Comparison with SQL

Since many potential pandas users have some familiarity with [SQL](https://en.wikipedia.org/wiki/SQL), this page is meant to provide some examples of how various SQL operations would be performed using pandas.

If you’re new to pandas, you might want to first read through [10 Minutes to pandas](https://pandas.pydata.org/docs/user_guide/10min.html#min) to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

**In [1]: import** **pandas** **as** **pd**

**In [2]: import** **numpy** **as** **np**

Most of the examples will utilize the tips dataset found within pandas tests. We’ll read the data into a DataFrame called tips and assume we have a database table of the same name and structure.

**In [3]:** url = (

**...:**  "https://raw.github.com/pandas-dev"

**...:**  "/pandas/main/pandas/tests/io/data/csv/tips.csv"

**...:** )

**...:**

**In [4]:** tips = pd.read\_csv(url)

**In [5]:** tips

**Out[5]:**

total\_bill tip sex smoker day time size

0 16.99 1.01 Female No Sun Dinner 2

1 10.34 1.66 Male No Sun Dinner 3

2 21.01 3.50 Male No Sun Dinner 3

3 23.68 3.31 Male No Sun Dinner 2

4 24.59 3.61 Female No Sun Dinner 4

.. ... ... ... ... ... ... ...

239 29.03 5.92 Male No Sat Dinner 3

240 27.18 2.00 Female Yes Sat Dinner 2

241 22.67 2.00 Male Yes Sat Dinner 2

242 17.82 1.75 Male No Sat Dinner 2

243 18.78 3.00 Female No Thur Dinner 2

[244 rows x 7 columns]

Copies vs. in place operations

Most pandas operations return copies of the Series/DataFrame. To make the changes “stick”, you’ll need to either assign to a new variable:

sorted\_df = df.sort\_values("col1")

or overwrite the original one:

df = df.sort\_values("col1")

**Note**

You will see an inplace=True keyword argument available for some methods:

df.sort\_values("col1", inplace=**True**)

Its use is discouraged. [More information.](https://pandas.pydata.org/docs/user_guide/indexing.html" \l "indexing-view-versus-copy)

SELECT

In SQL, selection is done using a comma-separated list of columns you’d like to select (or a \* to select all columns):

**SELECT** total\_bill, tip, smoker, **time**

**FROM** tips;

With pandas, column selection is done by passing a list of column names to your DataFrame:

**In [6]:** tips[["total\_bill", "tip", "smoker", "time"]]

**Out[6]:**

total\_bill tip smoker time

0 16.99 1.01 No Dinner

1 10.34 1.66 No Dinner

2 21.01 3.50 No Dinner

3 23.68 3.31 No Dinner

4 24.59 3.61 No Dinner

.. ... ... ... ...

239 29.03 5.92 No Dinner

240 27.18 2.00 Yes Dinner

241 22.67 2.00 Yes Dinner

242 17.82 1.75 No Dinner

243 18.78 3.00 No Dinner

[244 rows x 4 columns]

Calling the DataFrame without the list of column names would display all columns (akin to SQL’s \*).

In SQL, you can add a calculated column:

**SELECT** \*, tip/total\_bill **as** tip\_rate

**FROM** tips;

With pandas, you can use the **[DataFrame.assign()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.assign.html" \l "pandas.DataFrame.assign" \o "pandas.DataFrame.assign)** method of a DataFrame to append a new column:

**In [7]:** tips.assign(tip\_rate=tips["tip"] / tips["total\_bill"])

**Out[7]:**

total\_bill tip sex smoker day time size tip\_rate

0 16.99 1.01 Female No Sun Dinner 2 0.059447

1 10.34 1.66 Male No Sun Dinner 3 0.160542

2 21.01 3.50 Male No Sun Dinner 3 0.166587

3 23.68 3.31 Male No Sun Dinner 2 0.139780

4 24.59 3.61 Female No Sun Dinner 4 0.146808

.. ... ... ... ... ... ... ... ...

239 29.03 5.92 Male No Sat Dinner 3 0.203927

240 27.18 2.00 Female Yes Sat Dinner 2 0.073584

241 22.67 2.00 Male Yes Sat Dinner 2 0.088222

242 17.82 1.75 Male No Sat Dinner 2 0.098204

243 18.78 3.00 Female No Thur Dinner 2 0.159744

[244 rows x 8 columns]

WHERE

Filtering in SQL is done via a WHERE clause.

**SELECT** \*

**FROM** tips

**WHERE** **time** = 'Dinner';

DataFrames can be filtered in multiple ways; the most intuitive of which is using [boolean indexing](https://pandas.pydata.org/docs/user_guide/indexing.html" \l "indexing-boolean).

**In [8]:** tips[tips["total\_bill"] > 10]

**Out[8]:**

total\_bill tip sex smoker day time size

0 16.99 1.01 Female No Sun Dinner 2

1 10.34 1.66 Male No Sun Dinner 3

2 21.01 3.50 Male No Sun Dinner 3

3 23.68 3.31 Male No Sun Dinner 2

4 24.59 3.61 Female No Sun Dinner 4

.. ... ... ... ... ... ... ...

239 29.03 5.92 Male No Sat Dinner 3

240 27.18 2.00 Female Yes Sat Dinner 2

241 22.67 2.00 Male Yes Sat Dinner 2

242 17.82 1.75 Male No Sat Dinner 2

243 18.78 3.00 Female No Thur Dinner 2

[227 rows x 7 columns]

The above statement is simply passing a Series of True/False objects to the DataFrame, returning all rows with True.

**In [9]:** is\_dinner = tips["time"] == "Dinner"

**In [10]:** is\_dinner

**Out[10]:**

0 True

1 True

2 True

3 True

4 True

...

239 True

240 True

241 True

242 True

243 True

Name: time, Length: 244, dtype: bool

**In [11]:** is\_dinner.value\_counts()

**Out[11]:**

True 176

False 68

Name: time, dtype: int64

**In [12]:** tips[is\_dinner]

**Out[12]:**

total\_bill tip sex smoker day time size

0 16.99 1.01 Female No Sun Dinner 2

1 10.34 1.66 Male No Sun Dinner 3

2 21.01 3.50 Male No Sun Dinner 3

3 23.68 3.31 Male No Sun Dinner 2

4 24.59 3.61 Female No Sun Dinner 4

.. ... ... ... ... ... ... ...

239 29.03 5.92 Male No Sat Dinner 3

240 27.18 2.00 Female Yes Sat Dinner 2

241 22.67 2.00 Male Yes Sat Dinner 2

242 17.82 1.75 Male No Sat Dinner 2

243 18.78 3.00 Female No Thur Dinner 2

[176 rows x 7 columns]

Just like SQL’s OR and AND, multiple conditions can be passed to a DataFrame using | (OR) and & (AND).

Tips of more than $5 at Dinner meals:

**SELECT** \*

**FROM** tips

**WHERE** **time** = 'Dinner' **AND** tip > 5.00;

**In [13]:** tips[(tips["time"] == "Dinner") & (tips["tip"] > 5.00)]

**Out[13]:**

total\_bill tip sex smoker day time size

23 39.42 7.58 Male No Sat Dinner 4

44 30.40 5.60 Male No Sun Dinner 4

47 32.40 6.00 Male No Sun Dinner 4

52 34.81 5.20 Female No Sun Dinner 4

59 48.27 6.73 Male No Sat Dinner 4

116 29.93 5.07 Male No Sun Dinner 4

155 29.85 5.14 Female No Sun Dinner 5

170 50.81 10.00 Male Yes Sat Dinner 3

172 7.25 5.15 Male Yes Sun Dinner 2

181 23.33 5.65 Male Yes Sun Dinner 2

183 23.17 6.50 Male Yes Sun Dinner 4

211 25.89 5.16 Male Yes Sat Dinner 4

212 48.33 9.00 Male No Sat Dinner 4

214 28.17 6.50 Female Yes Sat Dinner 3

239 29.03 5.92 Male No Sat Dinner 3

Tips by parties of at least 5 diners OR bill total was more than $45:

**SELECT** \*

**FROM** tips

**WHERE** **size** >= 5 **OR** total\_bill > 45;

**In [14]:** tips[(tips["size"] >= 5) | (tips["total\_bill"] > 45)]

**Out[14]:**

total\_bill tip sex smoker day time size

59 48.27 6.73 Male No Sat Dinner 4

125 29.80 4.20 Female No Thur Lunch 6

141 34.30 6.70 Male No Thur Lunch 6

142 41.19 5.00 Male No Thur Lunch 5

143 27.05 5.00 Female No Thur Lunch 6

155 29.85 5.14 Female No Sun Dinner 5

156 48.17 5.00 Male No Sun Dinner 6

170 50.81 10.00 Male Yes Sat Dinner 3

182 45.35 3.50 Male Yes Sun Dinner 3

185 20.69 5.00 Male No Sun Dinner 5

187 30.46 2.00 Male Yes Sun Dinner 5

212 48.33 9.00 Male No Sat Dinner 4

216 28.15 3.00 Male Yes Sat Dinner 5

NULL checking is done using the **[notna()](https://pandas.pydata.org/docs/reference/api/pandas.Series.notna.html" \l "pandas.Series.notna" \o "pandas.Series.notna)** and **[isna()](https://pandas.pydata.org/docs/reference/api/pandas.Series.isna.html" \l "pandas.Series.isna" \o "pandas.Series.isna)** methods.

**In [15]:** frame = pd.DataFrame(

**....:**  {"col1": ["A", "B", np.NaN, "C", "D"], "col2": ["F", np.NaN, "G", "H", "I"]}

**....:** )

**....:**

**In [16]:** frame

**Out[16]:**

col1 col2

0 A F

1 B NaN

2 NaN G

3 C H

4 D I

Assume we have a table of the same structure as our DataFrame above. We can see only the records where col2 IS NULL with the following query:

**SELECT** \*

**FROM** frame

**WHERE** col2 **IS** **NULL**;

**In [17]:** frame[frame["col2"].isna()]

**Out[17]:**

col1 col2

1 B NaN

Getting items where col1 IS NOT NULL can be done with **[notna()](https://pandas.pydata.org/docs/reference/api/pandas.Series.notna.html" \l "pandas.Series.notna" \o "pandas.Series.notna)**.

**SELECT** \*

**FROM** frame

**WHERE** col1 **IS** **NOT** **NULL**;

**In [18]:** frame[frame["col1"].notna()]

**Out[18]:**

col1 col2

0 A F

1 B NaN

3 C H

4 D I

GROUP BY

In pandas, SQL’s GROUP BY operations are performed using the similarly named **[groupby()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html" \l "pandas.DataFrame.groupby" \o "pandas.DataFrame.groupby)** method. **[groupby()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html" \l "pandas.DataFrame.groupby" \o "pandas.DataFrame.groupby)** typically refers to a process where we’d like to split a dataset into groups, apply some function (typically aggregation) , and then combine the groups together.

A common SQL operation would be getting the count of records in each group throughout a dataset. For instance, a query getting us the number of tips left by sex:

**SELECT** sex, **count**(\*)

**FROM** tips

**GROUP** **BY** sex;

*/\**

*Female 87*

*Male 157*

*\*/*

The pandas equivalent would be:

**In [19]:** tips.groupby("sex").size()

**Out[19]:**

sex

Female 87

Male 157

dtype: int64

Notice that in the pandas code we used [**size()**](https://pandas.pydata.org/docs/reference/api/pandas.core.groupby.DataFrameGroupBy.size.html#pandas.core.groupby.DataFrameGroupBy.size) and not **[count()](https://pandas.pydata.org/docs/reference/api/pandas.core.groupby.DataFrameGroupBy.count.html" \l "pandas.core.groupby.DataFrameGroupBy.count" \o "pandas.core.groupby.DataFrameGroupBy.count)**. This is because **[count()](https://pandas.pydata.org/docs/reference/api/pandas.core.groupby.DataFrameGroupBy.count.html" \l "pandas.core.groupby.DataFrameGroupBy.count" \o "pandas.core.groupby.DataFrameGroupBy.count)** applies the function to each column, returning the number of NOT NULL records within each.

**In [20]:** tips.groupby("sex").count()

**Out[20]:**

total\_bill tip smoker day time size

sex

Female 87 87 87 87 87 87

Male 157 157 157 157 157 157

Alternatively, we could have applied the **[count()](https://pandas.pydata.org/docs/reference/api/pandas.core.groupby.DataFrameGroupBy.count.html" \l "pandas.core.groupby.DataFrameGroupBy.count" \o "pandas.core.groupby.DataFrameGroupBy.count)** method to an individual column:

**In [21]:** tips.groupby("sex")["total\_bill"].count()

**Out[21]:**

sex

Female 87

Male 157

Name: total\_bill, dtype: int64

Multiple functions can also be applied at once. For instance, say we’d like to see how tip amount differs by day of the week - **agg()** allows you to pass a dictionary to your grouped DataFrame, indicating which functions to apply to specific columns.

**SELECT** **day**, **AVG**(tip), **COUNT**(\*)

**FROM** tips

**GROUP** **BY** **day**;

*/\**

*Fri 2.734737 19*

*Sat 2.993103 87*

*Sun 3.255132 76*

*Thu 2.771452 62*

*\*/*

**In [22]:** tips.groupby("day").agg({"tip": np.mean, "day": np.size})

**Out[22]:**

tip day

day

Fri 2.734737 19

Sat 2.993103 87

Sun 3.255132 76

Thur 2.771452 62

Grouping by more than one column is done by passing a list of columns to the **[groupby()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html" \l "pandas.DataFrame.groupby" \o "pandas.DataFrame.groupby)** method.

**SELECT** smoker, **day**, **COUNT**(\*), **AVG**(tip)

**FROM** tips

**GROUP** **BY** smoker, **day**;

*/\**

*smoker day*

*No Fri 4 2.812500*

*Sat 45 3.102889*

*Sun 57 3.167895*

*Thu 45 2.673778*

*Yes Fri 15 2.714000*

*Sat 42 2.875476*

*Sun 19 3.516842*

*Thu 17 3.030000*

*\*/*

**In [23]:** tips.groupby(["smoker", "day"]).agg({"tip": [np.size, np.mean]})

**Out[23]:**

tip

size mean

smoker day

No Fri 4 2.812500

Sat 45 3.102889

Sun 57 3.167895

Thur 45 2.673778

Yes Fri 15 2.714000

Sat 42 2.875476

Sun 19 3.516842

Thur 17 3.030000

JOIN

JOINs can be performed with **[join()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.join.html" \l "pandas.DataFrame.join" \o "pandas.DataFrame.join)** or **[merge()](https://pandas.pydata.org/docs/reference/api/pandas.merge.html" \l "pandas.merge" \o "pandas.merge)**. By default, **[join()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.join.html" \l "pandas.DataFrame.join" \o "pandas.DataFrame.join)** will join the DataFrames on their indices. Each method has parameters allowing you to specify the type of join to perform (LEFT, RIGHT, INNER, FULL) or the columns to join on (column names or indices).

**Warning**

If both key columns contain rows where the key is a null value, those rows will be matched against each other. This is different from usual SQL join behaviour and can lead to unexpected results.

**In [24]:** df1 = pd.DataFrame({"key": ["A", "B", "C", "D"], "value": np.random.randn(4)})

**In [25]:** df2 = pd.DataFrame({"key": ["B", "D", "D", "E"], "value": np.random.randn(4)})

Assume we have two database tables of the same name and structure as our DataFrames.

Now let’s go over the various types of JOINs.

INNER JOIN

**SELECT** \*

**FROM** df1

**INNER** **JOIN** df2

**ON** df1.**key** = df2.**key**;

# merge performs an INNER JOIN by default

**In [26]:** pd.merge(df1, df2, on="key")

**Out[26]:**

key value\_x value\_y

0 B -0.282863 1.212112

1 D -1.135632 -0.173215

2 D -1.135632 0.119209

[**merge()**](https://pandas.pydata.org/docs/reference/api/pandas.merge.html#pandas.merge) also offers parameters for cases when you’d like to join one DataFrame’s column with another DataFrame’s index.

**In [27]:** indexed\_df2 = df2.set\_index("key")

**In [28]:** pd.merge(df1, indexed\_df2, left\_on="key", right\_index=**True**)

**Out[28]:**

key value\_x value\_y

1 B -0.282863 1.212112

3 D -1.135632 -0.173215

3 D -1.135632 0.119209

LEFT OUTER JOIN

Show all records from df1.

**SELECT** \*

**FROM** df1

**LEFT** **OUTER** **JOIN** df2

**ON** df1.**key** = df2.**key**;

**In [29]:** pd.merge(df1, df2, on="key", how="left")

**Out[29]:**

key value\_x value\_y

0 A 0.469112 NaN

1 B -0.282863 1.212112

2 C -1.509059 NaN

3 D -1.135632 -0.173215

4 D -1.135632 0.119209

RIGHT JOIN

Show all records from df2.

**SELECT** \*

**FROM** df1

**RIGHT** **OUTER** **JOIN** df2

**ON** df1.**key** = df2.**key**;

**In [30]:** pd.merge(df1, df2, on="key", how="right")

**Out[30]:**

key value\_x value\_y

0 B -0.282863 1.212112

1 D -1.135632 -0.173215

2 D -1.135632 0.119209

3 E NaN -1.044236

FULL JOIN

pandas also allows for FULL JOINs, which display both sides of the dataset, whether or not the joined columns find a match. As of writing, FULL JOINs are not supported in all RDBMS (MySQL).

Show all records from both tables.

**SELECT** \*

**FROM** df1

**FULL** **OUTER** **JOIN** df2

**ON** df1.**key** = df2.**key**;

**In [31]:** pd.merge(df1, df2, on="key", how="outer")

**Out[31]:**

key value\_x value\_y

0 A 0.469112 NaN

1 B -0.282863 1.212112

2 C -1.509059 NaN

3 D -1.135632 -0.173215

4 D -1.135632 0.119209

5 E NaN -1.044236

UNION

UNION ALL can be performed using **[concat()](https://pandas.pydata.org/docs/reference/api/pandas.concat.html" \l "pandas.concat" \o "pandas.concat)**.

**In [32]:** df1 = pd.DataFrame(

**....:**  {"city": ["Chicago", "San Francisco", "New York City"], "rank": range(1, 4)}

**....:** )

**....:**

**In [33]:** df2 = pd.DataFrame(

**....:**  {"city": ["Chicago", "Boston", "Los Angeles"], "rank": [1, 4, 5]}

**....:** )

**....:**

**SELECT** city, rank

**FROM** df1

**UNION** **ALL**

**SELECT** city, rank

**FROM** df2;

*/\**

*city rank*

*Chicago 1*

*San Francisco 2*

*New York City 3*

*Chicago 1*

*Boston 4*

*Los Angeles 5*

*\*/*

**In [34]:** pd.concat([df1, df2])

**Out[34]:**

city rank

0 Chicago 1

1 San Francisco 2

2 New York City 3

0 Chicago 1

1 Boston 4

2 Los Angeles 5

SQL’s UNION is similar to UNION ALL, however UNION will remove duplicate rows.

**SELECT** city, rank

**FROM** df1

**UNION**

**SELECT** city, rank

**FROM** df2;

*-- notice that there is only one Chicago record this time*

*/\**

*city rank*

*Chicago 1*

*San Francisco 2*

*New York City 3*

*Boston 4*

*Los Angeles 5*

*\*/*

In pandas, you can use **[concat()](https://pandas.pydata.org/docs/reference/api/pandas.concat.html" \l "pandas.concat" \o "pandas.concat)** in conjunction with **[drop\_duplicates()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.drop_duplicates.html" \l "pandas.DataFrame.drop_duplicates" \o "pandas.DataFrame.drop_duplicates)**.

**In [35]:** pd.concat([df1, df2]).drop\_duplicates()

**Out[35]:**

city rank

0 Chicago 1

1 San Francisco 2

2 New York City 3

1 Boston 4

2 Los Angeles 5

LIMIT

**SELECT** \* **FROM** tips

**LIMIT** 10;

**In [36]:** tips.head(10)

**Out[36]:**

total\_bill tip sex smoker day time size

0 16.99 1.01 Female No Sun Dinner 2

1 10.34 1.66 Male No Sun Dinner 3

2 21.01 3.50 Male No Sun Dinner 3

3 23.68 3.31 Male No Sun Dinner 2

4 24.59 3.61 Female No Sun Dinner 4

5 25.29 4.71 Male No Sun Dinner 4

6 8.77 2.00 Male No Sun Dinner 2

7 26.88 3.12 Male No Sun Dinner 4

8 15.04 1.96 Male No Sun Dinner 2

9 14.78 3.23 Male No Sun Dinner 2

pandas equivalents for some SQL analytic and aggregate functions

Top n rows with offset

*-- MySQL*

**SELECT** \* **FROM** tips

**ORDER** **BY** tip **DESC**

**LIMIT** 10 **OFFSET** 5;

**In [37]:** tips.nlargest(10 + 5, columns="tip").tail(10)

**Out[37]:**

total\_bill tip sex smoker day time size

183 23.17 6.50 Male Yes Sun Dinner 4

214 28.17 6.50 Female Yes Sat Dinner 3

47 32.40 6.00 Male No Sun Dinner 4

239 29.03 5.92 Male No Sat Dinner 3

88 24.71 5.85 Male No Thur Lunch 2

181 23.33 5.65 Male Yes Sun Dinner 2

44 30.40 5.60 Male No Sun Dinner 4

52 34.81 5.20 Female No Sun Dinner 4

85 34.83 5.17 Female No Thur Lunch 4

211 25.89 5.16 Male Yes Sat Dinner 4

Top n rows per group

*-- Oracle's ROW\_NUMBER() analytic function*

**SELECT** \* **FROM** (

**SELECT**

t.\*,

ROW\_NUMBER() OVER(PARTITION **BY** **day** **ORDER** **BY** total\_bill **DESC**) **AS** rn

**FROM** tips t

)

**WHERE** rn < 3

**ORDER** **BY** **day**, rn;

**In [38]:** (

**....:**  tips.assign(

**....:**  rn=tips.sort\_values(["total\_bill"], ascending=**False**)

**....:**  .groupby(["day"])

**....:**  .cumcount()

**....:**  + 1

**....:**  )

**....:**  .query("rn < 3")

**....:**  .sort\_values(["day", "rn"])

**....:** )

**....:**

**Out[38]:**

total\_bill tip sex smoker day time size rn

95 40.17 4.73 Male Yes Fri Dinner 4 1

90 28.97 3.00 Male Yes Fri Dinner 2 2

170 50.81 10.00 Male Yes Sat Dinner 3 1

212 48.33 9.00 Male No Sat Dinner 4 2

156 48.17 5.00 Male No Sun Dinner 6 1

182 45.35 3.50 Male Yes Sun Dinner 3 2

197 43.11 5.00 Female Yes Thur Lunch 4 1

142 41.19 5.00 Male No Thur Lunch 5 2

the same using rank(method='first') function

**In [39]:** (

**....:**  tips.assign(

**....:**  rnk=tips.groupby(["day"])["total\_bill"].rank(

**....:**  method="first", ascending=**False**

**....:**  )

**....:**  )

**....:**  .query("rnk < 3")

**....:**  .sort\_values(["day", "rnk"])

**....:** )

**....:**

**Out[39]:**

total\_bill tip sex smoker day time size rnk

95 40.17 4.73 Male Yes Fri Dinner 4 1.0

90 28.97 3.00 Male Yes Fri Dinner 2 2.0

170 50.81 10.00 Male Yes Sat Dinner 3 1.0

212 48.33 9.00 Male No Sat Dinner 4 2.0

156 48.17 5.00 Male No Sun Dinner 6 1.0

182 45.35 3.50 Male Yes Sun Dinner 3 2.0

197 43.11 5.00 Female Yes Thur Lunch 4 1.0

142 41.19 5.00 Male No Thur Lunch 5 2.0

*-- Oracle's RANK() analytic function*

**SELECT** \* **FROM** (

**SELECT**

t.\*,

RANK() OVER(PARTITION **BY** sex **ORDER** **BY** tip) **AS** rnk

**FROM** tips t

**WHERE** tip < 2

)

**WHERE** rnk < 3

**ORDER** **BY** sex, rnk;

Let’s find tips with (rank < 3) per gender group for (tips < 2). Notice that when using rank(method='min') function rnk\_min remains the same for the same tip (as Oracle’s RANK() function)

**In [40]:** (

**....:**  tips[tips["tip"] < 2]

**....:**  .assign(rnk\_min=tips.groupby(["sex"])["tip"].rank(method="min"))

**....:**  .query("rnk\_min < 3")

**....:**  .sort\_values(["sex", "rnk\_min"])

**....:** )

**....:**

**Out[40]:**

total\_bill tip sex smoker day time size rnk\_min

67 3.07 1.00 Female Yes Sat Dinner 1 1.0

92 5.75 1.00 Female Yes Fri Dinner 2 1.0

111 7.25 1.00 Female No Sat Dinner 1 1.0

236 12.60 1.00 Male Yes Sat Dinner 2 1.0

237 32.83 1.17 Male Yes Sat Dinner 2 2.0

UPDATE[¶](https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_sql.html#update)

**UPDATE** tips

**SET** tip = tip\*2

**WHERE** tip < 2;

**In [41]:** tips.loc[tips["tip"] < 2, "tip"] \*= 2

DELETE

**DELETE** **FROM** tips

**WHERE** tip > 9;

In pandas we select the rows that should remain instead of deleting them:

**In [42]:** tips = tips.loc[tips["tip"] <= 9]