Pandemy

Release 1.0.0

Anton Lydell

2022-02-12

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**: 1.0.0 | 2022-02-12
- **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 the source code is hosted on GitHub at: https://github.com/antonlydell/Pandemy

Install Pandemy by running:

```
$ pip install Pandemy
```

Dependencies

The core dependencies of Pandemy are:

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

To work with other databases than SQLite (which is the only supported database in version 1.0.0) additional optional dependencies may need to be installed. Support for Oracle and Microsoft SQL Server databases is planned for the future.

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
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)
   db.save_df(df=df, table='Item', conn=conn)
```

```
$ python overview_save_df.py
```

The database is managed through the *DatabaseManager* class which is this case is the *SQLiteDb* instance. Each SQL dialect will be a subclass of *DatabaseManager*. The creation of the *DatabaseManager* instance creates the database `engine`, which is used to create a connection to the database. The `engine` is created with the `sqlalchemy.create_engine()` function. The connection is automatically closed 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 DataFrame `df` to the table *Item* in the database `db` by using the `pandas.DataFrame.to_sql()` method.

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 *DatabaseManager.load_table()* method takes either a table name or a SQL statement for the `sql` parameter and uses the `pandas.read_sql()` function.

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 string needed to create the `engine` is all handled during the initialization of an instance of *DatabaseManager*. This class is never used on its own instead it serves as the facilitator of functionality for each database dialect.

Database dialects

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

- **SQLite**: *SQLiteDb*.
- **Oracle**: Planned.
- **Microsoft SQL Server**: Planned.

Core functionality

All database 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`.
- *save_df()*: Save a `pandas.DataFrame` to a table in the database.

Examples of using these methods are shown in the *SQLite* section, 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,
```

(continues on next page)

(continued from previous page)

```
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.'}
]
```

(continues on next page)

(continued from previous page)

```
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')
(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'}
```

(continues on next page)

(continued from previous page)

```
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
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.
```

(continues on next page)

(continued from previous page)

```
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.
""")

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.
- *'fail'*: Raise *pandemy.TableExistsError* if the table exists.

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. If the target table does not exist it will be created, which is also how `pandas.DataFrame.to_sql()` operates by default.

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
```

```
                    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.
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 use the table *Customer* from the database *Runescape.db* 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

   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
""")

df = pd.read_csv(filepath_or_buffer=data, sep=';', index_col='CustomerId',
                 parse_dates=['BirthDate'])  # Create a DataFrame

with db.engine.connect() 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',
                            parse_dates=['Birthdate'])
```

(continues on next page)

(continued from previous page)

```
   df_dt_aware = db.load_table(sql='Customer', conn=conn, index_col='CustomerId',
                               parse_dates=['Birthdate'], localize_tz='UTC', target_tz=
↪'CET')

print(f'df:\n{df}\n')
print(f'df_naive:\n{df_naive}\n')
print(f'df_dt_aware:\n{df_dt_aware}')
```

```
$ python load_table_localize_tz.py
```

```
df:
            CustomerName  BirthDate   Residence  IsAdventurer
CustomerId
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

df_naive:
            CustomerName  BirthDate   Residence  IsAdventurer
CustomerId
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

df_dt_aware:
            CustomerName                 BirthDate   Residence  IsAdventurer
CustomerId
1                 Zezima 1990-07-14 02:00:00+02:00     Yanille             1
2              Dr Harlow 1970-01-14 01:00:00+01:00     Varrock             0
3                 Baraek 1968-12-13 01:00:00+01:00     Varrock             0
4             Gypsy Aris 1996-03-24 01:00:00+01:00     Varrock             0
5              Not a Bot 2006-05-31 02:00:00+02:00    Catherby             1
6               Max Pure 2007-08-20 02:00:00+02:00  Port Sarim             1
```

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
```

(continues on next page)

(continued from previous page)

```
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
```

```
                    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.
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.
```

(continues on next page)

(continued from previous page)

```
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
    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)
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
      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;
```

(continues on next page)

(continued from previous page)

```
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.

1.3 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**(*container=None*, *engine_config=None*, ***kwargs*)

Bases: abc.ABC

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. *DatabaseManager* is never used on its own, but merely provides the methods to interact with the database to its subclasses.

Initialization of a *DatabaseManager* creates the connection string (*conn_str*) and the database *engine*, which are used to connect to and interact with the database. These are available as attributes on the instance. The initializer can contain any number of parameters needed to connect to the database and should always support *container*, *engine_config* and ***kwargs*.

Parameters

- **container** (SQLContainer *or None, default None*) – A container of database statements that can be used by the *DatabaseManager*.
- **engine_config** (*dict or None*) – Additional keyword arguments passed to the sqlalchemy.create_engine() function.

- ****kwargs** (*dict*) – Additional keyword arguments that are not used by the *DatabaseManager*.

Attributes

- **conn_str** (*str*) – The connection string for connecting to the database.
- **engine** (sqlalchemy.engine.Engine) – The engine for interacting with the database.

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

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

See also:

- sqlalchemy.engine.Connection.execute() : The method used for executing the SQL statement.
- sqlalchemy.engine.CursorResult : The return type from the method.

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='mydb.db')

with db.engine.connect() as conn:
    result = db.execute('SELECT * FROM MyTable;', 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*, *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 as dates.
  - 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.
- **localize_tz** (*str or None, default None*) – Localize naive datetime columns of the returned `pandas.DataFrame` to specified timezone. If None no localization is performed.
- **target_tz** (*str or None, default None*) – The timezone to convert the datetime columns of the returned `pandas.DataFrame` into after they have been localized. If None no conversion is performed.

- **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.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.
- ***pandemy.DataTypeConversionError*** – If errors when converting data types using the *dtypes* parameter.

See also:

- `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='mydb.db')

with db.engine.connect() as conn:
    df_gen = db.load_table(sql='MyTableName', 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.InvalidInputError*** – If invalid input is supplied to *action*.
- ***pandemy.ExecuteStatementError*** – If changing the handling of foreign key constraint fails.

save_df(*df*, *table*, *conn*, *if_exists='append'*, *index=True*, *index_label=None*, *chunksize=None*, *schema=None*, *dtype=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', 'fail'}*) – How to update an existing table in the database:
  - 'append': Append the pandas.DataFrame to the existing table.
  - 'replace': Delete all records from the table and then write the pandas.DataFrame to the table.
  - 'fail': Raise *pandemy.TableExistsError* if the table exists.
- **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.
- **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.TableExistsError* – If the table exists and `if_exists='fail'`.
- *pandemy.DeleteFromTableError* – If data in the table cannot be deleted when `if_exists='replace'`.
- *pandemy.InvalidInputError* – Invalid values or types for input parameters.
- *pandemy.InvalidTableNameError* – If the supplied table name is invalid.
- *pandemy.SaveDataFrameError* – If the `pandas.DataFrame` cannot be saved to the table.

See also:

- `pandas.DataFrame.to_sql()` : Write records stored in a DataFrame to a SQL database.
- pandas SQL insertion method : Details about using the *method* parameter.

SQLiteDb

`SQLiteDb` is a *DatabaseManager* for the flexible file based database SQLite.

class pandemy.**SQLiteDb**(*file=':memory:'*, *must_exist=False*, *container=None*, *engine_config=None*, ***kwargs*)

Bases: *pandemy.databasemanager.DatabaseManager*

A SQLite *DatabaseManager*.

Parameters

- **file** (*str or pathlib.Path, default ':memory:'*) – The file (with path) to the SQLite database. 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.
- **engine_config** (*dict or None*) – Additional keyword arguments passed to the `sqlalchemy.create_engine()` function.
- ****kwargs** (*dict*) – Additional keyword arguments that are not used by *SQLiteDb*.

Raises

- *pandemy.InvalidInputError* – If invalid types are supplied to *file*, *must_exist* and *container*.
- *pandemy.DatabaseFileNotFoundError* – If the database *file* does not exist when `must_exist=True`.
- *pandemy.CreateEngineError* – If the creation of the database engine fails.

See also:

- *pandemy.DatabaseManager* : The parent class.

- sqlalchemy.create_engine() : The function used to create the database engine.
- SQLAlchemy SQLite dialect : Implementation of the SQLite dialect in SQLAlchemy.
- SQLite : The SQLite homepage.

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

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)
>>> 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',
```

(continues on next page)

(continued from previous page)

```
...                           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.

Placeholder

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

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

Bases: `tuple`

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

The *Placeholder* `namedtuple` 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.

Examples

Creating a *Placeholder* and accessing its attributes.

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

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: *pandemy.exceptions.PandemyError*

Invalid input to a function or method.

exception pandemy.**DatabaseManagerError**(*message*, *data=None*)
Bases: *pandemy.exceptions.PandemyError*

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

exception pandemy.**CreateEngineError**(*message*, *data=None*)
Bases: *pandemy.exceptions.DatabaseManagerError*

Error when creating the database Engine.

exception pandemy.**DatabaseFileNotFoundError**(*message*, *data=None*)
Bases: *pandemy.exceptions.DatabaseManagerError*

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

exception pandemy.**DataTypeConversionError**(*message*, *data=None*)
Bases: *pandemy.exceptions.DatabaseManagerError*

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

exception pandemy.**DeleteFromTableError**(*message*, *data=None*)
Bases: *pandemy.exceptions.DatabaseManagerError*

Errors when deleting data from a table in the database.

exception pandemy.**ExecuteStatementError**(*message*, *data=None*)
Bases: *pandemy.exceptions.DatabaseManagerError*

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

exception pandemy.**InvalidTableNameError**(*message*, *data=None*)
Bases: *pandemy.exceptions.DatabaseManagerError*

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

exception pandemy.**LoadTableError**(*message*, *data=None*)
Bases: *pandemy.exceptions.DatabaseManagerError*

Errors when loading tables from the database.

exception pandemy.**SaveDataFrameError**(*message*, *data=None*)
Bases: *pandemy.exceptions.DatabaseManagerError*

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

exception pandemy.**SetIndexError**(*message*, *data=None*)
Bases: *pandemy.exceptions.DatabaseManagerError*

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

exception pandemy.**TableExistsError**(*message*, *data=None*)
Bases: *pandemy.exceptions.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, 0, 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__ = '1.0.0'**
The Pandemy version string.

pandemy.**__releasedate__ = datetime.date(2022, 2, 12)**
The release date of the version specified in *__versiontuple__*.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX