Search the docs ...

pandas.DataFrame
pandas.DataFrame.index
pandas.DataFrame.columns
pandas.DataFrame.dtypes
pandas.DataFrame.info
pandas.DataFrame.select_dtypes
pandas.DataFrame.values
pandas.DataFrame.axes
pandas.DataFrame.ndim
pandas.DataFrame.size
pandas.DataFrame.shape
pandas.DataFrame.memory_usage
pandas.DataFrame.empty
pandas.DataFrame.set_flags
pandas.DataFrame.astype
pandas.DataFrame.convert_dtypes
pandas.DataFrame.infer_objects
pandas.DataFrame.copy
pandas.DataFrame.bool
pandas.DataFrame.head
pandas.DataFrame.at
pandas.DataFrame.iat
pandas.DataFrame.loc
pandas.DataFrame.iloc
pandas.DataFrame.insert
pandas.DataFrame.__iter__
pandas.DataFrame.items
pandas.DataFrame.iteritems
pandas.DataFrame.keys
pandas.DataFrame.iterrows
pandas.DataFrame.itertuples
pandas.DataFrame.lookup
pandas.DataFrame.pop
pandas.DataFrame.tail
pandas.DataFrame.xs
pandas.DataFrame.get
pandas.DataFrame.isin
pandas.DataFrame.where
pandas.DataFrame.mask
pandas.DataFrame.query
pandas.DataFrame.add
pandas.DataFrame.sub
pandas.DataFrame.mul
pandas.DataFrame.div
pandas.DataFrame.truediv
pandas.DataFrame.floordiv
pandas.DataFrame.mod
pandas.DataFrame.pow

pandas.DataFrame.to_sql

`DataFrame.to_sql`(*name, con, schema=None, if_exists='fail', index=True, index_label=None, chunksize=None, dtype=None, method=None*) [source]

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [1] are supported. Tables can be newly created, appended to, or overwritten.

Parameters:

name : *str*

Name of SQL table.

con : *sqlalchemy.engine.(Engine or Connection) or sqlite3.Connection*

Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable See here.

schema : *str, optional*

Specify the schema (if database flavor supports this). If None, use default schema.

if_exists : *{'fail', 'replace', 'append'}, default 'fail'*

How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

index : *bool, default True*

Write DataFrame index as a column. Uses *index_label* as the column name in the table.

index_label : *str or sequence, default None*

Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

chunksize : *int, optional*

Specify the number of rows in each batch to be written at a time. By default, all rows will be written at once.

dtype : *dict or scalar, optional*

Specifying the datatype for columns. If a dictionary is used, the keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode. If a scalar is provided, it will be applied to all columns.

method : *{None, 'multi', callable}, optional*

Controls the SQL insertion clause used:

- None : Uses standard SQL `INSERT` clause (one per row).
- 'multi': Pass multiple values in a single `INSERT` clause.
- callable with signature `(pd_table, conn, keys, data_iter)`.

Details and a sample callable implementation can be found in the section insert method.

New in version 0.24.0.

Raises:

ValueError

When the table already exists and *if_exists* is 'fail' (the default).

See also

`read_sql`

Read a DataFrame from a table.

Notes

Timezone aware datetime columns will be written as `Timestamp with timezone` type with SQLAlchemy if supported by the database. Otherwise, the datetimes will be stored as timezone unaware timestamps local to the original timezone.

New in version 0.24.0.

References

[1] https://docs.sqlalchemy.org

2 https://www.python.org/dev/peps/pep-0249/

Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

An *sqlalchemy.engine.Connection* can also be passed to *con*:

```
>>> with engine.begin() as connection:
...     df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
...     df1.to_sql('users', con=connection, if_exists='append')
```

This is allowed to support operations that require that the same DBAPI connection is used for the entire operation.

```
>>> df2 = pd.DataFrame({'name' : ['User 6', 'User 7']})
>>> df2.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5'), (0, 'User 6'),
 (1, 'User 7')]
```

Overwrite the table with just `df2`.

```
>>> df2.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 6'), (1, 'User 7')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```


<< pandas.DataFrame.to_records

pandas.DataFrame.to_stata >>