Introduction to Urban Data Science

CRP 4680/5680 (Spring 2025)

Week3 Data Management (II)

Wenzheng Li
Hazel (Yujin) Lee

OUTLINE

- 0. Quick Review
- 1. Data cleaning
  - Data Types
  - *NaN* Values
- 2. Joining multiple DataFrames
  - Concatenate Dataframes along the rows
  - Merging DataFrames along the columns
- 3. Slicing string columns

Review

0

df.columns

```
df_2012.columns
```

```
Index(['HouseID', 'CommunityID', 'TotalPrice', 'TransYear', 'Bedroom',
       'Livingroom', 'Bathroom', 'Size', 'FloorLevel', 'WinSouth',
       'WinSouthNorth', 'Decoration', 'TotalFloor', 'BuiltYear', 'Elevation',
       'Heating', 'TransMonth', 'TransDay', 'District', 'CensusTract',
       'XIAOQUWEB', 'SchQuality', 'NumSubway1km', 'Dist2Subway', 'HospQuality',
       'Dist2Hosp', 'NumHosp1km', 'NumBus200m', 'Dist2CBD', 'Dist2Center',
       'UnitPrice'],
      dtype='object')
```

df_2012.columns returns **an index** object, not a list. To get the corresponding index based on value:

- list(df_2012.columns).index(“HouseID”)
- df_2012.columns.tolist().index(“HouseID”)

Indexing and Slicing a Dataframe

how to select a subset of a Dataframe?

- ***Indexing***: simply selecting a particular row or column from a Dataframe.
- ***Slicing*****:** selecting some rows and some columns

- Three ways of selecting particular rows and columns of a Dataframe
  - df[ ]
  - df.loc[rows_***label*** , columns_***label***]
  - df.iloc[row_***position*** , column_***position***]

- A ***label***: one name in the column list or an index in the row index (the column at far left).
- A ***position***: the corresponding position of column name or index in a sequence, starting from zero.

Label and Position

df

Row Positions	Column Positions →	0	1	2	3	4	5	6	7	8	9	
	Row index (Labels) \ Column names (Labels)	HouseID	CommunityID	TotalPrice	TransYear	Bedroom	Livingroom	Bathroom	Size	FloorLevel	WinSouth	...
0	0	BJCP85139027	1735	2080032.42	2012	3	2	2	124.62	3	1	...
1	1	BJCY84814525	2023	1440023.27	2012	1	1	1	48.37	4	1	...
2	2	BJHD61617745									1	...
3	3	BJCY00382544									1	...
4	4	BJCY84554915									1	...

Task: select the first 2 rows and the first 2 columns

df.loc[row Labels , col Labels]

df.iloc[row Positions , col Positions]

`df.loc[ 0:1 , [ 'HouseID' , 'CommunityID' ]]`

`df.iloc[ 0:2 , 0:2 ]`

Filtering DataFrames

- `df.loc[df["Dist2Subway"] <= 1500, : ]`:
  - step1, `df["Dist2Subway"] <= 1500` return a series with values of ***False*** or ***True*** (boolean type); this is known as **boolean indexing**
  - step2, it is enclosed by `df.loc[]` and can return a subset of the candidate rows
  - step3, assign the returned DataFrame to a new dataframe called `df_subway`

```
df_2012["Dist2Subway"] <= 1500
```

```
0        True
1       False
2        True
3        True
4        True
         ...
```

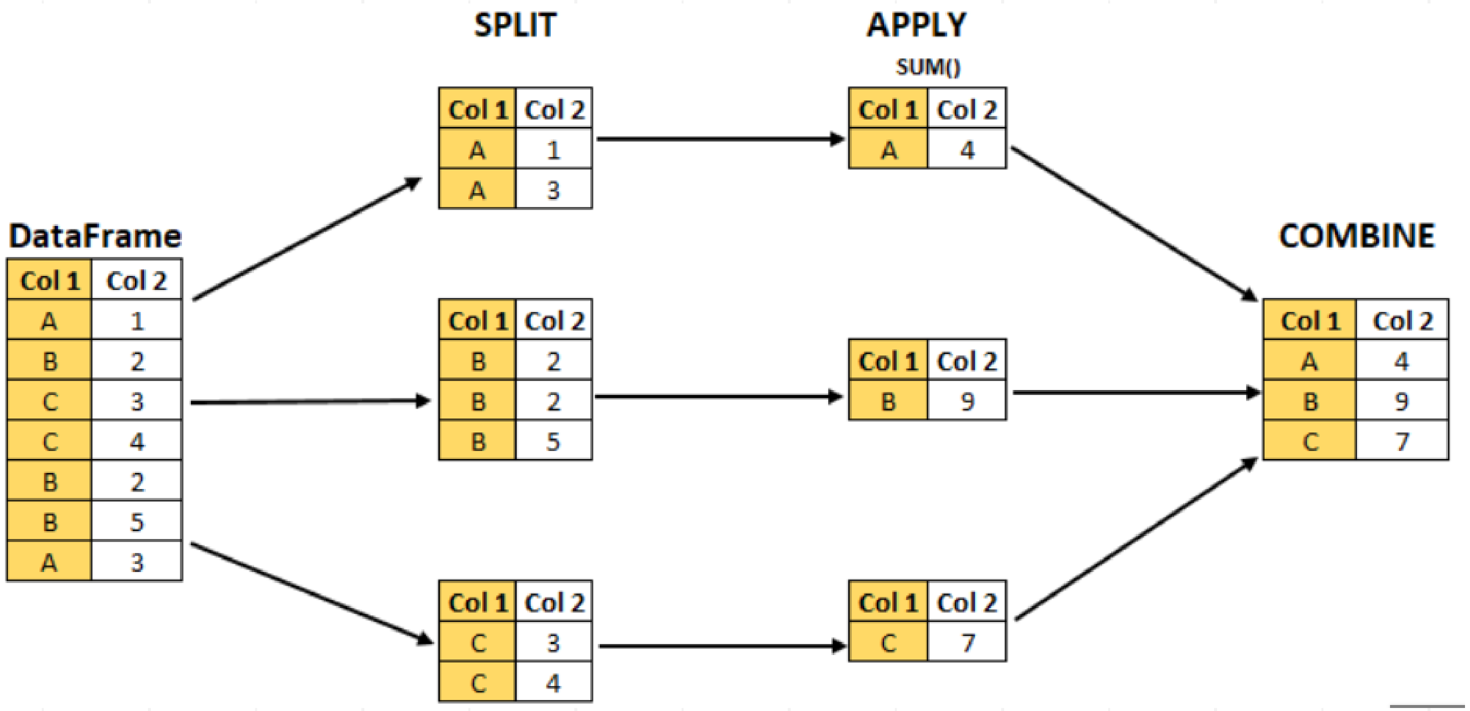
- In order to filter by more than one condition, you must:
  - Put all conditions in `()`
  - Separate the condtions by:
    - `|` if an `OR` condition
    - `&` if an `AND` condition

```
# houses within 1500m of subway stations with at least two Bathrooms
df_2012.loc[(df_2012["Dist2Subway"] <= 1500) & (df_2012["Bathroom"] >= 2) ]
```

- Here, you cannot use the `.iloc` function becuase it used positional indexing. Boolean indexing (e.g., `(df_2012["Dist2Subway"] <= 1500) & (df_2012["Bathroom"] >= 2))` is used to filter rows or columns based on conditions, and it works with `df.loc[]` or directly within square brackets `df[]`.

By “group by” we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria.
- **Applying** a function to each group independently.
- **Combining** the results into a data structure.



df.reset_index()

```
# .reset_index()
df_2012.groupby("Sublevel")[[ "UnitPrice" ]].mean()
```

	UnitPrice
Sublevel	
Level 1	26882.939484
Level 2	25937.768385
Level 3	20309.530499

drop: If True, the current index is removed and is not added as a column

```
# .reset_index()
df_2012.groupby("Sublevel")[[ "UnitPrice" ]].mean().reset_index()
```

	Sublevel	UnitPrice
0	Level 1	26882.939484
1	Level 2	25939.017190
2	Level 3	20328.317299

by default: **drop = False**

```
# .reset_index()
df_2012.groupby("Sublevel")[[ "UnitPrice" ]].mean().reset_index(drop = True)
```

	UnitPrice
0	26882.939484
1	25937.768385
2	20309.530499

by default: **drop = True**

```python
# groupby applied to UnitPrice
df_new = df_2012.groupby("Sublevel")[["UnitPrice"]].mean()
df_new
```

	UnitPrice
Sublevel	
Level 1	26882.939484
Level 2	25941.133662
Level 3	20328.317299

df.reset_index()

- inplace=True: Modifies the original DataFrame directly without returning a new DataFrame.
- inplace=False: Returns a new sorted DataFrame, leaving the original DataFrame unchanged.

```python
# .reset_index()
df_new.reset_index(inplace = False) # this is the default
df_new
```

	UnitPrice
Sublevel	
Level 1	26882.939484
Level 2	25941.133662
Level 3	20328.317299

```python
# .reset_index()
df_new2 = df_new.reset_index(inplace = False)
df_new2
```

	Sublevel	UnitPrice
0	Level 1	26882.939484
1	Level 2	25941.133662
2	Level 3	20328.317299

```python
# .reset_index()
df_new.reset_index(inplace = True)
df_new
```

	Sublevel	UnitPrice
0	Level 1	26882.939484
1	Level 2	25941.133662
2	Level 3	20328.317299

1 Data cleaning

Pandas dtype

- Each column/row in a Pandas (and GeoPandas) DataFrame has a data type, called *dtype* attribute.

- Here is the mapping between Pandas dtypes and python data types.

- Note that the object dtype means that the column is a mix of types or it's a string.

Pandas dtype	Python type	Usage
object	str or mixed	Text or mixed numeric and non-numeric values
int64	int	Integer numbers
float64	float	Floating point numbers
bool	bool	True/False values
datetime64	NA	Date and time values
timedelta[ns]	NA	Differences between two datetimes
category	NA	Finite list of text values

Null values

- Null values are when a particular value doesn’t exist in a cell.
- In Pandas, you might see three different types of null values appear;
  - NaN (Not a Number), None, NA (only rarely)

	Column_None	Column_NaN	Column_String	Column_NA
0	1	1.1	apple	1
1	NaN	NaN	banana	NA
2	3	3.3	None	3
3	4	4.4	cherry	4

Null values

- None means a missing entry, but it's not a numeric type. It is of type **object** and is often found in columns that contain strings or mixed data types.
- NaN (Not a Number) used by Pandas for representing missing data in numeric columns.
- Na is Pandas' newer, more flexible missing data indicator that can be used across different data types.)

	Column_None	Column_NaN	Column_String	Column_NA
0	1	1.1	apple	1
1	NaN	NaN	banana	NA
2	3	3.3	None	3
3	4	4.4	cherry	4

NaN for Missing Value

- *NaN* is used for representing missing data in numeric columns.
- The data type of *NaN* **is float**. even if the rest of the column contains integers.
- To detect *NaN*, Pandas provides the *.isna()* and *.notna()* functions.
- Some Pandas operations will generate *NaN*. For example, when we concatenate or merge two DataFrames with a different number of columns or keys, the missing columns or rows will be filled with *NaN*.

	2020-01-21	2020-01-22	2020-01-23	2020-01-24	2020-01-25	2020-01-26	2020-01-27	2020-01-28	2020-01-29	2020-01-30
Washington	1	1	1	1	1	1	1	1	1	1
Illinois	NaN	NaN	NaN	1	1	1	1	1	1	2
California	NaN	NaN	NaN	NaN	1	2	2	2	2	2
Arizona	NaN	NaN	NaN	NaN	NaN	1	1	1	1	1
Massachusetts	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NaN for Missing Value

- Removing data:
  - If it’s an important cell, we might remove the entire row the cell belongs to.
- Imputing data:
  - We might want to replace it with:
    - The most frequent value (mode), if we think that there's some default value
    - The median value (if you think there are outliers in the sample that might be skewing the mean)
    - The average value (if you don’t want the replaced data to influence your regression values).
  - Fill forward or backward: Fill missing values with the previous or next value (useful in time series data).
  - Or if you have more knowledge of the substantive topic (for ex: body temperature of mammals might typically be XX, but this species, it might be YY)
- Indicate that the data is missing in a new column
- Use linear regression or machine learning to predict the missing value.

Summary of Data Cleaning Steps and Exclusions

Table 5: Data Cleaning

Total Observation: 356,942

Irrelevant Category

Description	Count	Exclusion
Buildings height < 3 stories	-696	1-story or 2-story buildings
Non-flat unit	-1,534	dwellings with multiple levels, e.g., loft, duplex, split-level, and penthouse
Not 70-year property	-8,002	40-year and 50-year properties, which are non-residential
Not commodity property	-4,185	purchased public, affordable, state-owned, and all other non-commodity housing
Not normal residential property	-2,901	villa, apartment, courtyard, bungalow, commercial-mixed
Not basement	-954	

Total Observation: 338,670

Outliers

Description	Count	Exclusion
Per square meter price	-446	< 2,000 RMB/square meter
Total price	-1	< 100,000 RMB
Floor area	-98	< 20 square meter or > 400 square meter

Total Observation: 338,125

Missing Value

Description	Count	Exclusion
Floor Level	-917	missing
Elevator	-8,334	missing
Year Built	-4,920	missing
Heating	-3,181	missing

Total Observation: 320,773

Additional Drop

Description	Count	Exclusion
Outside 6th Ring Road	-16,232	observations outside the 6th ring road
More than 5 bedrooms	-876	observations with more than 5 bedrooms
More than 3 living rooms	-462	observations with more than 3 living rooms
More than 4 bathrooms	-117	observations with more than 4 bathrooms
Has no bathrooms	-112	observations with no bathrooms

Total Observation: 302,974

Li and Zhang (2024), Vertical Housing Gradients in Beijing. Under Review.

Joining multiple DataFrames

2

Concatenating multiple Dataframes along the row axis (axis = 0)

- concatenating along the rows: joining df2 to df1 vertically using pd.concat(axis=0)

- this means stacking your DataFrames on top of one another. If columns share the same names, they're combined into a single column; if not, new columns are created and filled with missing values.

df1

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

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

Result

		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7

2011

BOROUGH	NEIGHBORHOOD	BUILDING CLASS CATEGORY	TAX CLASS AT PRESENT	BLOCK	LOT	EASE-MENT	BUILDING CLASS AT PRESENT	ADDRESS
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2B	372	19		C7	292 EAST THIRD STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2A	377	53		C2	269 EAST 7TH STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	385	53		C4	234 EAST 2ND STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2B	386	63		D7	215 EAST 3RD STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	400	53		C1	209 EAST 4 STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	402	28		C4	168 EAST 7TH STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	405	5		C1	182 AVENUE A
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	406	9		C7	506 EAST 13 STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	406	42		C7	543 EAST 12TH STREET
1	ALPHABET CITY	08 RENTALS - ELEVATOR APARTMENTS	2	379	53		D1	EAST 9TH STREET

df_2011

2012

BOROUGH	NEIGHBORHOOD	BUILDING CLASS CATEGORY	TAX CLASS AT PRESENT	BLOCK	LOT	EASE-MENT	BUILDING CLASS AT PRESENT	ADDRESS
1	ALPHABET CITY	01 ONE FAMILY HOMES	1	372	38		S1	15 AVENUE D
1	ALPHABET CITY	01 ONE FAMILY HOMES	1	372	38		S1	15 AVENUE D
1	ALPHABET CITY	02 TWO FAMILY HOMES	1	376	32		S2	91 AVENUE D
1	ALPHABET CITY	03 THREE FAMILY HOMES	1	373	16		C0	326 EAST 4TH STREET
1	ALPHABET CITY	03 THREE FAMILY HOMES	1	393	9		C0	604 EAST 11TH STREET
1	ALPHABET CITY	04 TAX CLASS 1 CONDOS	1C	399	1102		R6	238 EAST 4TH STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	372	39		C7	11 AVENUE D
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	372	39		C7	11 AVENUE D
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2A	373	17		C3	328 EAST 4TH STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	385	38		C4	21-23 AVENUE C

df_2012

Combine Two DataFrames

BOROUGH	NEIGHBORHOOD	BUILDING CLASS CATEGORY	TAX CLASS AT PRESENT	BLOCK	LOT	EASE-MENT	BUILDING CLASS AT PRESENT	ADDRESS
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2B	372	19		C7	292 EAST THIRD STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2A	377	53		C2	269 EAST 7TH STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	385	53		C4	234 EAST 2ND STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2B	386	63		D7	215 EAST 3RD STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	400	53		C1	209 EAST 4 STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	402	28		C4	168 EAST 7TH STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	405	5		C1	182 AVENUE A
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	406	9		C7	506 EAST 13 STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	406	42		C7	543 EAST 12TH STREET
1	ALPHABET CITY	08 RENTALS - ELEVATOR APARTMENTS	2	379	53		D1	EAST 9TH STREET
1	ALPHABET CITY	01 ONE FAMILY HOMES	1	372	38		S1	15 AVENUE D
1	ALPHABET CITY	01 ONE FAMILY HOMES	1	372	38		S1	15 AVENUE D
1	ALPHABET CITY	02 TWO FAMILY HOMES	1	376	32		S2	91 AVENUE D
1	ALPHABET CITY	03 THREE FAMILY HOMES	1	373	16		C0	326 EAST 4TH STREET
1	ALPHABET CITY	03 THREE FAMILY HOMES	1	393	9		C0	604 EAST 11TH STREET
1	ALPHABET CITY	04 TAX CLASS 1 CONDOS	1C	399	1102		R6	238 EAST 4TH STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	372	39		C7	11 AVENUE D
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	372	39		C7	11 AVENUE D
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2A	373	17		C3	328 EAST 4TH STREET
1	ALPHABET CITY	07 RENTALS - WALKUP APARTMENTS	2	385	38		C4	21-23 AVENUE C

2011

2012

df_combine

df_combine = pd.concat([df_2011, df_2012])

Combine Two DataFrames

df_2011

	BOROUGH	NEIGHBORHOOD	BUILDINGCLASS	BLOCK	LOT	ADDRESS
0	1	ALPHABET CITY	07 RENTALS - WALKUP	372	19	292 EAST THIRD STREET
1	1	ALPHABET CITY	08 RENTALS - ELEVATOR	379	53	EAST 9TH STREET

df_2012

	BOROUGH	NEIGHBORHOOD	BUILDING_CLASS	BLOCK	LOT	PRESENT_ADDRESS
0	1	ALPHABET CITY	01 ONE FAMILY HOMES	372	38	15 AVENUE D
1	1	ALPHABET CITY	02 TWO FAMILY HOMES	376	32	91 AVENUE D

df_combine

	BOROUGH	NEIGHBORHOOD	BUILDINGCLASS	BLOCK	LOT	ADDRESS	BUILDING_CLASS	PRESENT_ADDRESS
0	1	ALPHABET CITY	07 RENTALS - WALKUP	372	19	292 EAST THIRD STREET	NaN	NaN
1	1	ALPHABET CITY	08 RENTALS - ELEVATOR	379	53	EAST 9TH STREET	NaN	NaN
0	1	ALPHABET CITY	NaN	372	38	NaN	01 ONE FAMILY HOMES	15 AVENUE D
1	1	ALPHABET CITY	NaN	376	32	NaN	02 TWO FAMILY HOMES	91 AVENUE D

columns that don’t match exactly become separate columns in the combined DF, leaving us with NaN values in places where the original DF had no corresponding data.

Combine Multiple DataFrames: Task

Let us combine the five housing datasets (2012 to 2016) together

initialization: df_combine ← Blank

Combine Multiple DataFrames: Task

Let us combine the five housing datasets (2012 to 2016) together

initialization: df_combine ← Blank

i = 2012: df_temp = housing data 2012

df_combine = df_temp

df_combine ← 2012

Combine Multiple DataFrames: Task

Let us combine the five housing datasets (2012 to 2016) together

initialization: df_combine ← Blank

i = 2012: df_temp = housing data 2012

df_combine = df_temp

df_combine ← 2012

i = 2013: df_temp = housing data 2013

2012

df_combine = pd.concat([df_combine, df_temp])

df_combine ← 2012, 2013

Combine Multiple DataFrames: Task

Let us combine the five housing datasets (2012 to 2016) together

initialization: df_combine ← Blank

i = 2012: df_temp = housing data 2012

df_combine = df_temp

df_combine ← 2012

i = 2013: df_temp = housing data 2013

2012

df_combine = pd.concat([df_combine, df_temp])

df_combine ← 2012, 2013

i = 2014: df_temp = housing data 2014

2012, 2013

df_combine = pd.concat([df_combine, df_temp])

df_combine ← 2012, 2013,2014

Merging Dataframes along the column

- **Merging along the columns** means merging DF B to DF A horizontally based on a **merge key** (the column (or set of columns) whose values are used to match rows across the two DFs.).
- Function: pd.merge( )
- **pd.concat()** can also be used to merge along columns by changing the argument axis = 1; **pd.merge()** can **ONLY** be used to merge along the columns.

left

	key	A	B
0	K0	A0	B0
1	K1	A1	B1
2	K2	A2	B2
3	K3	A3	B3

right

	key	C	D
0	K0	C0	D0
1	K1	C1	D1
2	K2	C2	D2
3	K3	C3	D3

Result

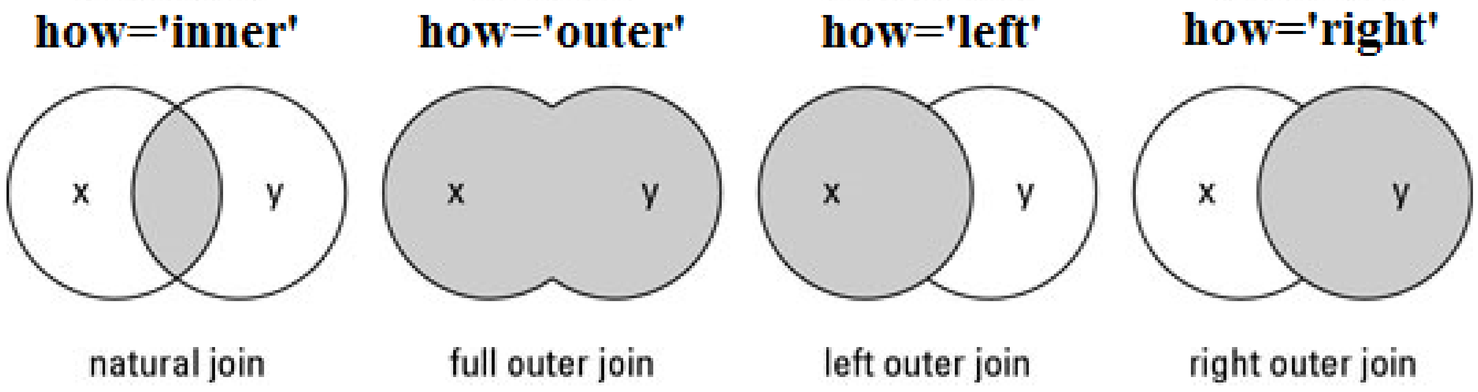
	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

Merging Dataframes along the column axis

pandas.DataFrame.merge

```
DataFrame.merge(right, how='inner', on=None, left_on=None, right_on=None,
left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'),
copy=None, indicator=False, validate=None)
```

- **Inner join** (how='inner'): Only rows with matching keys in both DataFrames are returned.
- **Outer join** (how='outer'): All rows from both DataFrames are returned, matched where possible, with missing values where not.
- **Left join** (how='left'): All rows from the left DataFrame are returned, matched with the right DataFrame where keys overlap, with missing values for unmatched right keys.
- **Right join** (how='right'): All rows from the right DataFrame are returned, matched with the left DataFrame where keys overlap, with missing values for unmatched left keys.



See different merge scenarios at: https://pandas.pydata.org/docs/user_guide/merging.html df_left

	key	value_left
0	1	A
1	2	B
2	3	C

df_right

	key	value_right
0	2	X
1	3	Y
2	4	Z

	key	value_left	value_right
0	2	B	X
1	3	C	Y

df_inner

	key	value_left	value_right
0	1	A	NaN
1	2	B	X
2	3	C	Y
3	4	NaN	Z

df_outer

	key	value_left	value_right
0	1	A	NaN
1	2	B	X
2	3	C	Y

df_left_join

	key	value_left	value_right
0	2	B	X
1	3	C	Y
2	4	NaN	Z

df_right_join

Q&A

df.shape

```
type(df_2012.shape)
```

```
tuple
```

```
T = ( 20, 'Jessa', 35.75, [30,60,90] )
      T[0]  T[1]   T[2]   T[3]
```

tuple:

- a type of data structure very similar to **lists**
- a collection that is ordered and **unchangeable**.
- Tuple items are indexed. e.g., df_2012.shape[0], df_2012.shape[1]