b
2	2	a
3	3	c
4	4	a

df2		
	data2	key
0	0	a
1	1	b
2	2	d

```
pd.merge(df1, df2)
  key  data1  data2
0  b      0      1
1  b      1      1
2  a      2      0
3  a      4      0
```

Same result with:  
`pd.merge(df1, df2,  
on='key')`



# Database-Style DataFrame Joins

- If the **column names** are **different** in each object, you can **specify them** separately.

```
df3 = pd.DataFrame({'lkey': ['b',  
                             'c', 'a'], 'data1': range(3)})  
df4 = pd.DataFrame({'rkey': ['a',  
                             'b', 'd'], 'data2': range(3)})  
pd.merge(df3, df4, left_on='lkey',  
         right_on='rkey')
```

	lkey	data1	rkey	data2
0	b	0	b	1
1	a	2	a	0

# Database-Style DataFrame Joins

- In addition to the default **'inner'** join, you can specify:

- **'left'**
- **'right'**
- **'outer'**

```
pd.merge(df3, df4, left_on='lkey',  
         right_on='rkey',  
         how='outer')
```

	lkey	data1	rkey	data2
0	b	0.0	b	1.0
1	c	1.0	NaN	NaN
2	a	2.0	a	0.0
3	NaN	NaN	d	2.0

# Database-Style DataFrame Joins

- **Many-to-many** joins form the **Cartesian product** of the rows.

```
df1 = pd.DataFrame({'key': ['b',  
                             'b', 'a', 'c'],  
                   'data1': range(4)})  
df2 = pd.DataFrame({'key': ['a',  
                             'b', 'a', 'd'],  
                   'data2': range(4)})  
pd.merge(df1, df2, how='left')
```

	key	data1	data2
0	b	0	1.0
1	b	1	1.0
2	a	2	0.0
3	a	2	2.0
4	c	3	NaN

# Database-Style DataFrame Joins

- To merge with **multiple keys**, pass a list of column names.

```
left = pd.DataFrame(  
    {'key1': ['foo', 'foo', 'bar'],  
     'key2': ['one', 'two', 'one'],  
     'lval': [1, 2, 3]})  
right = pd.DataFrame(  
    {'key1': ['foo', 'foo', 'bar', 'bar'],  
     'key2': ['one', 'one', 'one', 'two'],  
     'rval': [4, 5, 6, 7]})  
pd.merge(left, right,  
         on=['key1', 'key2'], how='outer')
```

	key1	key2	lval	rval
0	foo	one	1.0	4.0
1	foo	one	1.0	5.0
2	foo	two	2.0	NaN
3	bar	one	3.0	6.0
4	bar	two	NaN	7.0

# Database-Style DataFrame Joins

- Pandas has default treatment of **overlapping column names**.
- You can also use the **suffixes** option for specifying strings to append to overlapping names in the left and right DataFrame objects.

```
pd.merge(left, right, on='key1')
  key1 key2_x  lval key2_y  rval
0  foo   one    1   one    4
...
```

```
pd.merge(left, right, on='key1',
         suffixes=('_left', '_right'))
  key1 key2_left  lval key2_right  rval
0  foo      one    1      one    4
...
```

# Merge function arguments

Argument	Description
<code>left</code>	DataFrame to be merged on the left side.
<code>right</code>	DataFrame to be merged on the right side.
<code>how</code>	One of 'inner', 'outer', 'left', or 'right'; defaults to 'inner'.
<code>on</code>	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in <code>left</code> and <code>right</code> as the join keys.
<code>left_on</code>	Columns in <code>left</code> DataFrame to use as join keys.
<code>right_on</code>	Analogous to <code>left_on</code> for <code>right</code> DataFrame.
<code>left_index</code>	Use row index in <code>left</code> as its join key (or keys, if a MultiIndex).
<code>right_index</code>	Analogous to <code>left_index</code> .
<code>sort</code>	Sort merged data lexicographically by join keys; <code>True</code> by default (disable to get better performance in some cases on large datasets).
<code>suffixes</code>	Tuple of string values to append to column names in case of overlap; defaults to ('_x', '_y') (e.g., if 'data' in both DataFrame objects, would appear as 'data_x' and 'data_y' in result).
<code>copy</code>	If <code>False</code> , avoid copying data into resulting data structure in some exceptional cases; by default always copies.
<code>indicator</code>	Adds a special column <code>_merge</code> that indicates the source of each row; values will be 'left_only', 'right_only', or 'both' based on the origin of the joined data in each row.

# Merging on Index

- The **merge key(s)** can be **in the index**.
- You can pass **left\_index=True** or **right\_index=True** (or both).

```
left
  key1  key2  data
0  Ohio  2000  0.0
1  Ohio  2001  1.0
2  Ohio  2002  2.0
3  Nevada 2001  3.0
4  Nevada 2002  4.0
```

```
right
      event1  event2
Nevada 2001      0      1
        2000      2      3
Ohio    2000      4      5
        2000      6      7
        2001      8      9
        2002     10     11
```

```
pd.merge(left, right,
         left_on=['key1', 'key2'],
         right_index=True)
```

```
   key1  key2  data  event1  event2
0  Ohio  2000  0.0      4      5
0  Ohio  2000  0.0      6      7
1  Ohio  2001  1.0      8      9
2  Ohio  2002  2.0     10     11
3  Nevada 2001  3.0      0      1
```

# Merging on Index

- When **indices are similar**, you can also use the **join** method.

	Ohio	Nevada
a	1.0	2.0
c	3.0	4.0
e	5.0	6.0

	Missouri	Alabama
b	7.0	8.0
c	9.0	10.0
d	11.0	12.0
e	13.0	14.0

```
left2.join(right2, how='outer')
```

	Ohio	Nevada	Missouri	Alabama
a	1.0	2.0	NaN	NaN
b	NaN	NaN	7.0	8.0
c	3.0	4.0	9.0	10.0
d	NaN	NaN	11.0	12.0
e	5.0	6.0	13.0	14.0

- You can also join **multiple** DataFrames at once.

```
left2.join([right2, right3])
```



# Concatenating Along an Axis

- **Concatenation, binding, or stacking** is supported by **concat** function.
- You can **glue** multiple **Series**.
- You can use keys to **identify** the **origin** of each entry.

s1	
a	0
b	1

s2	
c	2
d	3
e	4

s3	
f	5
g	6

```
s1 = pd.Series([0, 1], index=['a', 'b'])
s2 = pd.Series([2, 3, 4],
               index=['c', 'd', 'e'])
s3 = pd.Series([5, 6], index=['f', 'g'])
pd.concat([s1, s2, s3])
pd.concat([s1, s2, s3],
          keys=['one', 'two', 'three'])
```

one	a	0
	b	1
two	a	0
	b	1
three	f	5
	g	6

# Concatenating Along an Axis

- **concat** also work on **DataFrames**.

df1	one	two
a	0	1
b	2	3
c	4	5

df2	three	four
a	5	6
c	7	8

```
pd.concat([df1, df2], axis=1,
          keys=['level1', 'level2'])
```

	level1		level2	
	one	two	three	four
a	0	1	5.0	6.0
b	2	3	NaN	NaN
c	4	5	7.0	8.0

- When the row index has **no relevant data**, pass **ignore\_index=True**.

```
pd.concat([df1, df2], ignore_index=True)
```

Stacking

# Combining Data with Overlap

- When you want to **concatenate** two objects and **patching** the **NA** values from the caller object, use **combine\_first**.

```
df1.combine_first(df2)
```

df1	a	b	c
0	1.0	NaN	2
1	NaN	2.0	6
2	5.0	NaN	10
3	NaN	6.0	14

df2	a	b
0	5.0	NaN
1	4.0	3.0
2	NaN	4.0
3	3.0	6.0
4	7.0	8.0

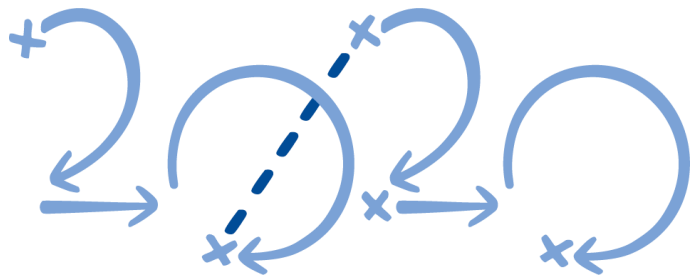
	a	b	c
0	1.0	NaN	2.0
1	4.0	2.0	6.0
2	5.0	4.0	10.0
3	3.0	6.0	14.0
4	7.0	8.0	NaN

# Outline

8.1 Hierarchical Indexing

8.2 Combining and Merging Datasets

8.3 Reshaping and Pivoting



- Reshaping with Hierarchical Indexing
- Pivoting “Long” to “Wide” Format
- Pivoting “Wide” to “Long” Format

# Reshaping with Hierarchical Indexing

- Pandas objects can be reshaped using:
  - **T** (returns a transposed view)
  - **stack**: “rotates” or pivots from the columns in the data to the rows.
  - **unstack**: pivots from the rows into the columns.
- **Example**: Find the totals over all states.

Key		Pop		
Details		Total	Male	Female
State	Year			
CA	2000	34	18	16
	2010	37	19	16
NY	2000	19	10	9
	2010	20	10	10
TX	2000	21	11	10
	2010	25	13	12

```
popf.unstack().sum()
```

# Reshaping with Hierarchical Indexing

- By default the **innermost** level is (un)stacked.
- When you unstack, the level unstacked becomes the innermost.
- You can (un)stack a different level by passing a level **number** or **name**.
- Unstacking might introduce **missing data**.

```
popf.unstack()
```

Key	Pop		Male		Female	
Details	Total					
Year	<u>2000</u>	<u>2010</u>	<u>2000</u>	<u>2010</u>	<u>2000</u>	<u>2010</u>
State						
CA	34	37	18	19	16	16
NY	19	20	10	10	9	10
TX	21	25	11	13	10	12

```
popf.unstack(0)
```

```
popf.unstack('State')
```

# Pivoting *Long* to *Wide* Format

- When you want to unstack based on keys (not index), use **pivot**.

**pivot(index,  
columns,  
values)**

```
data
  Name      Item  Value
0  Mohd      Age   22.00
1  Mohd     Height  1.75
2  Mohd     Weight  75.00
3  Tala      Age   21.00
4  Tala     Height  1.65
5  Tala     Weight  62.00
```

```
data.pivot('Name', 'Item', 'Value')
Item  Age  Height  Weight
Name
Mohd  22.0  1.75   75.0
Tala  21.0  1.65   62.0
```

# Pivoting *Wide* to *Long* Format

- When you want to stack based on keys (not index), use `melt`.

```
pd.melt(df,  
        id_vars,  
        value_vars)
```

```
data
```

	Dept	Exps	Sales	Serves
0	A	1200	3000	500
1	B	2300	400	2500

```
pd.melt(data, id_vars=['Dept'],  
        value_vars=['Sales', 'Serves'])
```

	Dept	variable	value
0	A	Sales	3000
1	B	Sales	400
2	A	Serves	500
3	B	Serves	2500



# Summary

8.1 Hierarchical Indexing

8.2 Combining and Merging Datasets

8.3 Reshaping and Pivoting