How to use Pandas for SQL-like actions?

Hessam Mohammad Hosseini

January 3, 2023

Abstract

This document is part 1 of my cheat sheet on **Pandas** which provide overall review on how to use **Pandas** for those familiar with **SQL**.

Contents

In this cheetsheet, I try to make readable and easy to use reference for SQL users aiming to explain how to do similar action in Pandas. My assumption is you know how to have your data as dataframe in Pandas. As soon as you have a dataframe, you can query like a table in SQL. Many possibilities are avaiable IO Tools (CSV, EXCEL,DB connection . . .)but most important ones are reading from Excel and CSV.

For example, following line of code will read file located at `file_address`, skip first row, consider `Time` column to be parsed as `datetime` format and consider`,` as thousands identifer while reading file.

```
df = pd.read_csv(file_name,
                 skiprows=10, # ignore first 10 rows of file
                 parse_dates=['Time'], # Parse Time as datetime column
                 thousands=',') # numbers have , to demonstrate 000 sepration
```

1. read_csv Function has lots of useful options to fasilitate work and avoid doing extra data cleaning tasks after data loading. Here are options for `read_csv`.

   ```
   df = pandas.read_csv(filepath_or_buffer, sep=', ', delimiter=None,
   header='infer', names=None, index_col=None, usecols=None,
   squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None,
   engine=None, converters=None, true_values=None, false_values=None,
   skipinitialspace=False, skiprows=None, skipfooter=0, nrows=None,
   na_values=None, keep_default_na=True, na_filter=True, verbose=False,
   skip_blank_lines=True, parse_dates=False, infer_datetime_format=False,
   keep_date_col=False, date_parser=None, dayfirst=False, iterator=False,
   chunksize=None, compression='infer', thousands=None, decimal=b'.',
   lineterminator=None, quotechar='"', quoting=0, doublequote=True,
   escapechar=None, comment=None, encoding=None, dialect=None,
   tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True,
   delim_whitespace=False, low_memory=True, memory_map=False,
   float_precision=None)
   ```

2. read_excel Functions has lots of useful options to fasilitate work. Here are options for `read_excel`:

   ```
   df = pandas.read_excel(io, sheet_name=0, header=0, names=None,
   index_col=None, parse_cols=None, usecols=None, squeeze=False, dtype=None,
   engine=None, converters=None, true_values=None, false_values=None,
   skiprows=None, nrows=None, na_values=None, keep_default_na=True,
   verbose=False, parse_dates=False, date_parser=None, thousands=None,
   comment=None, skip_footer=0, skipfooter=0, convert_float=True,
   mangle_dupe_cols=True, **kwds)
   ```

I tried to summerized and add to what was available on Pandas comparison with SQL aiming to simplify understanding. For details on functionality, please check and review Pandas documentation.

1 SELECT

SQL Sample	Pandas Sample
select * FROM table	table
select distinct c5 FROM table	table['c5'].unique() or table[['c5']].drop_duplicates()
select c1, c2,c10 FROM table	df[[c1, c2,c10]]
select c10, c2, c1 FROM table	df[[c10, c1, c2]]
select c10, c2*12 - c1 + c6 FROM table	df['new c'] = df.c2*12 - df.c1 + df.c6df[[c10,'new c']] or df.assign(c10 = df.c10, new_c = df.c2*12 - df.c1 + df.c6)
SELECT total_bill, tip, smoker, time FROM tips LIMIT 5	tips[['total_bill', 'tip', 'smoker','time']].head(5)

1.1 Update or delete

	SQL Sample	Pandas Sample

Delete	DELETE
	FROM tips
	WHERE tip > 9
		tips = tips.loc[tips['tip'] <= 9]

Update	UPDATE tips
	SET tip = tip*2
	WHERE tip < 2
		tips.loc[tips['tip'] < 2, 'tip'] *= 2

2 Conditioning at WHERE

	SQL Sample	Pandas Sample

-	SQL SELECT *
	FROM tips
	WHERE time = 'Dinner'
	LIMIT 5
		tips[tips['time'] == 'Dinner'].head(5) or is_dinner = tips['time'] == 'Dinner'tips[is_dinner].head(5)

AND	SELECT *
	FROM tips
	WHERE time = 'Dinner' AND tip > 5.00
		tips[(tips['time'] == 'Dinner') & (tips['tip'] > 5.00)]

	SQL Sample	Pandas Sample

OR	`SELECT *`
	`FROM tips`
	`WHERE size >= 5 OR total_bill > 45`
		`tips[(tips['size'] >= 5') or (tips['total_bill'] > 45)]`

IS NULL	`SELECT *`
	`FROM t`
	`WHERE col2 IS NULL`
		`t[t['col2'].isna()]`

IS NOT NULL	`SELECT *`
	`FROM t`
	`WHERE col IS NOT NULL`
		`t[t['col2'].notna()]`

SQL Sample	Pandas Sample

`SELECT *`
`FROM tips`
`ORDER BY tip DESC`
`LIMIT 10 OFFSET 5`
	`tips.nlargest(10 + 5, columns='tip').tail(10)`

`SELECT total_bill, tip, smoker, time`
`FROM tips`
`ORDER BY tip DESC`
`LIMIT 10 OFFSET 5`
	`tips[['total_bill', 'tip', 'smoker','time']]`
	`tips.nlargest(10 + 5, columns='tip').tail(10)`

3 GROUP BY

SQL Sample	Pandas Sample

`SELECT sex, count(*)`
`FROM tips`
`GROUP BY sex`
`select A, sum(C), sum(D)`
`FROM df`
`GROUP BY A`
	`tips.groupby('sex').size()`
	or
	`tips.groupby('sex')['total_bill'].count()`
	`df.groupby('A')['B','C'].sum()`

SQL Sample	Pandas Sample

`SELECT day, AVG(tip), COUNT(*)`
`FROM tips`
`GROUP BY day`
	`tips.groupby('day').agg({'tip': np.mean, 'day': np.size})`

SQL Sample	Pandas Sample

`SELECT smoker, day,`
`COUNT(*), AVG(tip)`
`FROM tips`
`GROUP BY smoker, day`
	`tips.groupby(['smoker,'day']).agg({'tip': [np.size, np.mean]})`

SQL Sample	Pandas Sample

`SELECT c1, COUNT(*)`
`FROM df`
`where country='IR'`
`GROUP BY c1`
`having count(*)>1000`
	`df[df.country == 'IR'].groupby('c1').filter(lambda g: len(g) > 1000).groupby('c1').size()`

`SELECT c1, COUNT(*)`
`FROM df`
`WHERE country='IR'`
`GROUP BY c1`
`HAVING count(*)>1000`
`ORDER BY count(*) desc`
	`df[df.country == 'IR'].groupby('c1').filter(lambda g: len(g) > 1000).groupby('c1').size().sort_values(ascending=False)`

4 ORDER BY

SQL Sample	Pandas Sample

`SELECT *`
`FROM df`
`ORDER BY A, B`
	`df.sort_values(['A', 'B'])`

`SELECT *`
`FROM df`
`ORDER BY A desc, C`
	`df.sort_values(['A', 'B'],ascending=[False, True])`

5 UNION, JOIN and other set related operations

I will work to provide more comprehensive explanations on this part.

5.1 Union

SQL Sample	Pandas Sample

`SELECT c1, c2`
`FROM df1`
`UNION ALL`
`SELECT c1, c2`
`FROM df2`
	`pd.concat([df1, df2])`

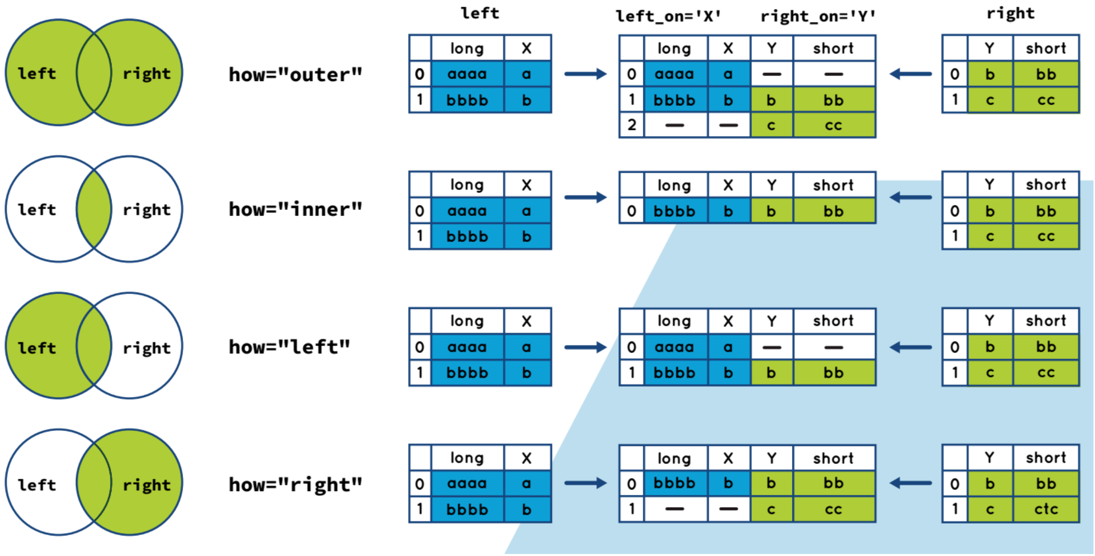
Difference between `union all` and `union` is that `union` will remove duplicates.

SQL Sample	Pandas Sample

`SELECT c1, c2`
`FROM df1`
`UNION`
`SELECT c1, c2`
`FROM df2`
	`pd.concat([df1, df2])`
	`.drop_duplicates()`

5.2 Different Join cases

I will add parts to make eplanation on `join` more clear and compehensive. Below image extracted from Enthoughtnamed "Enthought-Python-Pandas-Cheat-Sheets-1-8-v1.0.2" worth more than 100 sentences to explain different types of `join`. You can get whole file via registration on Enthought.



SQL Sample	Pandas Sample

`SELECT *`
`FROM df1`
`INNER JOIN df2`
`ON df1.key = df2.key`
	`pd.merge(df1, df2,`
	`on='key')`

`SELECT *`
`FROM df1`
`INNER JOIN df2`
`ON df1.c7 = df2.c5`
	`pd.merge(df1, df2,`
	`left_on='c7',right_on='c5')`

6 Time functionality

In order to have possibility to use time related functionalities, we need help `Pandas` understand which columns are to be treated as time. Of course, the columns should be in for that converting them to time format is possible. For details, please check and review Pandas documentation. If you manage to let `Pandas` know properly which column(s) to be time related column(s), they will end up having `datetime64[ns]` format. `.dtypes` on dataframe provides you with columns formats. Pass date related column(s) you need to `parse_dates` to `read_csv` or `read_excel` functions. Check Pandas documentation for more details.

Doing so, you can apply .dt on column to have date - time selection like

```
dt.dayofweek

dt.minute

dt.hour

dt.second

dt.quarter

dt.month

dt.month_name

dt.weekday_name

dt.weekday

dt.weekofyear

dt.year
```

6.1 How to get current date time using pandas?

```
pd.datetime.now()

pd.datetime.now().date()

pd.datetime.now().year

pd.datetime.now().month

pd.datetime.now().day

pd.datetime.now().hour

pd.datetime.now().minute

pd.datetime.now().second

pd.datetime.now().microsecond
```

Again, check Pandas documentation for more! Here, we assume sdate column to have datetime64[ns] format.

	SQL Sample	Pandas Sample

sysdate - n	SELECT *
	FROM df
	WHERE sdate> sysdate-5
		df[df['sdate'].dt.date()> pd.datetime.now().date()-5]

month	SELECT *
	FROM df
	WHERE sdate in Q1
		df[(df.sdate.dt.month >= 1) & (df.sdate.dt.month <= 3)]

	SQL Sample	Pandas Sample

between	SELECT *
	FROM t
	WHERE to_char(sdate,'yyyy') between 1998 AND 2018
		t[(t.sdate.dt.year >= 1998) & (t.sdate.dt.year <= 2018)]

	SELECT *
	FROM t
	WHERE to_char(sdate ,'day')= 'Friday'
		df[df.sdate.dt.day_name() == 'Friday']

7 String related functionality like like, Substr

For columns with string content, we could access string related funftionality by applying .str on column. Here are few samples: str.contains - contains options:

```
Series.str.contains(pat, case=True, flags=0, na=nan, regex=True)
```

Here are list of main ‘string‘ functions.

str.upper

str.lower

str.extract

str.extractall

str.find

str.findall

str.len

str.replace

str.slice

str.split

str.strip

Check Pandas documentation for more!

	SQL Sample	Pandas Sample

regex	SELECT
	upper(trim(to_char(LAC,'xxxxx')) ||'-' || trim(to_char(CI,'xxxxx'))) AS "LAC-CI(HEX)"
	FROM t
		t = t['LAC','CI']\
		.apply(lambda x: x\
		.astype(str)\
		.map(lambda x: int(x, base=16)))t
		.assig(LAC-CI(HEX) = t['LAC']+'-'+t['CI']

substr	SELECT *
	FROM tips
	WHERE substr(time,1,2) = 'Di'
		tips[tips['time']\
		.str[:2] == 'Di']

<table>
<tr><th></th><th>SQL Sample</th><th>Pandas Sample</th></tr>
<tr><td>regex</td><td>SELECT
upper(trim(to_char(LAC,'xxxxx'))
||'-' ||
trim(to_char(CI,'xxxxx')))
AS "LAC-CI(HEX)"
FROM t</td><td>t = t['LAC','CI']\
.apply(lambda x: x\
.astype(str)\
.map(lambda x: int(x,
base=16)))t
.assig(LAC-CI(HEX) =
t['LAC']+'-'+t['CI']</td></tr>
<tr><td>like</td><td>SELECT *
FROM df
WHERE Country like '%IR%'</td><td>df[df['Country']\
.str.contains('IR') ==
True]</td></tr>
<tr><td>like</td><td>SELECT *
FROM df
WHERE Country like 'IR%'</td><td>df[df['Country']\
.str.startswith('IR')
== True]</td></tr>
<tr><td>like</td><td>SELECT *
FROM df
WHERE Country like '%AN'</td><td>df[df['Country']\
.str.endswith('AN') ==
True]</td></tr>
<tr><td>in</td><td>SELECT *
FROM df
WHERE City in ('TEHRAN',
'BERLIN','STOKHOLM')</td><td>df[df['City']\
.isin(['TEHRAN',
'BERLIN','STOKHOLM'])</td></tr>
<tr><td>regex</td><td>SELECT last_name
FROM contacts
WHERE REGEXP_LIKE
(last_name, '^+A(*)')</td><td>contacts[contacts['last_name']\
.str.contains('^+A(*)')]</td></tr>
<tr><td>regex</td><td>SELECT c1
FROM t
WHERE
REGEXP_LIKE(c1,'([A-Z][\d]{4})')</td><td>t[t['c1']\
.str.contains(([A-Z][\d]{4}))]</td></tr>
</table>

|| provide concatenation functonality in PL/SQL. In **Python**, + on string values resulted in contanated text.

SQL Sample	Pandas Sample

SELECT *
FROM
(SELECT t.*,
ROW_NUMBER()
OVER(PARTITION BY day
ORDER BY total_bill
DESC) AS r
FROM tips t)
WHERE r <
ORDER BY day, r;
	(tips.assign(r=tips\
	.sort_values(['total_bill'],
	ascending=False)\
	.groupby(['day'])\
	.cumcount()+1)\
	.query('r < 3')
	.sort_values(['day', 'r']))

Oracle's ROW_NUMBER() analytic function

8 To check for missing values

- `df.notnull()` Use to Drop Missing Values
- `df.dropna()` Filling Missing Values — Direct Replace
- `df.fillna()`

Besides Pandas comparison with SQL, I also get ideas from following references:

1. pandas-cheatsheet-for-sql-people-part-1
2. how-to-rewrite-your-sql-queries-in-pandas-and-more
3. thinking-like-sql-in-pandas
4. did-you-know-pandas-can-do-so-much
5. 10-python-pandas-tricks-that-make-your-work-more-efficient