Pandemy

Release 1.2.0

Anton Lydell

2023-02-06

CONTENTS

Pandemy is a wrapper around pandas and SQLAlchemy to provide an easy class based interface for working with DataFrames and databases. This package is for those who enjoy working with pandas and SQL but do not want to learn all "bells and whistles" of SQLAlchemy. Use your existing SQL knowledge and provide text based SQL statements to load DataFrames from and write DataFrames to databases.

- **Release**: 1.2.0 | 2023-02-06
- **License**: Pandemy is distributed under the MIT-license.

CHAPTER

ONE

DISPOSITION

The documentation consists of 3 parts:

- *Getting started* : Install Pandemy and get a brief overview of the package.
- *User guide* : The structure of Pandemy and a walkthrough of how to use it.
- *API reference* : Details about the API of Pandemy.

1.1 Getting started

This chapter describes how to install Pandemy and showcases a brief overview of saving a `pandas.DataFrame` to and reading a `pandas.DataFrame` from a SQLite database. A teaser for what the *User guide* has to offer.

Throughout this documentation `$` will be used to distinguish a shell prompt from Python code and `>>>` indicates a Python interactive shell. `$` and `>>>` should *not* be copied from the examples when trying them out on your own.

1.1.1 Installation

Pandemy is available for installation through PyPI using pip and conda-forge using conda. The source code is hosted on GitHub at: https://github.com/antonlydell/Pandemy

Install with pip:

```
$ pip install Pandemy
```

Install with conda:

```
$ conda install -c conda-forge pandemy
```

Dependencies

The core dependencies of Pandemy are:

- pandas : powerful Python data analysis toolkit
- SQLAlchemy : The Python SQL Toolkit and Object Relational Mapper

Optional dependencies

Databases except for SQLite require a third-party database driver package to be installed. SQLAlchemy uses the database driver to communicate with the database. The default database driver for SQLite is built-in to Python in the `sqlite3` module[1]. The optional dependencies of Pandemy can be installed by supplying an optional dependency identifier to the pip installation command. The table below lists database driver packages for supported databases and their corresponding optional dependency identifier.

Table 1: Optional dependencies of Pandemy.

Database	Driver package	Optional dependency identifier	Version added
Oracle	cx_Oracle	oracle	1.1.0

To install cx_Oracle together with Pandemy run:

```
$ pip install Pandemy[oracle]
```

When using conda supply the driver package as a separate argument to the install command:

```
$ conda install -c conda-forge pandemy cx_oracle
```

1.1.2 Overview

This section shows a simple example of using Pandemy to write a `pandas.DataFrame` to a SQLite database and reading it back again.

Save a DataFrame to a table

Let's create a new SQLite database and save a `pandas.DataFrame` to it.

```
# overview_save_df.py

import io
import pandas as pd
import pandemy

# Data to save to the database
data = io.StringIO("""
ItemId,ItemName,MemberOnly,Description
1,Pot,0,This pot is empty.
2,Jug,0,This jug is empty.
3,Shears,0,For shearing sheep.
4,Bucket,0,It's a wooden bucket.
5,Bowl,0,Useful for mixing things.
6,Amulet of glory,1,A very powerful dragonstone amulet.
""")

df = pd.read_csv(filepath_or_buffer=data, index_col='ItemId')  # Create a DataFrame

# SQL statement to create the table Item in which to save the DataFrame df
```

(continues on next page)

[1] There are other drivers for SQLite, see SQLite drivers in the SQLAlchemy documentation.

(continued from previous page)

```
create_table_item = """
-- The available items in General Stores
CREATE TABLE IF NOT EXISTS Item (
   ItemId      INTEGER,
   ItemName    TEXT    NOT NULL,
   MemberOnly  INTEGER NOT NULL,
   Description TEXT,

   CONSTRAINT ItemPk PRIMARY KEY (ItemId)
);
"""

db = pandemy.SQLiteDb(file='Runescape.db')  # Create the SQLite DatabaseManager instance

with db.engine.begin() as conn:
   db.execute(sql=create_table_item, conn=conn)
   db.save_df(df=df, table='Item', conn=conn)
```

```
$ python overview_save_df.py
```

The database is managed through the *DatabaseManager* class (in this case the *SQLiteDb* instance). Each SQL dialect is a subclass of *DatabaseManager*. The initialization of the *DatabaseManager* creates the database `engine`, which is used to create a connection to the database. The `engine.begin()` method returns a context manager with an open database transaction, which commits the statements if no errors occur or performs a rollback on error. The connection is automatically returned to the engine's connection pool when the context manager exits. If the database file does not exist it will be created.

The *DatabaseManager.execute()* method allows for execution of arbitrary SQL statements such as creating a table. The *DatabaseManager.save_df()* method saves the `pandas.DataFrame` `df` to the table *Item* in the database `db`.

Load a table into a DataFrame

The `pandas.DataFrame` saved to the table *Item* of the database *Runescape.db* can easily be read back into a `pandas.DataFrame`.

```
# overview_load_table.py

import pandemy

db = pandemy.SQLiteDb(file='Runescape.db', must_exist=True)

sql = """SELECT * FROM Item ORDER BY ItemId;"""  # Query to read back table Item into a␣
↪DataFrame

with db.engine.connect() as conn:
   df_loaded = db.load_table(sql=sql, conn=conn, index_col='ItemId')

print(df_loaded)
```

```
$ python overview_load_table.py
```

```
                ItemName  MemberOnly                          Description
ItemId
1                    Pot           0                    This pot is empty.
2                    Jug           0                    This jug is empty.
3                 Shears           0                  For shearing sheep.
4                 Bucket           0                 It's a wooden bucket.
5                    Bowl           0             Useful for mixing things.
6       Amulet of glory           1  A very powerful dragonstone amulet.
```

If the `must_exist` parameter is set to `True` *pandemy.DatabaseFileNotFoundError* will be raised if the database file is not found. This is useful if you expect the database to exist and you want to avoid creating a new database by mistake if it does not exist. The `engine.connect()` method is similar to `engine.begin()`, but without opening a transaction. The *DatabaseManager.load_table()* method supports either a table name or a sql statement for the `sql` parameter.

1.2 User guide

This chapter explains the main concepts and use cases of Pandemy. It starts with a description about the core components of Pandemy: the *DatabaseManager* and *SQLContainer* classes. Thereafter the implemented SQL dialects are described.

1.2.1 The DatabaseManager

DatabaseManager is the base class that defines the interface of how to interact with the database and provides the methods to do so. Each SQL dialect will inherit from the *DatabaseManager* and define the specific details of how to connect to the database and create the database `engine`. The database `engine` is the core component that allows for connection and interaction with the database. The `engine` is created through the `sqlalchemy.create_engine()` function. The creation of the connection URL needed to create the `engine` is handled during the initialization of *DatabaseManager*. In cases where a subclass of *DatabaseManager* for the desired SQL dialect does not exist this class can be used on its own (starting in version 1.2.0) but with limited functionality. Some methods that require dialect specific SQL statements such as *merge_df()* will not be available. Using *DatabaseManager* on its own also requires initialization through a SQLAlchemy `URL` or `Engine`, which require some knowledge about SQLAlchemy.

SQL dialects

This section describes the available SQL dialects in Pandemy and the dialects planned for future releases.

- **SQLite**: *SQLiteDb*.
- **Oracle**: *OracleDb* (*New in version 1.1.0*).
- **Microsoft SQL Server**: Planned.

Core functionality

All SQL dialects inherit these methods from *DatabaseManager*:

- *delete_all_records_from_table()*: Delete all records from an existing table in the database.
- *execute()*: Execute arbitrary SQL statements on the database.
- *load_table()*: Load a table by name or SQL query into a pandas.DataFrame.
- *manage_foreign_keys()*: Manage how the database handles foreign key constraints.
- *merge_df()*: Merge data from a pandas.DataFrame into a table (*OracleDb* only).
- *save_df()*: Save a pandas.DataFrame to a table in the database.
- *upsert_table()*: Update a table with data from a pandas.DataFrame and insert new rows if any.

New in version 1.2.0: *merge_df()* and *upsert_table()*

Examples of using these methods are shown in the *SQLite* and *Oracle* sections, but they work the same regardless of the SQL dialect used.

The SQLContainer

When initializing a subclass of *DatabaseManager* it can optionally be passed a *SQLContainer* class to the `container` parameter. The purpose of the *SQLContainer* is to store SQL statements used by an application in one place where they can be easily accessed by the *DatabaseManager*. Just like the *DatabaseManager* the *SQLContainer* should be subclassed and not used directly. If your application supports multiple SQL databases you can write the SQL statements the application needs in each SQL dialect and store the statements in one *SQLContainer* per dialect. Examples of using the *SQLContainer* with the SQLite DatabaseManager *SQLiteDb* are shown in section *Using the SQLContainer*.

1.2.2 SQLite

This section describes the SQLite DatabaseManager *SQLiteDb*.

Initialization

Initializing a SQLite *DatabaseManager* with no arguments creates a database that lives in memory.

```
>>> import pandemy
>>> db = pandemy.SQLiteDb()
>>> print(db)
SQLiteDb(file=':memory:', must_exist=False)
```

The `file` parameter is used to connect to a database file and if the file does not exist it will be created, which is the standard behavior of SQLite. The `file` parameter can be a string or pathlib.Path with the full path to the file or the filename relative to the current working directory. Specifying `file=':memory:'` will create an in memory database which is the default.

```
>>> import pandemy
>>> db = pandemy.SQLiteDb(file='my_db.db')
>>> print(db)
SQLiteDb(file='my_db.db', must_exist=False)
```

Require the database to exist

Sometimes creating a new database if the database file does not exist is not a desired outcome. Your application may expect a database to already exist and be ready for use and if it does not exist the application cannot function correctly. For these circumstances you can set the `must_exist` parameter to `True` which will raise *pandemy.DatabaseFileNotFoundError* if the file is not found.

```
>>> import pandemy
>>> db = pandemy.SQLiteDb('my_db_does_not_exist.db', must_exist=True)
Traceback (most recent call last):
...
pandemy.exceptions.DatabaseFileNotFoundError: file='my_db_does_not_exist.db' does not␣
↪exist and and must_exist=True. Cannot instantiate the SQLite DatabaseManager.
```

The execute method

The *DatabaseManager.execute()* method can be used to execute arbitrary SQL statements e.g. creating a table. Let us create a SQLite database that can be used to further demonstrate how Pandemy works.

The complete SQLite test database that is used for the test suite of Pandemy can be downloaded below to test Pandemy on your own.

```
# create_database.py

import pandemy

# SQL statement to create the table Item in which to save the DataFrame df
create_table_item = """
-- The available items in General Stores
CREATE TABLE IF NOT EXISTS Item (
ItemId      INTEGER,
ItemName    TEXT    NOT NULL,
MemberOnly  INTEGER NOT NULL,
Description TEXT,

CONSTRAINT ItemPk PRIMARY KEY (ItemId)
);
"""

db = pandemy.SQLiteDb(file='Runescape.db')  # Create the SQLite DatabaseManager instance

with db.engine.connect() as conn:
   db.execute(sql=create_table_item, conn=conn)
```

```
$ python create_database.py
```

Parametrized SQL statements

A parametrized SQL statement can be created by using the `params` parameter. The SQL statement should contain placeholders that will be replaced by values when the statement is executed. The placeholders should be prefixed by a colon (*:*) (e.g. `:myparameter`) in the SQL statement. The parameter `params` takes a dictionary that maps the parameter name to its value(s) or a list of such dictionaries. Note that parameter names in the dictionary should not be prefixed with a colon i.e. the key in the dictionary that references the SQL placeholder `:myparameter` should be named `'myparameter'`.

Let's insert some data into the *Item* table we just created in *Runescape.db*.

```
# execute_insert_into.py

import pandemy

# SQL statement to insert values into the table Item
insert_into_table_item = """
INSERT INTO Item (ItemId, ItemName, MemberOnly, Description)
   VALUES (:itemid, :itemname, :memberonly, :description);
"""

params = {'itemid': 1, 'itemname': 'Pot', 'memberonly': 0, 'description': 'This pot is␣
↪empty'}

db = pandemy.SQLiteDb(file='Runescape.db')  # Create the SQLite DatabaseManager instance

with db.engine.connect() as conn:
   db.execute(sql=insert_into_table_item, conn=conn, params=params)

# Add some more rows
params = [
   {'itemid': 2, 'itemname': 'Jug', 'memberonly': 0, 'description': 'This jug is empty'},
   {'itemid': 3, 'itemname': 'Shears', 'memberonly': 0, 'description': 'For shearing␣
↪sheep'},
   {'itemid': 4, 'itemname': 'Bucket', 'memberonly': 0, 'description': 'It''s a wooden␣
↪bucket.'}
]

with db.engine.connect() as conn:
   db.execute(sql=insert_into_table_item, conn=conn, params=params)

# Retrieve the inserted rows
query = """SELECT * FROM Item;"""

with db.engine.connect() as conn:
   result = db.execute(sql=query, conn=conn)

   for row in result:
      print(row)
```

```
$ python execute_insert_into.py
```

```
(1, 'Pot', 0, 'This pot is empty')
```

(continues on next page)

(continued from previous page)

```
(2, 'Jug', 0, 'This jug is empty')
(3, 'Shears', 0, 'For shearing sheep')
(4, 'Bucket', 0, 'Its a wooden bucket.')
```

The *DatabaseManager.execute()* method returns an object called `sqlalchemy.engine.CursorResult` (the variable `result`). This object is an iterator that can be used to retrieve rows from the result set of a *SELECT* statement.

Note: The database connection must remain open while iterating over the rows in the `CursorResult` object, since it is fetching one row from the database at the time. This means that the for loop must be placed inside the context manager.

Using transactions

Database transactions can be invoked by calling the `begin()` method of the database `engine` instead of `connect()`. When executing SQL statements under an open transaction all statements will automatically be rolled back to the latest valid state if an error occurs in one of the statements. This differs from using the connect method where only the statement where the error occurs will be rolled back. The example below illustrates this difference.

```
# execute_insert_into_transaction.py

# Using the previously created database Runescape.db
db = pandemy.SQLiteDb(file='Runescape.db')

# Clear previous content in the table Item
with db.engine.connect() as conn:
   db.delete_all_records_from_table(table='Item', conn=conn)

# SQL statement to insert values into the table Item
insert_into_table_item = """
INSERT INTO Item (ItemId, ItemName, MemberOnly, Description)
   VALUES (:itemid, :itemname, :memberonly, :description);
"""

row_1 = {'itemid': 1, 'itemname': 'Pot', 'memberonly': 0, 'description': 'This pot is␣
↪empty'}

with db.engine.connect() as conn:
   db.execute(sql=insert_into_table_item, conn=conn, params=row_1)

# Add a second row
row_2 = {'itemid': 2, 'itemname': 'Jug', 'memberonly': 0, 'description': 'This jug is␣
↪empty'},

# Add some more rows (the last row contains a typo for the itemid parameter)
rows_3_4 = [{'itemid': 3, 'itemname': 'Shears', 'memberonly': 0, 'description': 'For␣
↪shearing sheep'},
            {'itemi': 4, 'itemname': 'Bucket', 'memberonly': 0, 'description': 'It''s a␣
↪wooden bucket.'}]

# Insert with a transaction
```

(continues on next page)

(continued from previous page)

```
try:
   with db.engine.begin() as conn:
      db.execute(sql=insert_into_table_item, conn=conn, params=row_2)
      db.execute(sql=insert_into_table_item, conn=conn, params=rows_3_4)
except pandemy.ExecuteStatementError as e:
   print(f'{e.args}\n')

# Retrieve the inserted rows
query = """SELECT * FROM Item;"""

with db.engine.connect() as conn:
   result = db.execute(sql=query, conn=conn)
   result = result.fetchall()

print(f'All statements under the transaction are rolled back when an error occurs:\n
↪{result}\n\n')

# Using connect instead of begin
try:
   with db.engine.connect() as conn:
      db.execute(sql=insert_into_table_item, conn=conn, params=row_2)
      db.execute(sql=insert_into_table_item, conn=conn, params=rows_3_4)
except pandemy.ExecuteStatementError as e:
   print(f'{e.args}\n')

# Retrieve the inserted rows
query = """SELECT * FROM Item;"""

with db.engine.connect() as conn:
   result = db.execute(sql=query, conn=conn)
   result = result.fetchall()

print(f'Only the statement with error is rolled back when using connect:\n{result}')
```

```
$ python execute_insert_into_transaction.py
```

```
('StatementError: ("(sqlalchemy.exc.InvalidRequestError) A value is required for bind␣
↪parameter \'itemid\', in parameter group 1",)',)

All statements under the transaction are rolled back when an error occurs:
[(1, 'Pot', 0, 'This pot is empty')]


('StatementError: ("(sqlalchemy.exc.InvalidRequestError) A value is required for bind␣
↪parameter \'itemid\', in parameter group 1",)',)

Only the statement with error is rolled back when using connect:
[(1, 'Pot', 0, 'This pot is empty'), (2, 'Jug', 0, 'This jug is empty')]
```

Note: The `sqlalchemy.engine.CursorResult.fetchall()` method can be used to retrieve all rows from the

query into a list.

Warning: The method *DatabaseManager.delete_all_records_from_table()* will delete all records from a table. Use this method with caution. It is mainly used to clear all content of a table before replacing it with new data. This method is used by the *DatabaseManager.save_df()* method when using `if_exists='replace'`, which is described in the next section.

See also:

The SQLAlchemy documentation provides more information about transactions:

- `sqlalchemy.engine.Engine.begin()` : Establish a database connection with a transaction.
- `sqlalchemy.engine.Engine.connect()` : Establish a database connection.
- `sqlalchemy.engine.Transaction` : A database transaction object.

Save a DataFrame to a table

Executing insert statements with the *DatabaseManager.execute()* method as shown above is not very practical if you have a lot of data in a `pandas.DataFrame` that you want to save to a table. In these cases it is better to use the method *DatabaseManager.save_df()*. It uses the power of `pandas.DataFrame.to_sql()` method and lets you save a `pandas.DataFrame` to a table in a single line of code.

The example below shows how to insert data to the table *Item* using a `pandas.DataFrame`.

```
# save_df.py

import io
import pandas as pd
import pandemy

# The content to write to table Item
data = io.StringIO(r"""
ItemId;ItemName;MemberOnly;Description
1;Pot;0;This pot is empty.
2;Jug;0;This jug is empty.
3;Shears;0;For shearing sheep.
4;Bucket;0;It's a wooden bucket.
5;Bowl;0;Useful for mixing things.
6;Amulet of glory;1;A very powerful dragonstone amulet.
7;Tinderbox;0;Useful for lighting a fire.
8;Chisel;0;Good for detailed Crafting.
9;Hammer;0;Good for hitting things.
10;Newcomer map;0;Issued to all new citizens of Gielinor.
11;Unstrung symbol;0;It needs a string so I can wear it.
12;Dragon Scimitar;1;A vicious, curved sword.
13;Amulet of glory;1;A very powerful dragonstone amulet.
14;Ranarr seed;1;A ranarr seed - plant in a herb patch.
15;Swordfish;0;I'd better be careful eating this!
16;Red dragonhide Body;1;Made from 100% real dragonhide.
""")
```

(continues on next page)

(continued from previous page)

```
df = pd.read_csv(filepath_or_buffer=data, sep=';', index_col='ItemId')  # Create the␣
↪DataFrame

db = pandemy.SQLiteDb(file='Runescape.db')  # Create the SQLite DatabaseManager instance

with db.engine.connect() as conn:
   db.save_df(df=df, table='Item', conn=conn, if_exists='replace')
```

```
$ python save_df.py
```

DatabaseManager.save_df() implements all parameters of pandas.DataFrame.to_sql(). The `if_exists` parameter is slightly different. `if_exists` controls how to save a pandas.DataFrame to an existing table in the database.

`if_exists` accepts the following values:

- *'append'*: Append the pandas.DataFrame to the existing table (default).
- *'replace'*: Delete all records from the table and then write the pandas.DataFrame to the table.
- *'drop-replace'*: Drop the table, recreate it, and then write the pandas.DataFrame to the table.
- *'fail'*: Raise *pandemy.TableExistsError* if the table exists.

New in version 1.2.0: 'drop-replace'

In the pandas.DataFrame.to_sql() method *'fail'* is the default value. The option *'replace'* drops the existing table, recreates it with the column definitions from the pandas.DataFrame, and inserts the data. By dropping the table and recreating it you loose important information such as primary keys and constraints.

In *DatabaseManager.save_df()* *'replace'* deletes all current records before inserting the new data rather than dropping the table. This preserves the existing columns definitions and constraints of the table. Deleting the current records is done with the *DatabaseManager.delete_all_records_from_table()* method. The *'drop-replace'* option is the equivalent of *'replace'* in pandas.DataFrame.to_sql().

Load a DataFrame from a table

To load data from a table into a pandas.DataFrame the *DatabaseManager.load_table()* method is used. It uses the pandas.read_sql() function with some extra features.

Let us load the table *Item* back into a pandas.DataFrame.

```
# load_table.py

import pandemy

db = pandemy.SQLiteDb(file='Runescape.db', must_exist=True)

query = """SELECT * FROM Item ORDER BY ItemId ASC;"""

with db.engine.connect() as conn:
   df = db.load_table(sql=query, conn=conn, index_col='ItemId')

print(df)
```

```
$ python load_table.py
```

ItemId	ItemName	MemberOnly	Description
1	Pot	0	This pot is empty.
2	Jug	0	This jug is empty.
3	Shears	0	For shearing sheep.
4	Bucket	0	It's a wooden bucket.
5	Bowl	0	Useful for mixing things.
6	Amulet of glory	1	A very powerful dragonstone amulet.
7	Tinderbox	0	Useful for lighting a fire.
8	Chisel	0	Good for detailed Crafting.
9	Hammer	0	Good for hitting things.
10	Newcomer map	0	Issued to all new citizens of Gielinor.
11	Unstrung symbol	0	It needs a string so I can wear it.
12	Dragon Scimitar	1	A vicious, curved sword.
13	Amulet of glory	1	A very powerful dragonstone amulet.
14	Ranarr seed	1	A ranarr seed - plant in a herb patch.
15	Swordfish	0	I'd better be careful eating this!
16	Red dragonhide Body	1	Made from 100% real dragonhide.

Note: The `sql` parameter can be either a SQL query or a table name. Using a table name will not guarantee the order of the retrieved rows.

Working with datetimes and timezones

Columns with datetime information can be converted into datetime columns by using the `parse_dates` keyword argument, which is a direct link to the `parse_dates` option of `pandas.read_sql()` function.

`parse_dates` only returns naive datetime columns. To load datetime columns with timezone information the keyword arguments `localize_tz` and `target_tz` can be specified. `localize_tz` lets you localize the the naive datetime columns to a specified timezone and `target_tz` can optionally convert the localized datetime columns into a desired timezone.

Let's create the table *Customer* from the database *Runescape.db* and load it into a `pandas.DataFrame` to illustrate this.

```
# load_table_localize_tz.py

import io
import pandas as pd
import pandemy

# SQL statement to create the table Customer in which to save the DataFrame df
create_table_customer = """
-- Customers that have traded in a General Store
CREATE TABLE IF NOT EXISTS Customer (
   CustomerId         INTEGER,
   CustomerName       TEXT    NOT NULL,
   BirthDate          TEXT,
   Residence          TEXT,
   IsAdventurer       INTEGER NOT NULL, -- 1 if Adventurer and 0 if NPC
```

(continues on next page)

(continued from previous page)

```
    CONSTRAINT CustomerPk PRIMARY KEY (CustomerId)
);
"""

db = pandemy.SQLiteDb(file='Runescape.db')  # Create the SQLite DatabaseManager instance

data = io.StringIO("""
CustomerId;CustomerName;BirthDate;Residence;IsAdventurer
1;Zezima;1990-07-14;Yanille;1
2;Dr Harlow;1970-01-14;Varrock;0
3;Baraek;1968-12-13;Varrock;0
4;Gypsy Aris;1996-03-24;Varrock;0
5;Not a Bot;2006-05-31;Catherby;1
6;Max Pure;2007-08-20;Port Sarim;1
""")

dtypes = {
    'CustomerId': 'int8',
    'CustomerName': 'string',
    'Residence': 'string',
    'IsAdventurer': 'boolean'
}

df = pd.read_csv(filepath_or_buffer=data, sep=';', index_col='CustomerId', dtype=dtypes)

with db.engine.begin() as conn:
    db.execute(sql=create_table_customer, conn=conn)
    db.save_df(df=df, table='Customer', conn=conn, if_exists='replace')

    df_naive = db.load_table(
        sql='Customer',
        conn=conn,
        index_col='CustomerId',
        dtypes=dtypes,
        parse_dates={'BirthDate': r'%Y-%m-%d'}
    )

    df_dt_aware = db.load_table(
        sql='Customer',
        conn=conn,
        index_col='CustomerId',
        dtypes=dtypes,
        parse_dates={'BirthDate': r'%Y-%m-%d'},
        localize_tz='UTC',
        target_tz='CET'
    )

print(f'df:\n{df}\n')

print(f'df_naive:\n{df_naive}\n')
print(f'df_naive.dtypes:\n{df_naive.dtypes}\n')
```

(continues on next page)

(continued from previous page)

```
print(f'df_dt_aware:\n{df_dt_aware}\n')
print(f'df_dt_aware.dtypes:\n{df_dt_aware.dtypes}')
```

```
$ python load_table_localize_tz.py
```

```
df:
            CustomerName  BirthDate   Residence  IsAdventurer
CustomerId
1                 Zezima 1990-07-14     Yanille          True
2              Dr Harlow 1970-01-14     Varrock         False
3                 Baraek 1968-12-13     Varrock         False
4              Gypsy Aris 1996-03-24     Varrock         False
5              Not a Bot 2006-05-31    Catherby          True
6               Max Pure 2007-08-20  Port Sarim          True

df_naive:
             CustomerName  BirthDate   Residence  IsAdventurer
CustomerId
1                 Zezima 1990-07-14     Yanille           True
2              Dr Harlow 1970-01-14     Varrock          False
3                 Baraek 1968-12-13     Varrock          False
4              Gypsy Aris 1996-03-24     Varrock          False
5              Not a Bot 2006-05-31    Catherby           True
6               Max Pure 2007-08-20  Port Sarim           True

df_naive.dtypes:
CustomerName            string
BirthDate       datetime64[ns]
Residence               string
IsAdventurer           boolean
dtype: object

df_dt_aware:
             CustomerName                  BirthDate   Residence  IsAdventurer
CustomerId
1                 Zezima 1990-07-14 02:00:00+02:00     Yanille           True
2              Dr Harlow 1970-01-14 01:00:00+01:00     Varrock          False
3                 Baraek 1968-12-13 01:00:00+01:00     Varrock          False
4              Gypsy Aris 1996-03-24 01:00:00+01:00     Varrock          False
5              Not a Bot 2006-05-31 02:00:00+02:00    Catherby           True
6               Max Pure 2007-08-20 02:00:00+02:00  Port Sarim           True

df_dt_aware.dtypes:
CustomerName                 string
BirthDate       datetime64[ns, CET]
Residence                    string
IsAdventurer                boolean
dtype: object
```

Update and insert data into a table (upsert)

Sometimes you have a pandas.DataFrame that represents an existing table in the database that already contains data. If you want to update the existing records of the table with data from the pandas.DataFrame and also insert new rows (that exist in the pandas.DataFrame but not in the table) you should use the *DatabaseManager.upsert_table()* (update and insert) method. The method can update and insert new rows or only update existing rows. It works by creating and executing an UPDATE statement followed by an optional INSERT statement derived from the structure of the pandas.DataFrame. Let's look at some examples of using *DatabaseManager.upsert_table()* with the *Customer* table of *Runescape.db*.

```
# upsert_table.py

from datetime import datetime
import pandas as pd
import pandemy

db = pandemy.SQLiteDb(file='Runescape.db', must_exist=True)

query = """SELECT * FROM Customer ORDER BY CustomerId ASC"""

dtypes = {
   'CustomerName': 'string',
   'Residence': 'string',
   'IsAdventurer': 'boolean'
}

with db.engine.connect() as conn:
   df = db.load_table(
      sql=query,
      conn=conn,
      index_col='CustomerId',
      dtypes=dtypes,
      parse_dates=['BirthDate']
   )

print(f'Customer table original:\n\n{df}')

# Change some data
df.loc[1, ['BirthDate', 'Residence']] = [datetime(1891, 7, 15), 'Falador']
df.loc[4, 'IsAdventurer'] = True

# Add new data
df.loc[9, :] = ['Prince Ali', datetime(1969, 6, 20), 'Al Kharid', False]
df.loc[10, :] = ['Mosol Rei', datetime(1983, 4, 30), 'Shilo Village', False]

with db.engine.begin() as conn:
   # Update and insert the new data
   db.upsert_table(
      df=df,
      table='Customer',
      conn=conn,
      where_cols=['CustomerName'],
      upsert_index_cols=False,
```

(continues on next page)

(continued from previous page)

```
      update_only=False,
      datetime_cols_dtype='str',
      datetime_format=r'%Y-%m-%d'
   )

   # Load the data back
   df_upsert = db.load_table(
      sql=query,
      conn=conn,
      index_col='CustomerId',
      dtypes=dtypes,
      parse_dates=['BirthDate']
   )

print(f'\n\nCustomer table after upsert:\n\n{df_upsert}')
```

```
$ python upsert_table.py
```

```
Customer table original:

           CustomerName   BirthDate   Residence  IsAdventurer
CustomerId
1                Zezima  1990-07-14     Yanille          True
2             Dr Harlow  1970-01-14     Varrock         False
3                Baraek  1968-12-13     Varrock         False
4             Gypsy Aris  1996-03-24     Varrock         False
5              Not a Bot  2006-05-31    Catherby          True
6               Max Pure  2007-08-20  Port Sarim          True


Customer table after upsert:

           CustomerName  BirthDate      Residence  IsAdventurer
CustomerId
1                Zezima 1891-07-15        Falador          True
2             Dr Harlow 1970-01-14        Varrock         False
3                Baraek 1968-12-13        Varrock         False
4             Gypsy Aris 1996-03-24        Varrock          True
5              Not a Bot 2006-05-31       Catherby          True
6               Max Pure 2007-08-20     Port Sarim          True
7             Prince Ali 1969-06-20      Al Kharid         False
8              Mosol Rei 1983-04-30  Shilo Village         False
```

The *Customer* table is loaded from the database into a `pandas.DataFrame` (`df`). The data is modified and two new rows are added to `df`. *`DatabaseManager.upsert_table()`* is called with the updated version of `df`. The `where_cols` parameter is set to the *CustomerName* column which means that rows from `df` with a *CustomerName* that already exists in the *Customer* table will be updated. Rows that do not have a matching *CustomerName* will be inserted instead. The *BirthDate* column is inserted as formatted strings (YYYY-MM-DD) by the parameters `datetime_cols_dtype='str'` and `datetime_format=r'%Y-%m-%d'`. Setting the parameter `update_only=True` would have only updated existing rows and not inserted any new rows.

Note: The Primary key column *CustomerId* is not included in database statements sent to the database in the example

above. This is due to the parameter `upsert_index_cols=False`, which is also the default behavior. The values of *CustomerId* in `df` of the two new rows (9 and 10) differ from the ones inserted into the database (7 and 8). The *CustomerId* column is defined as an INTEGER data type in the database and if it is not supplied in the INSERT statement SQLite will autoincrement the value by one from the previous row. It is useful to exclude the Primary key from the upsert if it is generated by the database.

Using the dry_run parameter

If you want to look at the SQL statements sent to the database you can set the parameter `dry_run=True`. This will return the SQL statements that **would have been** executed on the database for every row in `df`. Nothing gets executed on the database. This is useful to verify that you have set the parameters correct to make the statements do what you expect. If `update_only=True` the returned INSERT statement will be `None`.

The next example illustrates using the `dry_run` parameter with the *Customer* table from before.

```
# upsert_table_dry_run.py

from datetime import datetime
import pandas as pd
import pandemy

db = pandemy.SQLiteDb(file='Runescape.db', must_exist=True)

query = """SELECT * FROM Customer ORDER BY CustomerId ASC"""

dtypes = {
   'CustomerName': 'string',
   'Residence': 'string',
   'IsAdventurer': 'boolean'
}

with db.engine.connect() as conn:
   df = db.load_table(
      sql=query,
      conn=conn,
      index_col='CustomerId',
      dtypes=dtypes,
      parse_dates=['BirthDate']
   )

print(f'Customer table:\n\n{df}')

with db.engine.begin() as conn:
   # Get the UPDATE and INSERT statements
   update_stmt, insert_stmt = db.upsert_table(
      df=df,
      table='Customer',
      conn=conn,
      where_cols=['CustomerName', 'BirthDate'],
      upsert_cols=['Residence', 'CustomerName'],
      upsert_index_cols=True,
      update_only=False,
```

(continues on next page)

(continued from previous page)

```
      dry_run=True
   )

print(f'\n\nUPDATE statement:\n\n{update_stmt}')
print(f'INSERT statement:\n\n{insert_stmt}')
```

```
$ python upsert_table_dry_run.py
```

```
Customer table:

            CustomerName  BirthDate      Residence  IsAdventurer
CustomerId
1                  Zezima 1891-07-15        Falador          True
2               Dr Harlow 1970-01-14        Varrock         False
3                  Baraek 1968-12-13        Varrock         False
4              Gypsy Aris 1996-03-24        Varrock          True
5               Not a Bot 2006-05-31       Catherby          True
6                Max Pure 2007-08-20     Port Sarim          True
7              Prince Ali 1969-06-20      Al Kharid         False
8               Mosol Rei 1983-04-30  Shilo Village         False


UPDATE statement:

UPDATE Customer
SET
    Residence = :Residence,
    CustomerId = :CustomerId
WHERE
    CustomerName = :CustomerName AND
    BirthDate = :BirthDate

INSERT statement:

INSERT INTO Customer (
    Residence,
    CustomerName,
    CustomerId
)
    SELECT
        :Residence,
        :CustomerName,
        :CustomerId
    WHERE
        NOT EXISTS (
            SELECT
                1
            FROM Customer
                WHERE
                    CustomerName = :CustomerName AND
                    BirthDate = :BirthDate
        )
```

Here we use the columns *CustomerName* and *BirthDate* in the WHERE clause and specify that the index column (*CustomerId*) should also be updated. If the index is a `pandas.MultiIndex` all levels are included if `upsert_index_cols=True`. A list of level names can be used to only select desired levels of the index to the upsert. By specifying the `upsert_cols` parameter a subset of the columns of `df` can be selected for the upsert, in this case the columns *Residence* and *CustomerName*. Since *CustomerName* is also supplied to the `where_cols` parameter it is excluded from the columns to update, because that would otherwise result in an update to the same value.

Using the SQLContainer

The *SQLContainer* class is a container for the SQL statements used by an application. The database managers can optionally be initialized with a *SQLContainer* through the keyword argument `container`. *SQLContainer* is the base class and provides some useful methods. If you want to use a *SQLContainer* in your application you should subclass from *SQLContainer*. The SQL statements are stored as class variables on the *SQLContainer*. The previously used SQL statements may be stored in a *SQLContainer* like this.

```
# sql_container.py

import pandemy


class SQLiteSQLContainer(pandemy.SQLContainer):
   r"""A container of SQLite database statements."""

   create_table_item = """
   -- The available items in General Stores
   CREATE TABLE IF NOT EXISTS Item (
   ItemId      INTEGER,
   ItemName    TEXT    NOT NULL,
   MemberOnly  INTEGER NOT NULL,
   Description TEXT,

   CONSTRAINT ItemPk PRIMARY KEY (ItemId)
   );
   """

   insert_into_table_item = """
   INSERT INTO TABLE Item (ItemId, ItemName, MemberOnly, Description)
      VALUES (:itemid, :itemname, :memberonly, :description);
   """

   select_all_items = """SELECT * FROM Item ORDER BY ItemId ASC;"""


db = pandemy.SQLiteDb(file='Runescape.db', container=SQLiteSQLContainer)

with db.engine.connect() as conn:
   df = db.load_table(sql=db.container.select_all_items, conn=conn, index_col='ItemId')

print(df)
```

```
$ python sql_container.py
```

ItemId	ItemName	MemberOnly	Description
1	Pot	0	This pot is empty.
2	Jug	0	This jug is empty.
3	Shears	0	For shearing sheep.
4	Bucket	0	It's a wooden bucket.
5	Bowl	0	Useful for mixing things.
6	Amulet of glory	1	A very powerful dragonstone amulet.
7	Tinderbox	0	Useful for lighting a fire.
8	Chisel	0	Good for detailed Crafting.
9	Hammer	0	Good for hitting things.
10	Newcomer map	0	Issued to all new citizens of Gielinor.
11	Unstrung symbol	0	It needs a string so I can wear it.
12	Dragon Scimitar	1	A vicious, curved sword.
13	Amulet of glory	1	A very powerful dragonstone amulet.
14	Ranarr seed	1	A ranarr seed - plant in a herb patch.
15	Swordfish	0	I'd better be careful eating this!
16	Red dragonhide Body	1	Made from 100% real dragonhide.

Replace placeholders

The *SQLContainer.replace_placeholders()* method is used to replace placeholders within a parametrized SQL statement. The purpose of this method is to handle the case of a parametrized query using an *IN* clause with a variable number of arguments. The *IN* clause receives a single placeholder initially which can later be replaced by the correct amount of placeholders once this is determined. The method can of course be used to replace any placeholder within a SQL statement.

The method takes the SQL statement and a single or a sequence of *Placeholder*. It returns the SQL statement with replaced placeholders and a dictionary called `params`. *Placeholder* has 3 parameters:

1. `placeholder` : The placeholder to replace e.g. `':myplaceholder'`.
2. `replacements` : A value or sequence of values to use for replacing `placeholder`.

3. `return_new_placeholders` : A boolean, where `True` indicates that *replace_placeholders()* should return new placeholders mapped to their respective `replacements` as a key value pair in the dictionary `params`. The dictionary `params` can be passed to the `params` keyword argument of the *execute()* or *load_table()* methods of a *DatabaseManager*. The default value is `True`. A value of `False` causes the replaced placeholder to not appear in the returned `params` dictionary.

The use of *replace_placeholders()* and *Placeholder* is best illustrated by some examples using the previously created database *Runescape.db*.

```
# replace_placeholder.py

import pandemy


class SQLiteSQLContainer(pandemy.SQLContainer):
    r"""A container of SQLite database statements."""

    # Retrieve items from table Item by their ItemId
    get_items_by_id = """
    SELECT ItemId, ItemName, MemberOnly, Description
```

(continues on next page)

(continued from previous page)

```
   FROM Item
   WHERE ItemId IN (:itemid)
   ORDER BY ItemId ASC;
   """


items = [1, 3, 5]  # The items to retrieve from table Item

# The placeholder with the replacement values
placeholder = pandemy.Placeholder(placeholder=':itemid',
                                  replacements=items,
                                  return_new_placeholders=True)

db = pandemy.SQLiteDb(file='Runescape.db', container=SQLiteSQLContainer)

stmt, params = db.container.replace_placeholders(stmt=db.container.get_items_by_id,
                                                 placeholders=placeholder)

print(f'get_items_by_id after replacements:\n{stmt}\n')
print(f'The new placeholders with mapped replacements:\n{params}\n')

with db.engine.connect() as conn:
   df = db.load_table(sql=stmt, conn=conn, params=params, index_col='ItemId')

print(f'The DataFrame from the parametrized query:\n{df}')
```

```
$ python replace_placeholder.py
```

```
get_items_by_id after replacements:

   SELECT ItemId, ItemName, MemberOnly, Description
   FROM Item
   WHERE ItemId IN (:v0, :v1, :v2)
   ORDER BY ItemId ASC;

The new placeholders with mapped replacements:
{'v0': 1, 'v1': 3, 'v2': 5}

The DataFrame from the parametrized query:
       ItemName  MemberOnly                Description
ItemId
1           Pot           0         This pot is empty.
3        Shears           0        For shearing sheep.
5          Bowl           0  Useful for mixing things.
```

In this example the placeholder *:itemid* of the query `get_items_by_id` is replaced by three placeholders: *:v0*, *:v1* and *:v2* (one for each of the values in the list `items` in the order they occur). Since `return_new_placeholders=True` the returned dictionary `params` contains a mapping of the new placeholders to the values in the list `items`. If `return_new_placeholders=False` then `params` would be an empty dictionary. The updated version of the query `get_items_by_id` can then be executed with the parameters in `params`.

The next example shows how to replace multiple placeholders.

```
# replace_multiple_placeholders.py

import pandemy


class SQLiteSQLContainer(pandemy.SQLContainer):
   r"""A container of SQLite database statements."""

   get_items_by_id = """
   SELECT ItemId, ItemName, MemberOnly, Description
   FROM Item
   WHERE
      ItemId IN (:itemid)      AND
      MemberOnly = :memberonly AND
      Description LIKE :description
   ORDER BY :orderby;
   """


items = [10, 12, 13, 14, 16]  # The items to retrieve from table Item

# The placeholders with the replacement values
placeholders = [
   pandemy.Placeholder(placeholder=':itemid',
                       replacements=items,
                       return_new_placeholders=True),

   pandemy.Placeholder(placeholder=':memberonly',
                       replacements=1,
                       return_new_placeholders=True),

   pandemy.Placeholder(placeholder=':description',
                       replacements='A%',
                       return_new_placeholders=True),

   pandemy.Placeholder(placeholder=':orderby',
                       replacements='ItemId DESC',
                       return_new_placeholders=False),
]

db = pandemy.SQLiteDb(file='Runescape.db', container=SQLiteSQLContainer)

stmt, params = db.container.replace_placeholders(stmt=db.container.get_items_by_id,
                                                 placeholders=placeholders)

print(f'get_items_by_id after replacements:\n{stmt}\n')
print(f'The new placeholders with mapped replacements:\n{params}\n')

with db.engine.connect() as conn:
   df = db.load_table(sql=stmt, conn=conn, params=params, index_col='ItemId')

print(f'The DataFrame from the parametrized query:\n{df}')
```

```
$ python replace_multiple_placeholders.py
```

```
get_items_by_id after replacements:

   SELECT ItemId, ItemName, MemberOnly, Description
   FROM Item
   WHERE
      ItemId IN (:v0, :v1, :v2, :v3, :v4)      AND
      MemberOnly = :v5 AND
      Description LIKE :v6
   ORDER BY ItemId DESC;


The new placeholders with mapped replacements:
{'v0': 10, 'v1': 12, 'v2': 13, 'v3': 14, 'v4': 16, 'v5': 1, 'v6': 'A%'}

The DataFrame from the parametrized query:
               ItemName  MemberOnly                              Description
ItemId
14          Ranarr seed           1  A ranarr seed - plant in a herb patch.
13      Amulet of glory           1     A very powerful dragonstone amulet.
12      Dragon Scimitar           1                 A vicious, curved sword.
```

Note: The replacement value for the *:orderby* placeholder is not part of the returned `params` dictionary because `return_new_placeholders=False` for the last placeholder.

Warning: Replacing *:orderby* by an arbitrary value that is not a placeholder is not safe against SQL injection attacks the way placeholders are and is therefore discouraged. The feature is there if it is needed, but be aware of its security limitations.

1.2.3 Oracle

This section describes the Oracle *DatabaseManager* *OracleDb*. *OracleDb* is used in the same way as *SQLiteDb* except for the initialization. For information about how to work with *OracleDb* please refer to *The SQLite User guide*. There are several different ways of connecting to an Oracle database and *OracleDb* further simplifies the SQLAlchemy connection process for the most common methods. To use *OracleDb* the Oracle database driver package cx_Oracle needs to be installed. Please refer to the *Installation* section for how to install Pandemy and cx_Oracle.

Initialization

The initialization of *OracleDb* is described further in the *API reference*. Additional information about connecting to an Oracle database with SQLAlchemy can be found in the cx_Oracle part of the SQLAlchemy documentation.

Merge a DataFrame into a table

The *DatabaseManager.merge_df()* method lets you take a pandas.DataFrame and update existing and insert new rows into a table based on some criteria. It executes a combined UPDATE and INSERT statement – a MERGE statement. The MERGE statement is executed once for every row in the pandas.DataFrame. The method and its API is very similar to *DatabaseManager.upsert_table()*. See the section *Update and insert data into a table (upsert)* in the *SQLite User guide*.

An example of using *DatabaseManager.merge_df()* is shown below with the *Item* table from *Runescape.db*. Setting the parameter dry_run=True will return the generated MERGE statement as a string instead of executing it on the database. To avoid having to connect to a real Oracle database server the begin() method of the DatabaseManager.engine is mocked.

```
# merge_df.py

from datetime import datetime
from unittest.mock import MagicMock

import pandas as pd
import pandemy

db_sqlite = pandemy.SQLiteDb(file='Runescape.db', must_exist=True)

query = """SELECT * FROM Item ORDER BY ItemId ASC"""

dtypes = {
   'ItemId': 'int8',
   'ItemName': 'string',
   'MemberOnly': 'boolean',
   'Description': 'string'
}

with db_sqlite.engine.connect() as conn:
   df = db_sqlite.load_table(
      sql=query,
      conn=conn,
      index_col=['ItemId'],
      dtypes=dtypes,
   )

print(f'Item table:\n\n{df}')

# Change some data
df.loc[1, 'Description'] = 'This pot is not empty!'
df.loc[12, ['ItemName', 'MemberOnly']] = ['Dragon Super Scimitar', False]

# Add new data
df.loc[17, :] = ['Coal', False, 'Hmm a non-renewable energy source!']
df.loc[18, :] = ['Redberry pie', False, 'Looks tasty.']

db_oracle = pandemy.OracleDb(
   username='Fred_the_Farmer',
   password='Penguins-sheep-are-not',
   host='fred.farmer.rs',
```

(continues on next page)

(continued from previous page)

```
   port=1234,
   service_name='woollysheep'
)

db_oracle.engine.begin = MagicMock()  # Mock the begin method

print(f'\n\nRows to be updated or inserted:\n\n{df.loc[[1, 12, 17, 18], :]}')

with db_oracle.engine.begin() as conn:
   merge_stmt = db_oracle.merge_df(
      df=df,
      table='Item',
      conn=conn,
      on_cols=['ItemName'],
      merge_cols='all',
      merge_index_cols=False,
      dry_run=True
   )

print(f'\n\nMERGE statement:\n\n{merge_stmt}')
```

```
$ python merge_df.py
```

```
Item table:

                    ItemName  MemberOnly                              Description
ItemId
1                        Pot       False                         This pot is empty.
2                        Jug       False                         This jug is empty.
3                     Shears       False                        For shearing sheep.
4                     Bucket       False                      Its a wooden bucket.
5                       Bowl       False                  Useful for mixing things.
6                  Cake tin       False                  Useful for baking things.
7                 Tinderbox       False                Useful for lighting a fire.
8                    Chisel       False                Good for detailed Crafting.
9                    Hammer       False                   Good for hitting things.
10             Newcomer map       False  Issued to all new citizens of Gielinor.
11          Unstrung symbol       False      It needs a string so I can wear it.
12          Dragon Scimitar        True                  A vicious, curved sword.
13          Amulet of glory        True      A very powerful dragonstone amulet.
14              Ranarr seed        True   A ranarr seed - plant in a herb patch.
15                Swordfish       False        Id better be careful eating this!
16      Red dragonhide Body        True          Made from 100% real dragonhide.


Rows to be updated or inserted:

                      ItemName  MemberOnly                          Description
ItemId
1                          Pot       False              This pot is not empty!
12      Dragon Super Scimitar        False            A vicious, curved sword.
```

(continues on next page)

(continued from previous page)

```
17                      Coal       False  Hmm a non-renewable energy source!
18              Redberry pie       False                          Looks tasty.


MERGE statement:

MERGE INTO Item t

USING (
   SELECT
      :ItemName AS ItemName,
      :MemberOnly AS MemberOnly,
      :Description AS Description
   FROM DUAL
) s

ON (
   t.ItemName = s.ItemName
)

WHEN MATCHED THEN
   UPDATE
   SET
      t.MemberOnly = s.MemberOnly,
      t.Description = s.Description

WHEN NOT MATCHED THEN
   INSERT (
      t.ItemName,
      t.MemberOnly,
      t.Description
   )
   VALUES (
      s.ItemName,
      s.MemberOnly,
      s.Description
   )
```

The *Item* table is loaded from the database into a `pandas.DataFrame` (`df`). Two rows are modified and two new rows are added. *`DatabaseManager.merge_df()`* is called with `df` and `dry_run=True`. The parameter `merge_cols='all'` includes all columns from `df` in the MERGE statement, which is the default. It also accepts a list of column names to only include a subset of the columns. `merge_index_cols=False` excludes the index column from the statement, which is also the default.

In the returned MERGE statement the `t` (target) alias refers to the *Item* table in the database and the `s` (source) alias to `df`. When a value of the *ItemName* column in the database matches a value from the *ItemName* column of `df` an UPDATE statement is executed to update the *MemberOnly* and *Description* columns. The *ItemName* column does not get updated since it is part of the ON clause and would mean an update to the same value. When there is no match the values of the columns *ItemName*, *MemberOnly* and *Description* are inserted into their respective column counterparts of the *Item* table.

Oracle supports adding a WHERE clause to the UPDATE clause of the WHEN MATCHED THEN part. This can be controlled with the parameter `omit_update_where_clause`, which defaults to `True`. If set to `False` the columns to update will not be updated if their values from `df` are the same as in the database. If at least one value differs the update

will be executed. The next example illustrates this and also uses two columns in the ON clause (*ItemId* and *ItemName*). This time `df` is loaded with a `pandas.MultiIndex`. Setting `merge_index_cols=True` includes all column levels of the `pandas.MultiIndex` in the MERGE statement. You can also supply a list of column level names to only include the desired index levels.

```
# merge_df_omit_update_where_clause.py

from datetime import datetime
from unittest.mock import MagicMock

import pandas as pd
import pandemy

db_sqlite = pandemy.SQLiteDb(file='Runescape.db', must_exist=True)

query = """SELECT * FROM Item ORDER BY ItemId ASC"""

dtypes = {
   'ItemId': 'int8',
   'ItemName': 'string',
   'MemberOnly': 'boolean',
   'Description': 'string'
}

with db_sqlite.engine.connect() as conn:
   df = db_sqlite.load_table(
      sql=query,
      conn=conn,
      index_col=['ItemId', 'ItemName'],
      dtypes=dtypes,
   )

db_oracle = pandemy.OracleDb(
   username='Fred_the_Farmer',
   password='Penguins-sheep-are-not',
   host='fred.farmer.rs',
   port=1234,
   service_name='woollysheep'
)

db_oracle.engine.begin = MagicMock()  # Mock the begin method

with db_oracle.engine.begin() as conn:
   merge_stmt = db_oracle.merge_df(
      df=df,
      table='Item',
      conn=conn,
      on_cols=['ItemId', 'ItemName'],
      merge_cols='all',
      merge_index_cols=True,
      omit_update_where_clause=False,
      dry_run=True
   )
```

(continues on next page)

(continued from previous page)

```
print(f'MERGE statement:\n\n{merge_stmt}')
```

```
$ python merge_df_omit_update_where_clause.py
```

```
MERGE statement:

MERGE INTO Item t

USING (
   SELECT
      :MemberOnly AS MemberOnly,
      :Description AS Description,
      :ItemId AS ItemId,
      :ItemName AS ItemName
   FROM DUAL
) s

ON (
   t.ItemId = s.ItemId AND
   t.ItemName = s.ItemName
)

WHEN MATCHED THEN
   UPDATE
   SET
      t.MemberOnly = s.MemberOnly,
      t.Description = s.Description
   WHERE
      t.MemberOnly <> s.MemberOnly OR
      t.Description <> s.Description

WHEN NOT MATCHED THEN
   INSERT (
      t.MemberOnly,
      t.Description,
      t.ItemId,
      t.ItemName
   )
   VALUES (
      s.MemberOnly,
      s.Description,
      s.ItemId,
      s.ItemName
   )
```

Tip: If the MERGE includes a lot of columns and the statement needs to be tailored to suit a specific use case it is error prone to type all column names into the statement by hand. Using `dry_run=True` is useful to extract a template of the MERGE statement that can be further manually edited and parametrized. An empty `pandas.DataFrame` representing the table the MERGE acts on is enough to get a template statement. The final statement can then be added to

a *SQLContainer* and executed with the *DatabaseManager.execute()* method.

See also:

Diving into Oracle MERGE Statement : A short and informative tutorial.

1.3 API reference

This chapter explains the complete API of Pandemy.

Pandemy consists of two main classes: *DatabaseManager* and *SQLContainer*. Each SQL dialect is implemented as a subclass of *DatabaseManager*. The *SQLContainer* serves as a container of the SQL statements used by the *DatabaseManager* of an application.

1.3.1 DatabaseManager

DatabaseManager is the base class providing the methods to interact with databases.

class pandemy.**DatabaseManager**(*url=None*, *container=None*, *connect_args=None*, *engine_config=None*, *engine=None*, ***kwargs*)

Bases: object

Base class with functionality for managing a database.

Each database type will subclass from *DatabaseManager* and implement the initializer which is specific to each database type. Initialization of a *DatabaseManager* creates the database *engine*, which is used to connect to and interact with the database.

The *DatabaseManager* can be used on its own but with **limited** functionality and the initialization requires a SQLAlchemy URL or Engine, which require some knowledge about SQLAlchemy.

Note: Some methods like *upsert_table()* and *merge_df()* use dialect specific SQL syntax. These methods may not work properly if using the *DatabaseManager* directly. *DatabaseManager* should *only* be used if there is no subclass implemented that matches the desired SQL dialect.

Parameters

- **url** (str or sqlalchemy.engine.URL or None, default None) – A SQLAlchemy connection URL to use for creating the database engine. If None an engine is expected to be supplied to the *engine* parameter.
- **container** (SQLContainer *or None, default None*) – A container of database statements that *DatabaseManager* can use.
- **connect_args** (*dict or None, default None*) – Additional arguments sent to the driver upon connection that further customizes the connection.
- **engine_config** (*dict or None, default None*) – Additional keyword arguments passed to the sqlalchemy.create_engine() function.
- **engine** (sqlalchemy.engine.Engine or None, default None) – A SQLAlchemy Engine to use as the database engine of *DatabaseManager*. If None (the default) the engine will be created from *url*.
- ****kwargs** (*dict*) – Additional keyword arguments that are not used by *DatabaseManager*.

Raises

- *pandemy.CreateConnectionURLError* – If there are errors with *url*.
- *pandemy.CreateEngineError* – If the creation of the database engine fails.
- *pandemy.InvalidInputError* – If *url* and *engine* are specified or are `None` at the same time.

See also:

- *OracleDb* : An Oracle *DatabaseManager*.
- *SQLiteDb* : A SQLite *DatabaseManager*.

Examples

Create an instance of a *DatabaseManager* that connects to a SQLite in-memory database.

```
>>> import pandemy
>>> db = pandemy.DatabaseManager(url='sqlite://')
>>> db
DatabaseManager(
    url=sqlite://,
    container=None,
    connect_args={},
    engine_config={},
    engine=Engine(sqlite://)
)
```

delete_all_records_from_table(*table*, *conn*)

Delete all records from the specified table.

Parameters

- **table** (*str*) – The table to delete all records from.
- **conn** (*sqlalchemy.engine.base.Connection*) – An open connection to the database.

Raises

- *pandemy.DeleteFromTableError* – If data cannot be deleted from the table.
- *pandemy.InvalidTableNameError* – If the supplied table name is invalid.

See also:

- *load_table()* : Load a SQL table into a `pandas.DataFrame`.
- *save_df()* : Save a `pandas.DataFrame` to a table in the database.

Examples

Delete all records from a table in a SQLite in-memory database.

```
>>> import pandas as pd
>>> import pandemy
>>> df = pd.DataFrame(data=[
...         [1, 'Lumbridge General Supplies', 'Lumbridge', 1],
...         [2, 'Varrock General Store', 'Varrock', 2],
...         [3, 'Falador General Store', 'Falador', 3]
...     ],
...     columns=['StoreId', 'StoreName', 'Location', 'OwnerId']
... )
>>> df = df.set_index('StoreId')
>>> df
                         StoreName   Location  OwnerId
StoreId
1        Lumbridge General Supplies  Lumbridge        1
2             Varrock General Store    Varrock        2
3             Falador General Store    Falador        3
>>> db = pandemy.SQLiteDb()  # Create an in-memory database
>>> with db.engine.begin() as conn:
...     db.save_df(df=df, table='Store', conn=conn)
...     df_loaded = db.load_table(sql='Store', conn=conn, index_col='StoreId')
>>> df_loaded
                         StoreName   Location  OwnerId
StoreId
1        Lumbridge General Supplies  Lumbridge        1
2             Varrock General Store    Varrock        2
3             Falador General Store    Falador        3
>>> with db.engine.begin() as conn:
...     db.delete_all_records_from_table(table='Store', conn=conn)
...     df_loaded = db.load_table(sql='Store', conn=conn, index_col='StoreId')
>>> assert df_loaded.empty
>>> df_loaded
Empty DataFrame
Columns: [StoreName, Location, OwnerId]
Index: []
```

execute(*sql*, *conn*, *params=None*)

Execute a SQL statement.

Parameters

- **sql** (*str or sqlalchemy.sql.elements.TextClause*) – The SQL statement to execute. A string value is automatically converted to a `sqlalchemy.sql.elements.TextClause` with the `sqlalchemy.sql.expression.text()` function.
- **conn** (*sqlalchemy.engine.base.Connection*) – An open connection to the database.
- **params** (*dict or list of dict or None, default None*) – Parameters to bind to the SQL statement *sql*. Parameters in the SQL statement should be prefixed by a colon (*:*) e.g. `:myparameter`. Parameters in *params* should *not* contain the colon (`{'myparameter': 'myvalue'}`).

  Supply a list of parameter dictionaries to execute multiple parametrized statements in

the same method call, e.g. `[{'parameter1': 'a string'}, {'parameter2': 100}]`. This is useful for INSERT, UPDATE and DELETE statements.

Returns
A result object from the executed statement.

Return type
sqlalchemy.engine.CursorResult

Raises

- ***pandemy.ExecuteStatementError*** – If an error occurs when executing the statement.
- ***pandemy.InvalidInputError*** – If *sql* is not of type str or `sqlalchemy.sql.elements.TextClause`.

See also:

- `sqlalchemy.engine.Connection.execute()` : The method used for executing the SQL statement.

Examples

To process the result from the method the database connection must remain open after the method is executed i.e. the context manager *cannot* be closed before processing the result:

```
import pandemy

db = SQLiteDb(file='Runescape.db')

with db.engine.connect() as conn:
    result = db.execute('SELECT * FROM StoreSupply', conn=conn)

     for row in result:
         print(row)  # process the result
         ...
```

load_table(*sql*, *conn*, *params=None*, *index_col=None*, *columns=None*, *parse_dates=None*, *datetime_cols_dtype=None*, *datetime_format='%Y-%m-%d %H:%M:%S'*, *localize_tz=None*, *target_tz=None*, *dtypes=None*, *chunksize=None*, *coerce_float=True*)

Load a SQL table into a `pandas.DataFrame`.

Specify a table name or a SQL query to load the `pandas.DataFrame` from. Uses `pandas.read_sql()` function to read from the database.

Parameters

- **sql** (*str or sqlalchemy.sql.elements.TextClause*) – The table name or SQL query.
- **conn** (*sqlalchemy.engine.base.Connection*) – An open connection to the database to use for the query.
- **params** (*dict of str or None, default None*) – Parameters to bind to the SQL query *sql*. Parameters in the SQL query should be prefixed by a colon (*:*) e.g. `:myparameter`. Parameters in *params* should *not* contain the colon (`{'myparameter': 'myvalue'}`).
- **index_col** (*str or sequence of str or None, default None*) – The column(s) to set as the index of the `pandas.DataFrame`.

- **columns** (*list of str or None, default None*) – List of column names to select from the SQL table (only used when *sql* is a table name).
- **parse_dates** (*list or dict or None, default None*) –
  - List of column names to parse into datetime columns.
  - Dict of *{column_name: format string}* where format string is strftime compatible in case of parsing string times, or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
  - Dict of *{column_name: arg dict}*, where the arg dict corresponds to the keyword arguments of `pandas.to_datetime()`. Especially useful with databases without native datetime support, such as SQLite.
- **datetime_cols_dtype** (*{'str', 'int'} or None, default None*) – If the datetime columns of the loaded DataFrame *df* should be converted to string or integer data types. If `None` conversion of datetime columns is omitted, which is the default. When using `'int'` the datetime columns are converted to the number of seconds since the Unix Epoch of 1970-01-01. The timezone of the datetime columns should be in UTC when using `'int'`. You may need to specify *parse_dates* in order for columns be converted into datetime columns depending on the SQL driver.

  New in version 1.2.0.
- **datetime_format** (*str, default r'%Y-%m-%d %H:%M:%S'*) – The datetime (`strftime`) format to use when converting datetime columns to strings.

  New in version 1.2.0.
- **localize_tz** (*str or None, default None*) – The name of the timezone which to localize naive datetime columns into. If None (the default) timezone localization is omitted.
- **target_tz** (*str or None, default None*) – The name of the target timezone to convert the datetime columns into after they have been localized. If None (the default) timezone conversion is omitted.
- **dtypes** (*dict or None, default None*) – Desired data types for specified columns *{'column_name': data type}*. Use pandas or numpy data types or string names of those. If None no data type conversion is performed.
- **chunksize** (*int or None, default None*) – If *chunksize* is specified an iterator of DataFrames will be returned where *chunksize* is the number of rows in each `pandas.DataFrame`. If *chunksize* is supplied timezone localization and conversion as well as dtype conversion cannot be performed i.e. *localize_tz*, *target_tz* and *dtypes* have no effect.
- **coerce_float** (*bool, default True*) – Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets.

Returns

df – `pandas.DataFrame` with the result of the query or an iterator of DataFrames if *chunksize* is specified.

Return type

pandas.DataFrame or Iterator[pandas.DataFrame]

Raises

- ***pandemy.DataTypeConversionError*** – If errors when converting data types using the *dtypes* parameter.
- ***pandemy.InvalidInputError*** – Invalid values or types for input parameters or if the timezone localization or conversion fails.

- ***pandemy.LoadTableError*** – If errors when loading the table using `pandas.read_sql()`.
- ***pandemy.SetIndexError*** – If setting the index of the returned `pandas.DataFrame` fails when *index_col* is specified and chunksize is None.

See also:

- *save_df()* : Save a `pandas.DataFrame` to a table in the database.
- `pandas.read_sql()` : Read SQL query or database table into a `pandas.DataFrame`.
- `pandas.to_datetime()` : The function used for datetime conversion with *parse_dates*.

Examples

When specifying the *chunksize* parameter the database connection must remain open to be able to process the DataFrames from the iterator. The processing *must* occur *within* the context manager:

```
import pandemy

db = pandemy.SQLiteDb(file='Runescape.db')

with db.engine.connect() as conn:
    df_gen = db.load_table(sql='ItemTradedInStore', conn=conn, chunksize=3)

    for df in df_gen:
        print(df)  # Process your DataFrames
        ...
```

manage_foreign_keys(*conn*, *action*)

Manage how the database handles foreign key constraints.

Should be implemented by DatabaseManagers whose SQL dialect supports enabling/disabling checking foreign key constraints. E.g. SQLite.

Parameters

- **conn** (*sqlalchemy.engine.base.Connection*) – An open connection to the database.
- **action** (*str*) – How to handle foreign key constraints in the database.

Raises

- ***pandemy.ExecuteStatementError*** – If changing the handling of foreign key constraint fails.
- ***pandemy.InvalidInputError*** – If invalid input is supplied to *action*.

See also:

- *execute()* : Execute a SQL statement.

Examples

Enable and trigger a foreign key constraint using a SQLite in-memory database.

```
>>> import pandemy
>>> db = pandemy.SQLiteDb()  # Create an in-memory database
>>> with db.engine.begin() as conn:
...     db.execute(
...         sql="CREATE TABLE Owner(OwnerId INTEGER PRIMARY KEY, OwnerName TEXT)
↪",
...         conn=conn
...     )
...     db.execute(
...         sql=(
...             "CREATE TABLE Store("
...             "StoreId INTEGER PRIMARY KEY, "
...             "StoreName TEXT, "
...             "OwnerId INTEGER REFERENCES Owner(OwnerId)"
...             ")"
...         ),
...         conn=conn
...     )
...     db.execute(
...         sql="INSERT INTO Owner(OwnerId, OwnerName) VALUES(1, 'Shop keeper')
↪",
...         conn=conn
...     )
...     db.execute(
...         sql=(
...             "INSERT INTO Store(StoreId, StoreName, OwnerId) "
...             "VALUES(1, 'Lumbridge General Supplies', 2)"
...         ),
...         conn=conn  # OwnerId = 2 is not a valid FOREIGN KEY reference
...     )
...     db.manage_foreign_keys(conn=conn, action='ON')
...     db.execute(
...         sql=(
...             "INSERT INTO Store(StoreId, StoreName, OwnerId) "
...             "VALUES(1, 'Falador General Store', 3)"
...         ),
...         conn=conn  # OwnerId = 3 is not a valid FOREIGN KEY reference
...     )
Traceback (most recent call last):
...
pandemy.exceptions.ExecuteStatementError: IntegrityError: ('(sqlite3.
↪IntegrityError) FOREIGN KEY constraint failed',)
```

merge_df(*df*, *table*, *conn*, *on_cols*, *merge_cols='all'*, *merge_index_cols=False*, *omit_update_where_clause=True*, *chunksize=None*, *nan_to_none=True*, *datetime_cols_dtype=None*, *datetime_format='%Y-%m-%d %H:%M:%S'*, *localize_tz=None*, *target_tz=None*, *dry_run=False*)

Merge data from a `pandas.DataFrame` into a table.

This method executes a combined UPDATE and INSERT statement on a table using the MERGE statement.

The method is similar to *upsert_table()* but it only executes one statement instead of two.

Databases implemented in Pandemy that support the MERGE statement:

- Oracle

The column names of *df* and *table* must match.

New in version 1.2.0.

Parameters

- **df** (*pandas.DataFrame*) – The DataFrame with data to merge into *table*.
- **table** (*str*) – The name of the table to merge data into.
- **conn** (*sqlalchemy.engine.base.Connection*) – An open connection to the database.
- **on_cols** (*Sequence[str] or None*) – The columns from *df* to use in the ON clause to identify how to merge rows into *table*.
- **merge_cols** (*str or Sequence[str] or None, default 'all'*) – The columns of *table* to merge (update or insert) with data from *df*. The default string `'all'` will update all columns. If `None` no columns will be selected for merge. This is useful if only columns of the index of *df* should be updated by specifying *merge_index_cols*.
- **merge_index_cols** (*bool or Sequence[str], default False*) – If the index columns of *df* should be included in the columns to merge. `True` indicates that the index should be included. If the index is a `pandas.MultiIndex` a sequence of strings that maps against the levels to include can be used to only include the desired levels. `False` excludes the index column(s) from being updated which is the default.
- **omit_update_where_clause** (*bool, default True*) – If the WHERE clause of the UPDATE clause should be omitted from the MERGE statement. The WHERE clause is implemented as OR conditions where the target and source columns to update are not equal.

  Databases in Pandemy that support this option are: Oracle

  Example of the SQL generated when `omit_update_where_clause=True`:

```
[...]
WHEN MATCHED THEN
    UPDATE
    SET
        t.IsAdventurer = s.IsAdventurer,
        t.CustomerId = s.CustomerId,
        t.CustomerName = s.CustomerName
     WHERE
        t.IsAdventurer <> s.IsAdventurer OR
        t.CustomerId <> s.CustomerId OR
        t.CustomerName <> s.CustomerName
[...]
```

- **chunksize** (*int or None, default None*) – Divide *df* into chunks and perform the merge in chunks of *chunksize* rows. If None all rows of *df* are processed in one chunk, which is the default. If *df* has many rows dividing it into chunks may increase performance.
- **nan_to_none** (*bool, default True*) – If columns with missing values (NaN values) that are of type `pandas.NA` `pandas.NaT` or `numpy.nan` should be converted to standard Python `None`. Some databases do not support these types in parametrized SQL statements.

- **datetime_cols_dtype** (*{'str', 'int'} or None, default None*) – If the datetime columns of *df* should be converted to string or integer data types before updating the table. If `None` conversion of datetime columns is omitted, which is the default. When using `'int'` the datetime columns are converted to the number of seconds since the Unix Epoch of 1970-01-01. The timezone of the datetime columns should be in UTC when using `'int'`.
- **datetime_format** (*str, default r'%Y-%m-%d %H:%M:%S'*) – The datetime (`strftime`) format to use when converting datetime columns to strings.
- **localize_tz** (*str or None, default None*) – The name of the timezone which to localize naive datetime columns into. If None (the default) timezone localization is omitted.
- **target_tz** (*str or None, default None*) – The name of the target timezone to convert timezone aware datetime columns, or columns that have been localized by *localize_tz*, into. If None (the default) timezone conversion is omitted.
- **dry_run** (*bool, default False*) – Do not execute the merge. Instead return the SQL statement that would have been executed on the database as a string.

Returns

A result object from the executed statement or the SQL statement that would have been executed if *dry_run* is `True`. The result object will be `None` if *df* is empty.

Return type

sqlalchemy.engine.CursorResult or str or None

Raises

- ***pandemy.ExecuteStatementError*** – If an error occurs when executing the MERGE statement.
- ***pandemy.InvalidColumnNameError*** – If a column name of *merge_cols*, *merge_index_cols* or *on_cols* are not among the columns or index of the input DataFrame *df*.
- ***pandemy.InvalidInputError*** – Invalid values or types for input parameters or if the timezone localization or conversion fails.
- ***pandemy.InvalidTableNameError*** – If the supplied table name is invalid.
- ***pandemy.SQLStatementNotSupportedError*** – If the database dialect does not support the MERGE statement.

See also:

- *save_df()* : Save a `pandas.DataFrame` to specified table in the database.
- *upsert_table()* : Update a table with a `pandas.DataFrame` and optionally insert new rows.

Examples

Create a MERGE statement from an empty `pandas.DataFrame` that represents a table in a database by using the parameter `dry_run=True`. The `begin()` method of the `DatabaseManager.engine` is mocked to avoid having to connect to a real database.

```
>>> import pandas as pd
>>> import pandemy
>>> from unittest.mock import MagicMock
>>> df = pd.DataFrame(columns=['ItemId', 'ItemName', 'MemberOnly', 'IsAdventurer
↪'])
```

(continues on next page)

(continued from previous page)

```
>>> df = df.set_index('ItemId')
>>> df
Empty DataFrame
Columns: [ItemName, MemberOnly, IsAdventurer]
Index: []
>>> db = pandemy.OracleDb(
...     username='Fred_the_Farmer',
...     password='Penguins-sheep-are-not',
...     host='fred.farmer.rs',
...     port=1234,
...     service_name='woollysheep'
... )
>>> db.engine.begin = MagicMock()  # Mock the begin method
>>> with db.engine.begin() as conn:
...     merge_stmt = db.merge_df(
...         df=df,
...         table='Item',
...         conn=conn,
...         on_cols=['ItemName'],
...         merge_cols='all',
...         merge_index_cols=False,
...         dry_run=True
...     )
>>> print(merge_stmt)
MERGE INTO Item t

USING (
    SELECT
        :ItemName AS ItemName,
        :MemberOnly AS MemberOnly,
        :IsAdventurer AS IsAdventurer
    FROM DUAL
) s

ON (
    t.ItemName = s.ItemName
)

WHEN MATCHED THEN
    UPDATE
    SET
        t.MemberOnly = s.MemberOnly,
        t.IsAdventurer = s.IsAdventurer

WHEN NOT MATCHED THEN
    INSERT (
        t.ItemName,
        t.MemberOnly,
        t.IsAdventurer
    )
    VALUES (
        s.ItemName,
```

(continues on next page)

(continued from previous page)

```
        s.MemberOnly,
        s.IsAdventurer
    )
```

save_df(*df*, *table*, *conn*, *if_exists='append'*, *index=True*, *index_label=None*, *chunksize=None*, *schema=None*, *dtype=None*, *datetime_cols_dtype=None*, *datetime_format='%Y-%m-%d %H:%M:%S'*, *localize_tz=None*, *target_tz=None*, *method=None*)

Save the pandas.DataFrame *df* to specified table in the database.

If the table does not exist it will be created. If the table already exists the column names of the pandas.DataFrame *df* must match the table column definitions. Uses pandas.DataFrame.to_sql() method to write the pandas.DataFrame to the database.

Parameters

- **df** (*pandas.DataFrame*) – The DataFrame to save to the database.
- **table** (*str*) – The name of the table where to save the pandas.DataFrame.
- **conn** (*sqlalchemy.engine.base.Connection*) – An open connection to the database.
- **if_exists** (*str, {'append', 'replace', 'drop-replace', 'fail'}*) – How to update an existing table in the database:
  - 'append': Append *df* to the existing table.
  - 'replace': Delete all records from the table and then write *df* to the table.
  - 'drop-replace': Drop the table, recreate it, and then write *df* to the table.
  - 'fail': Raise *pandemy.TableExistsError* if the table exists.

  New in version 1.2.0: 'drop-replace'
- **index** (*bool, default True*) – Write pandas.DataFrame index as a column. Uses the name of the index as the column name for the table.
- **index_label** (*str or sequence of str or None, 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 pandas.DataFrame uses a pandas.MultiIndex.
- **chunksize** (*int or None, default None*) – The number of rows in each batch to be written at a time. If None, all rows will be written at once.
- **schema** (*str, None, default None*) – Specify the schema (if database flavor supports this). If None, use default schema.
- **dtype** (*dict or scalar, default None*) – Specifying the data type 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.
- **datetime_cols_dtype** (*{'str', 'int'} or None, default None*) – If the datetime columns of *df* should be converted to string or integer data types before saving the table. If None no conversion of datetime columns is performed, which is the default. When using 'int' the datetime columns are converted to the number of seconds since the Unix Epoch of 1970-01-01. The timezone of the datetime columns should be in UTC when using 'int'.

  New in version 1.2.0.

- **datetime_format** (*str, default r'%Y-%m-%d %H:%M:%S'*) – The datetime (strftime) format to use when converting datetime columns to strings.

  New in version 1.2.0.

- **localize_tz** (*str or None, default None*) – The name of the timezone which to localize naive datetime columns into. If None (the default) timezone localization is omitted.

  New in version 1.2.0.

- **target_tz** (*str or None, default None*) – The name of the target timezone to convert timezone aware datetime columns, or columns that have been localized by *localize_tz*, into. If None (the default) timezone conversion is omitted.

  New in version 1.2.0.

- **method** (*None, 'multi', callable, default None*) – Controls the SQL insertion clause used:

  - **None:**
    Uses standard SQL INSERT clause (one per row).
  - **'multi':**
    Pass multiple values in a single INSERT clause. It uses a special SQL syntax not supported by all backends. This usually provides better performance for analytic databases like Presto and Redshift, but has worse performance for traditional SQL backend if the table contains many columns. For more information check the SQLAlchemy documentation.
  - **callable with signature (pd_table, conn, keys, data_iter):**
    This can be used to implement a more performant insertion method based on specific backend dialect features. See: pandas SQL insertion method.

Raises

- *pandemy.DeleteFromTableError* – If data in the table cannot be deleted when `if_exists='replace'`.
- *pandemy.InvalidInputError* – Invalid values or types for input parameters or if the timezone localization or conversion fails.
- *pandemy.InvalidTableNameError* – If the supplied table name is invalid.
- *pandemy.SaveDataFrameError* – If the `pandas.DataFrame` cannot be saved to the table.
- *pandemy.TableExistsError* – If the table exists and `if_exists='fail'`.

See also:

- *load_table()* : Load a SQL table into a `pandas.DataFrame`.
- *merge_df()* : Merge data from a `pandas.DataFrame` into a table.
- *upsert_table()* : Update a table with a `pandas.DataFrame` and insert new rows if any.
- `pandas.DataFrame.to_sql()` : Write records stored in a DataFrame to a SQL database.
- pandas SQL insertion method : Details about using the *method* parameter.

Examples

Save a `pandas.DataFrame` to a SQLite in-memory database.

```
>>> import pandas as pd
>>> import pandemy
>>> df = pd.DataFrame(data=[
...         [1, 'Lumbridge General Supplies', 'Lumbridge', 1],
...         [2, 'Varrock General Store', 'Varrock', 2],
...         [3, 'Falador General Store', 'Falador', 3]
...     ],
...     columns=['StoreId', 'StoreName', 'Location', 'OwnerId']
... )
>>> df = df.set_index('StoreId')
>>> df
                          StoreName   Location  OwnerId
StoreId
1        Lumbridge General Supplies  Lumbridge        1
2             Varrock General Store    Varrock        2
3             Falador General Store    Falador        3
>>> db = pandemy.SQLiteDb()  # Create an in-memory database
>>> with db.engine.begin() as conn:
...     db.save_df(df=df, table='Store', conn=conn)
```

upsert_table(*df*, *table*, *conn*, *where_cols*, *upsert_cols='all'*, *upsert_index_cols=False*, *update_only=False*, *chunksize=None*, *nan_to_none=True*, *datetime_cols_dtype=None*, *datetime_format='%Y-%m-%d %H:%M:%S'*, *localize_tz=None*, *target_tz=None*, *dry_run=False*)

Update a table with data from a `pandas.DataFrame` and insert new rows if any.

This method executes an UPDATE statement followed by an INSERT statement (UPSERT) to update the rows of a table with a `pandas.DataFrame` and insert new rows. The INSERT statement can be omitted with the *update_only* parameter.

The column names of *df* and *table* must match.

New in version 1.2.0.

Parameters

- **df** (*pandas.DataFrame*) – The DataFrame with data to upsert.
- **table** (*str*) – The name of the table to upsert.
- **conn** (*sqlalchemy.engine.base.Connection*) – An open connection to the database.
- **where_cols** (*Sequence[str]*) – The columns from *df* to use in the WHERE clause to identify the rows to upsert.
- **upsert_cols** (*str or Sequence[str] or None, default 'all'*) – The columns from *table* to upsert with data from *df*. The default string `'all'` will upsert all columns. If `None` no columns will be selected for upsert. This is useful if only columns of the index of *df* should be upserted by specifying *upsert_index_cols*.
- **upsert_index_cols** (*bool or Sequence[str], default False*) – If the index columns of *df* should be included in the columns to upsert. `True` indicates that the index should be included. If the index is a `pandas.MultiIndex` a sequence of

strings that maps against the levels to include can be used to only include the desired levels. `False` excludes the index column(s) from being upserted which is the default.

- **update_only** (*bool, default False*) – If `True` the *table* should only be updated and new rows not inserted. If `False` (the default) perform an update and insert new rows.
- **chunksize** (*int or None, default None*) – Divide *df* into chunks and perform the upsert in chunks of *chunksize* rows. If None all rows of *df* are processed in one chunk, which is the default. If *df* has many rows dividing it into chunks may increase performance.
- **nan_to_none** (*bool, default True*) – If columns with missing values (NaN values) that are of type `pandas.NA` `pandas.NaT` or `numpy.nan` should be converted to standard Python `None`. Some databases do not support these types in parametrized SQL statements.
- **datetime_cols_dtype** (*{'str', 'int'} or None, default None*) – If the datetime columns of *df* should be converted to string or integer data types before upserting the table. SQLite cannot handle datetime objects as parameters and should use this option. If `None` conversion of datetime columns is omitted, which is the default. When using `'int'` the datetime columns are converted to the number of seconds since the Unix Epoch of 1970-01-01. The timezone of the datetime columns should be in UTC when using `'int'`.
- **datetime_format** (*str, default r'%Y-%m-%d %H:%M:%S'*) – The datetime (`strftime`) format to use when converting datetime columns to strings.
- **localize_tz** (*str or None, default None*) – The name of the timezone which to localize naive datetime columns into. If None (the default) timezone localization is omitted.
- **target_tz** (*str or None, default None*) – The name of the target timezone to convert timezone aware datetime columns, or columns that have been localized by *localize_tz*, into. If None (the default) timezone conversion is omitted.
- **dry_run** (*bool, default False*) – Do not execute the upsert. Instead return the SQL statements that would have been executed on the database. The return value is a tuple ('UPDATE statement', 'INSERT statement'). If *update_only* is `True` the INSERT statement will be `None`.

Returns

Result objects from the executed statements or the SQL statements that would have been executed if *dry_run* is `True`. The result objects will be `None` if *df* is empty.

Return type

Tuple[sqlalchemy.engine.CursorResult, Optional[sqlalchemy.engine.CursorResult]] or Tuple[str, Optional[str]] or Tuple[None, None]

Raises

- ***pandemy.ExecuteStatementError*** – If an error occurs when executing the UPDATE and or INSERT statement.
- ***pandemy.InvalidColumnNameError*** – If a column name of *upsert_cols* or *upsert_index_cols* are not among the columns or index of *df*.
- ***pandemy.InvalidInputError*** – Invalid values or types for input parameters or if the timezone localization or conversion fails.
- ***pandemy.InvalidTableNameError*** – If the supplied table name is invalid.

See also:

execute() : Execute a SQL statement.

load_table() : Load a table into a pandas.DataFrame.

merge_df() : Merge data from a pandas.DataFrame into a table.

save_df() : Save a pandas.DataFrame to a table in the database.

Examples

Create a simple table called *Customer* and insert some data from a pandas.DataFrame (df). Change the first row and add a new row to df. Finally upsert the table with df.

```
>>> import pandas as pd
>>> import pandemy
>>> df = pd.DataFrame(data={
...         'CustomerId': [1, 2],
...         'CustomerName': ['Zezima',  'Dr Harlow']
...     }
... )
>>> df = df.set_index('CustomerId')
>>> df
           CustomerName
CustomerId
1                Zezima
2             Dr Harlow
>>> db = pandemy.SQLiteDb()  # Create an in-memory database
>>> with db.engine.begin() as conn:
...     _ = db.execute(
...         sql=(
...             'CREATE TABLE Customer('
...             'CustomerId INTEGER PRIMARY KEY, '
...             'CustomerName TEXT NOT NULL)'
...         ),
...         conn=conn
...     )
...     db.save_df(df=df, table='Customer', conn=conn)
>>> df.loc[1, 'CustomerName'] = 'Baraek'  # Change data
>>> df.loc[3, 'CustomerName'] = 'Mosol Rei'  # Add new data
>>> df
           CustomerName
CustomerId
1                Baraek
2             Dr Harlow
3             Mosol Rei
>>> with db.engine.begin() as conn:
...     _, _ = db.upsert_table(
...         df=df,
...         table='Customer',
...         conn=conn,
...         where_cols=['CustomerId'],
...         upsert_index_cols=True
...     )
```

(continues on next page)

(continued from previous page)

```
...     df_upserted = db.load_table(
...         sql='SELECT * FROM Customer ORDER BY CustomerId ASC',
...         conn=conn,
...         index_col='CustomerId'
...     )
>>> df_upserted
           CustomerName
CustomerId
1                Baraek
2             Dr Harlow
3             Mosol Rei
```

class pandemy.**SQLiteDb**(*file=':memory:'*, *must_exist=False*, *container=None*, *driver='sqlite3'*, *url=None*, *connect_args=None*, *engine_config=None*, *engine=None*, ***kwargs*)

Bases: *DatabaseManager*

A SQLite *DatabaseManager*.

Parameters

- **file** (*str or pathlib.Path, default ':memory:'*) – The path (absolute or relative) to the SQLite database file. The default creates an in-memory database.
- **must_exist** (*bool, default False*) – If `True` validate that *file* exists unless `file=':memory:'`. If it does not exist *pandemy.DatabaseFileNotFoundError* is raised. If `False` the validation is omitted.
- **container** (*SQLContainer or None, default None*) – A container of database statements that the SQLite *DatabaseManager* can use.
- **driver** (*str, default 'sqlite3'*) – The database driver to use. The default is the Python built-in module `sqlite3`, which is also the default driver of SQLAlchemy. When the default is used no driver name is displayed in the connection URL.

  New in version 1.2.0.
- **url** (`str` or `sqlalchemy.engine.URL` or None, default None) – A SQLAlchemy connection URL to use for creating the database engine. It overrides the value of *file* and *must_exist*.

  New in version 1.2.0.
- **connect_args** (*dict or None, default None*) – Additional arguments sent to the driver upon connection that further customizes the connection.

  New in version 1.2.0.
- **engine_config** (*dict or None, default None*) – Additional keyword arguments passed to the `sqlalchemy.create_engine()` function.
- **engine** (`sqlalchemy.engine.Engine` or None, default None) – A SQLAlchemy Engine to use as the database engine of *SQLiteDb*. It overrides the value of *file* and *must_exist*. If specified the value of *url* should be `None`. If `None` (the default) the engine will be created from *file* or *url*.

  New in version 1.2.0.
- ****kwargs** (*dict*) – Additional keyword arguments that are not used by *SQLiteDb*.

Raises

- *pandemy.CreateConnectionURLError* – If there are errors with *url*.
- *pandemy.CreateEngineError* – If the creation of the database engine fails.
- *pandemy.DatabaseFileNotFoundError* – If the database *file* does not exist when `must_exist=True`.
- *pandemy.InvalidInputError* – If the parameters are specified with invalid input.

See also:

- *pandemy.DatabaseManager* : The parent class.
- `sqlalchemy.create_engine()` : The function used to create the database engine.
- SQLite dialect and drivers : The SQLite dialect and drivers in SQLAlchemy.
- SQLite : The SQLite homepage.

property conn_str

Backwards compatibility for the *conn_str* attribute.

The *conn_str* attribute is deprecated in version 1.2.0 and replaced by the url attribute.

manage_foreign_keys(*conn*, *action='ON'*)

Manage how the database handles foreign key constraints.

In SQLite the check of foreign key constraints is not enabled by default.

Parameters

- **conn** (*sqlalchemy.engine.base.Connection*) – An open connection to the database.
- **action** (*{'ON', 'OFF'}*) – Enable ('ON') or disable ('OFF') the check of foreign key constraints.

Raises

- *pandemy.ExecuteStatementError* – If the enabling/disabling of the foreign key constraints fails.
- *pandemy.InvalidInputError* – If invalid input is supplied to *action*.

See also:

- *execute()* : Execute a SQL statement.

Examples

Enable and trigger a foreign key constraint using an in-memory database.

```
>>> import pandemy
>>> db = pandemy.SQLiteDb()  # Create an in-memory database
>>> with db.engine.begin() as conn:
...     db.execute(
...         sql="CREATE TABLE Owner(OwnerId INTEGER PRIMARY KEY, OwnerName␣
↪TEXT)",
...         conn=conn
...     )
...     db.execute(
```

(continues on next page)

(continued from previous page)

```
...         sql=(
...             "CREATE TABLE Store("
...             "StoreId INTEGER PRIMARY KEY, "
...             "StoreName TEXT, "
...             "OwnerId INTEGER REFERENCES Owner(OwnerId)"
...             ")"
...         ),
...         conn=conn
...     )
...     db.execute(
...         sql="INSERT INTO Owner(OwnerId, OwnerName) VALUES(1, 'Shop keeper')
↪",
...         conn=conn
...     )
...     db.execute(
...         sql=(
...             "INSERT INTO Store(StoreId, StoreName, OwnerId) "
...             "VALUES(1, 'Lumbridge General Supplies', 2)"
...         ),
...         conn=conn  # OwnerId = 2 is not a valid FOREIGN KEY reference
...     )
...     db.manage_foreign_keys(conn=conn, action='ON')
...     db.execute(
...         sql=(
...             "INSERT INTO Store(StoreId, StoreName, OwnerId) "
...             "VALUES(1, 'Falador General Store', 3)"
...         ),
...         conn=conn  # OwnerId = 3 is not a valid FOREIGN KEY reference
...     )
Traceback (most recent call last):
...
pandemy.exceptions.ExecuteStatementError: IntegrityError: ('(sqlite3.
↪IntegrityError) FOREIGN KEY constraint failed',)
```

class pandemy.**OracleDb**(*username=None*, *password=None*, *host=None*, *port=None*, *service_name=None*, *sid=None*, *container=None*, *driver='cx_oracle'*, *url=None*, *connect_args=None*, *engine_config=None*, *engine=None*, ***kwargs*)

Bases: *DatabaseManager*

An Oracle *DatabaseManager*.

Requires the cx_Oracle package to be able to create a connection to the database.

To use a DSN connection string specified in the Oracle connection config file, *tnsnames.ora*, set *host* to the desired network service name in *tnsnames.ora* and leave *port*, *service_name* and *sid* as `None`. Using a *tnsnames.ora* file is needed to connect to Oracle Cloud Autononmous Databases.

New in version 1.1.0.

Parameters

- **username** (*str or None, default None*) – The username of the database account. Must be specified if *url* or *engine* are `None`.
- **password** (*str or None, default None*) – The password of the database account. Must be specified if *url* or *engine* are `None`.

- **host** (*str or None, default None*) – The host name or server IP-address where the database is located. Must be specified if *url* or *engine* are `None`.
- **port** (*int or str or None, default None*) – The port the *host* is listening on. The default port of Oracle databases is 1521.
- **service_name** (*str or None, default None*) – The name of the service used for the database connection.
- **sid** (*str or None, default None*) – The SID used for the connection. SID is the name of the database instance on the *host*. Note that *sid* and *service_name* should not be specified at the same time.
- **container** (*SQLContainer or None, default None*) – A container of database statements that *OracleDb* can use.
- **driver** (*str, default 'cx_oracle'*) – The database driver to use.

  New in version 1.2.0.
- **url** (`str` or `sqlalchemy.engine.URL` or None, default None) – A SQLAlchemy connection URL to use for creating the database engine. Specifying *url* overrides the parameters: *username*, *password*, *host port*, *service_name*, *sid* and *driver*. If *url* is specified *engine* should be `None`.
- **connect_args** (*dict or None, default None*) – Additional arguments sent to the driver upon connection that further customizes the connection.
- **engine_config** (*dict or None, default None*) – Additional keyword arguments passed to the `sqlalchemy.create_engine()` function.
- **engine** (`sqlalchemy.engine.Engine` or None, default None) – A SQLAlchemy Engine to use as the database engine of *OracleDb*. If `None` (the default) the engine will be created from the other parameters. When specified it overrides the parameters: *username*, *password*, *host*, *port*, *service_name*, *sid* and *driver*. If specified *url* should be `None`.
- ****kwargs** (*dict*) – Additional keyword arguments that are not used by *OracleDb*.

Raises

- ***pandemy.CreateConnectionURLError*** – If the creation of the connection URL fails.
- ***pandemy.CreateEngineError*** – If the creation of the database engine fails.
- ***pandemy.InvalidInputError*** – If invalid combinations of the parameters are used.

See also:

- *pandemy.DatabaseManager* : The parent class.
- `sqlalchemy.create_engine()` : The function used to create the database engine.
- The cx_Oracle database driver : Details of the cx_Oracle driver and its usage in SQLAlchemy.
- cx_Oracle documentation
- Specifying connect_args : Details about the *connect_args* parameter.
- tnsnames.ora : Oracle connection config file.
- Oracle Cloud Autononmous Databases

Examples

Create an instance of *OracleDb* and connect using a service:

```
>>> db = pandemy.OracleDb(
...     username='Fred_the_Farmer',
...     password='Penguins-sheep-are-not',
...     host='fred.farmer.rs',
...     port=1234,
...     service_name='woollysheep'
... )
>>> str(db)
'OracleDb(oracle+cx_oracle://Fred_the_Farmer:***@fred.farmer.rs:1234?service_
↪name=woollysheep)'
>>> db.username
'Fred_the_Farmer'
>>> db.password
'Penguins-sheep-are-not'
>>> db.host
'fred.farmer.rs'
>>> db.port
1234
>>> db.service_name
'woollysheep'
>>> db.url
oracle+cx_oracle://Fred_the_Farmer:***@fred.farmer.rs:1234?service_name=woollysheep
>>> db.engine
Engine(oracle+cx_oracle://Fred_the_Farmer:***@fred.farmer.rs:1234?service_
↪name=woollysheep)
```

Connect with a DSN connection string using a net service name from a *tnsnames.ora* config file and supply additional connection arguments and engine configuration:

```
>>> import cx_Oracle
>>> connect_args = {
...     'encoding': 'UTF-8',
...     'nencoding': 'UTF-8',
...     'mode': cx_Oracle.SYSDBA,
...     'events': True
... }
>>> engine_config = {
...     'coerce_to_unicode': False,
...     'arraysize': 40,
...     'auto_convert_lobs': False
... }
>>> db = pandemy.OracleDb(
...     username='Fred_the_Farmer',
...     password='Penguins-sheep-are-not',
...     host='my_dsn_name',
...     connect_args=connect_args,
...     engine_config=engine_config
... )
>>> db
OracleDb(
```

(continues on next page)

(continued from previous page)

```
    username='Fred_the_Farmer',
    password='***',
    host='my_dsn_name',
    port=None,
    service_name=None,
    sid=None,
    driver='cx_oracle',
    url=oracle+cx_oracle://Fred_the_Farmer:***@my_dsn_name,
    container=None,
    connect_args={'encoding': 'UTF-8', 'nencoding': 'UTF-8', 'mode': 2, 'events':␣
↪True},
    engine_config={'coerce_to_unicode': False, 'arraysize': 40, 'auto_convert_lobs
↪': False},
    engine=Engine(oracle+cx_oracle://Fred_the_Farmer:***@my_dsn_name)
)
```

If you are familiar with the connection URL syntax of SQLAlchemy you can create an instance of *OracleDb* directly from a URL:

```
>>> url = 'oracle+cx_oracle://Fred_the_Farmer:Penguins-sheep-are-not@my_dsn_name'
>>> db = pandemy.OracleDb(url=url)
>>> db
OracleDb(
    username='Fred_the_Farmer',
    password='***',
    host='my_dsn_name',
    port=None,
    service_name=None,
    sid=None,
    driver='cx_oracle',
    url=oracle+cx_oracle://Fred_the_Farmer:***@my_dsn_name,
    container=None,
    connect_args={},
    engine_config={},
    engine=Engine(oracle+cx_oracle://Fred_the_Farmer:***@my_dsn_name)
)
```

If you already have a database `engine` and would like to use it with *OracleDb* simply create the instance like this:

```
>>> from sqlalchemy import create_engine
>>> url = 'oracle+cx_oracle://Fred_the_Farmer:Penguins-sheep-are-not@fred.farmer.
↪rs:1234/shears'
>>> engine = create_engine(url, coerce_to_unicode=False)
>>> db = pandemy.OracleDb(engine=engine)
>>> db
OracleDb(
    username='Fred_the_Farmer',
    password='***',
    host='fred.farmer.rs',
    port=1234,
    service_name=None,
```

(continues on next page)

(continued from previous page)

```
    sid='shears',
    driver='cx_oracle',
    url=oracle+cx_oracle://Fred_the_Farmer:***@fred.farmer.rs:1234/shears,
    container=None,
    connect_args={},
    engine_config={},
    engine=Engine(oracle+cx_oracle://Fred_the_Farmer:***@fred.farmer.rs:1234/shears)
)
```

This is useful if you have special needs for creating the engine that cannot be accomplished with the engine constructor of *OracleDb*. See for instance the cx_Oracle SessionPool example in the SQLAlchemy docs.

classmethod from_url(*url*, *container=None*, *engine_config=None*)

Create an instance of *OracleDb* from a SQLAlchemy URL.

Deprecated since version 1.2.0: Use the *url* parameter of the normal initializer of *OracleDb* instead.

Parameters

- **url** (*str or sqlalchemy.engine.URL*) – A SQLAlchemy URL to use for creating the database engine.
- **container** (*SQLContainer or None, default None*) – A container of database statements that *OracleDb* can use.
- **engine_config** (*dict or None, default None*) – Additional keyword arguments passed to the `sqlalchemy.create_engine()` function.

Raises

pandemy.CreateConnectionURLError – If *url* is invalid.

Examples

If you are familiar with the connection URL syntax of SQLAlchemy you can create an instance of *OracleDb* directly from a URL:

```
>>> url = 'oracle+cx_oracle://Fred_the_Farmer:Penguins-sheep-are-not@my_dsn_
↪name'
>>> db = pandemy.OracleDb.from_url(url)
>>> db
OracleDb(
    username='Fred_the_Farmer',
    password='***',
    host='my_dsn_name',
    port=None,
    service_name=None,
    sid=None,
    driver='cx_oracle',
    url=oracle+cx_oracle://Fred_the_Farmer:***@my_dsn_name,
    container=None,
    connect_args={},
    engine_config={},
    engine=Engine(oracle+cx_oracle://Fred_the_Farmer:***@my_dsn_name)
)
```

classmethod from_engine(*engine*, *container=None*)

Create an instance of *OracleDb* from a SQLAlchemy Engine.

Deprecated since version 1.2.0: Use the *engine* parameter of the normal initializer of *OracleDb* instead.

Parameters

- **engine** (*sqlalchemy.engine.Engine*) – A SQLAlchemy Engine to use as the database engine of *OracleDb*.
- **container** (*SQLContainer or None, default None*) – A container of database statements that *OracleDb* can use.

Examples

If you already have a database engine and would like to use it with *OracleDb* simply create the instance like this:

```
>>> from sqlalchemy import create_engine
>>> url = 'oracle+cx_oracle://Fred_the_Farmer:Penguins-sheep-are-not@fred.
↪farmer.rs:1234/shears'
>>> engine = create_engine(url, coerce_to_unicode=False)
>>> db = pandemy.OracleDb.from_engine(engine)
>>> db
OracleDb(
    username='Fred_the_Farmer',
    password='***',
    host='fred.farmer.rs',
    port=1234,
    service_name=None,
    sid='shears',
    driver='cx_oracle',
    url=oracle+cx_oracle://Fred_the_Farmer:***@fred.farmer.rs:1234/shears,
    container=None,
    connect_args={},
    engine_config={},
    engine=Engine(oracle+cx_oracle://Fred_the_Farmer:***@fred.farmer.rs:1234/
↪shears)
)
```

This is useful if you have special needs for creating the engine that cannot be accomplished with the engine constructor of *OracleDb*. See for instance the cx_Oracle SessionPool example in the SQLAlchemy docs.

1.3.2 SQLContainer

The *SQLContainer* is a storage container for the SQL statements used by a *DatabaseManager* of an application. It also provides the *replace_placeholders()* method for pre-processing of placeholders in a SQL statement before it is executed on the database.

class pandemy.**SQLContainer**

Bases: object

Base class of a container of SQL statements.

Each SQL-dialect will subclass from *SQLContainer* and *SQLContainer* is never used on its own, but merely provides methods to work with SQL statements.

Each SQL statement is implemented as a class variable.

static replace_placeholders(*stmt*, *placeholders*)

Replace placeholders in a SQL statement.

Replace the placeholders in the SQL statement *stmt* that are specified by the *placeholder* parameter of a *Placeholder* instance, supplied to the *placeholders* parameter, with their respective replacements in the *replacements* parameter of a *Placeholder*. A placeholder in a SQL statement is always prefixed with a colon (*:*) e.g. `:myplaceholder`.

The main purpose of the method is to handle parametrized IN statements with a variable number of values. A single placeholder can be placed in the IN statement and later be replaced by new placeholders that match the length of the *replacements* parameter of a *Placeholder* instance.

The return values *stmt* and *params* can be used as input to the *execute()* and *load_table()* methods of a *DatabaseManager*.

Parameters

- **stmt** (*str*) – The SQL statement in which to replace placeholders.
- **placeholders** (*Placeholder or sequence of* Placeholder) – The replacements for each placeholder in *stmt*.

Returns

- **stmt** (*str*) – The SQL statement after placeholders have been replaced.
- **params** (*dict*) – The new placeholders and their replacement values from the *replacements* parameter of a *Placeholder*. Entries to *params* are only written if the parameter *return_new_placeholders* in a *Placeholder* is set to `True`.

  Example of a return value: `{'v0': 'value1', 'v1': 3.14}`. The new placeholders are always named *v* followed by a sequential number denoting the order (zero-indexed) in which the new placeholder occurs in the returned SQL statement *stmt*.

  The keys of *params* never contain the prefix colon (*:*) that is used in the SQL statement to identify a placeholder.

Raises

pandemy.InvalidInputError – If the replacement values in a *Placeholder* are not valid.

See also:

- *Placeholder* : Container of a placeholder and its replacement values.
- *DatabaseManager.execute()* : Execute a SQL statement.
- *DatabaseManager.load_table()* : Load a SQL table into a `pandas.DataFrame`.

Examples

Replace the placeholders of a SQL statement (`stmt`) with new placeholders and return a mapping of the new placeholders to the desired values (`params`).

```
>>> stmt = 'SELECT * FROM Item WHERE ItemId IN (:itemid);'
>>> p1 = pandemy.Placeholder(placeholder=':itemid',
...                          replacements=[1, 2, 3],
...                          return_new_placeholders=True)
```

(continues on next page)

(continued from previous page)

```
>>> stmt, params = pandemy.SQLContainer.replace_placeholders(stmt=stmt,␣
↪placeholders=p1)
>>> stmt
'SELECT * FROM Item WHERE ItemId IN (:v0, :v1, :v2);'
>>> params
{'v0': 1, 'v1': 2, 'v2': 3}
```

If the SQL statement contains more than one placeholder a sequence of *Placeholder* can be passed.

```
>>> stmt = ('SELECT * FROM Item '
...         'WHERE ItemId IN (:itemid) AND Description LIKE :desc '
...         'ORDER BY :orderby;')
...
>>> p1 = pandemy.Placeholder(placeholder=':itemid',
...                          replacements=[1, 2, 3],
...                          return_new_placeholders=True)
...
>>> p2 = pandemy.Placeholder(placeholder=':desc',
...                          replacements='A%',
...                          return_new_placeholders=True)
...
>>> p3 = pandemy.Placeholder(placeholder=':orderby',
...                          replacements='ItemName DESC',
...                          return_new_placeholders=False)
...
>>> stmt, params = pandemy.SQLContainer.replace_placeholders(stmt=stmt,
...                                                          placeholders=[p1,␣
↪p2, p3])
>>> stmt
'SELECT * FROM Item WHERE ItemId IN (:v0, :v1, :v2) AND Description LIKE :v3␣
↪ORDER BY ItemName DESC;'
>>> params
{'v0': 1, 'v1': 2, 'v2': 3, 'v3': 'A%'}
```

Note: The replacement for the *':orderby'* placeholder is not part of the returned `params` dictionary because `return_new_placeholders=False` for `p3`.

Warning: Replacing *':orderby'* by an arbitrary value that is not a placeholder is not safe against SQL injection attacks the way placeholders are and is therefore discouraged. The feature is there if it is needed, but be aware of its security limitations.

Input to the *SQLContainer.replace_placeholders()* method.

class pandemy.**Placeholder**(*placeholder*, *replacements*, *return_new_placeholders=True*)

Bases: `object`

Container of placeholders and their replacement values for parametrized SQL statements.

The *Placeholder* handles placeholders and their replacement values when building parametrized SQL statements. A SQL placeholder is always prefixed by a colon (*:*) e.g. `:myplaceholder` in the SQL statement. *Placeholder* is used as input to the *SQLContainer.replace_placeholders()* method.

Parameters

- **placeholder** (*str*) – The placeholder to replace in the SQL statement. E.g. ':myplaceholder'.
- **replacements** (*str or int or float or sequence of str or int or float*) – The value(s) to replace *placeholder* with.
- **return_new_placeholders** (*bool, default True*) – If *replacements* should be mapped to new placeholders in the *params* return value of the *SQLContainer.replace_placeholders()* method.

See also:

- *SQLContainer* : A container of SQL statements.

Examples

Creating a *Placeholder* and accessing its attributes.

```
>>> p1 = pandemy.Placeholder(
...     placeholder=':itemid',
...     replacements=[1, 2, 3]
... )
>>> p1
Placeholder(placeholder=':itemid', replacements=[1, 2, 3], return_new_
↪placeholders=True)
>>> p2 = pandemy.Placeholder(
...     placeholder=':itemname',
...     replacements='A%',
...     return_new_placeholders=False
... )
>>> p2
Placeholder(placeholder=':itemname', replacements='A%', return_new_
↪placeholders=False)
>>> p1.placeholder
':itemid'
>>> p2.replacements
'A%'
>>> p2.return_new_placeholders
False
```

1.3.3 Exceptions

The exception hierarchy of Pandemy.

exception pandemy.**PandemyError**(*message*, *data=None*)

Bases: Exception

The base Exception of Pandemy.

Parameters

- **message** (*str*) – The exception message.

- **data** (*Any, default None*) – Optional extra data to to save as an attribute on the instance. Useful to give more details about the cause of the exception.

exception pandemy.**InvalidInputError**(*message*, *data=None*)

Bases: *PandemyError*

Invalid input to a function or method.

exception pandemy.**DatabaseManagerError**(*message*, *data=None*)

Bases: *PandemyError*

Base `Exception` for errors related to the *DatabaseManager* class.

exception pandemy.**CreateConnectionURLError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Error when creating a connection URL to create the database `Engine`.

New in version 1.1.0.

exception pandemy.**CreateEngineError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Error when creating the database `Engine`.

exception pandemy.**DatabaseFileNotFoundError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Error when the file of a SQLite database cannot be found.

exception pandemy.**DataTypeConversionError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Errors when converting data types of columns in a `pandas.DataFrame`.

exception pandemy.**DeleteFromTableError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Errors when deleting data from a table in the database.

exception pandemy.**ExecuteStatementError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Errors when executing a SQL statement with a *DatabaseManager*.

exception pandemy.**InvalidColumnNameError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Errors when supplying an invalid column name to a database operation.

New in version 1.2.0.

exception pandemy.**InvalidTableNameError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Errors when supplying an invalid table name to a database operation.

exception pandemy.**LoadTableError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Errors when loading tables from the database.

exception pandemy.**SaveDataFrameError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Errors when saving a pandas.DataFrame to a table in the database.

exception pandemy.**SetIndexError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Errors when setting an index of a pandas.DataFrame after loading a table from the database.

exception pandemy.**SQLStatementNotSupportedError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Errors when executing a method that triggers a SQL statement not supported by the database dialect.

New in version 1.2.0.

exception pandemy.**TableExistsError**(*message*, *data=None*)

Bases: *DatabaseManagerError*

Errors when saving a pandas.DataFrame to a table and the table already exists.

1.3.4 Attributes

This section describes the special attributes of Pandemy.

pandemy.**__versiontuple__ = (1, 2, 0)**

The version of Pandemy in a comparable form.

Adheres to Semantic Versioning (MAJOR.MINOR.PATCH). Useful for checking if Pandemy is in a certain version range.

Examples

```
>>> pandemy.__versiontuple__
(1, 0, 0)
>>> pandemy.__versiontuple__ > (0, 0, 1) and pandemy.__versiontuple__ < (2, 0, 0)
True
```

pandemy.**__version__ = '1.2.0'**

The Pandemy version string.

pandemy.**__releasedate__ = datetime.date(2023, 2, 6)**

The release date of the version specified in *__versiontuple__*.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

C

D

E

F

I

L

M

O

P

R

S

T

U