SQLAlchemy Migrate Documentation

Release 0.6

Evan Rosson, Jan Dittberner, Domen Kožar

July 11, 2010

CONTENTS

1 Download and Development **3**
- 1.1 Download . . . 3
- 1.2 Development . . . 3

2 Dialect support **5**

3 User guide **9**
- 3.1 Database schema versioning workflow . . . 9
- 3.2 Database schema migrations . . . 16
- 3.3 Repository migration (0.4.5 -> 0.5.4) . . . 20

4 API Documentation **21**
- 4.1 Module `migrate.changeset` – Schema migration API . . . 21
- 4.2 Module `migrate.versioning` – Database versioning and repository management . . . 31

5 Changelog **41**
- 5.1 0.6 (11.07.2010) . . . 41
- 5.2 0.5.4 . . . 42
- 5.3 0.5.3 . . . 42
- 5.4 0.5.2 . . . 42
- 5.5 0.5.1.2 . . . 43
- 5.6 0.5.1.1 . . . 43
- 5.7 0.5.1 . . . 43
- 5.8 0.4.5 . . . 43
- 5.9 0.4.4 . . . 43
- 5.10 0.4.3 . . . 44
- 5.11 0.4.2 . . . 44
- 5.12 0.4.1 . . . 44
- 5.13 0.4.0 . . . 44
- 5.14 0.3 . . . 44
- 5.15 0.2.3 . . . 44
- 5.16 0.2.2 . . . 44
- 5.17 0.2.1 . . . 45

6 Indices and tables **47**

Module Index **49**

Index **51**

Author Evan Rosson

Maintainer Domen Kožar <domenNO@SPAMdev.si>

Source code http://code.google.com/p/sqlalchemy-migrate/issues/list

Issues http://code.google.com/p/sqlalchemy-migrate/

Generated July 11, 2010

Version 0.6

Overview

Inspired by Ruby on Rails' migrations, SQLAlchemy Migrate provides a way to deal with database schema changes in SQLAlchemy projects.
Migrate was started as part of Google's Summer of Code by Evan Rosson, mentored by Jonathan LaCour.
The project was taken over by a small group of volunteers when Evan had no free time for the project. It is now hosted as a Google Code project. During the hosting change the project was renamed to SQLAlchemy Migrate.
Currently, sqlalchemy-migrate supports Python versions from 2.4 to 2.6. SQLAlchemy Migrate 0.6.0 supports SQLAlchemy both 0.5.x and 0.6.x branches.

Warning: Version **0.6** breaks backward compatability, please read *changelog* for more info.

CHAPTER

ONE

DOWNLOAD AND DEVELOPMENT

1.1 Download

You can get the latest version of SQLAlchemy Migrate from the project's download page, the cheese shop, pip or via easy_install:

```
easy_install sqlalchemy-migrate
```

or:

```
pip install sqlalchemy-migrate
```

You should now be able to use the *migrate* command from the command line:

```
migrate
```

This should list all available commands. `migrate help COMMAND` will display more information about each command.

If you'd like to be notified when new versions of SQLAlchemy Migrate are released, subscribe to migrate-announce.

1.2 Development

Migrate's Mercurial repository is located at Google Code.

To get the latest trunk:

```
hg clone http://sqlalchemy-migrate.googlecode.com/hg/
```

Patches should be submitted to the issue tracker.

We use hudson Continuous Integration tool to help us run tests on all databases that migrate supports.

CHAPTER

TWO

DIALECT SUPPORT

Operation / Dialect	*sqlite*	*postgres*	*mysql*	*oracle*	*firebird*	mssql
ALTER TABLE RENAME TABLE	yes	yes	yes	yes	no	not supported
ALTER TABLE RENAME COLUMN	yes (workaround) [1]	yes	yes	yes	yes	not supported
ALTER TABLE ADD COLUMN	yes (with limitations) [2]	yes	yes	yes	yes	not supported
ALTER TABLE DROP COLUMN	yes (workaround) [5]	yes	yes	yes	yes	not supported
ALTER TABLE ALTER COLUMN	yes (workaround) [5]	yes	yes	yes (with limitations) [3]	yes [4]	not supported
ALTER TABLE ADD CONSTRAINT	no	yes	yes	yes	yes	not supported
ALTER TABLE DROP CONSTRAINT	no	yes	yes	yes	yes	not supported
RENAME INDEX	no	yes	no	yes	yes	not supported

[1] Table is renamed to temporary table, new table is created followed by INSERT statements.

[2] Visit http://www.sqlite.org/lang_altertable.html for more information.

[3] You can not change datatype or rename column if table has NOT NULL data, see http://blogs.x2line.com/al/archive/2005/08/30/1231.aspx for more information.

[4] Changing nullable is not supported

CHAPTER

THREE

USER GUIDE

SQLAlchemy Migrate is split into two parts, database schema versioning (`migrate.versioning`) and database migration management (`migrate.changeset`). The versioning API is available as the *migrate* command.

3.1 Database schema versioning workflow

SQLAlchemy migrate provides the `migrate.versioning` API that is also available as the *migrate* command.

Purpose of this package is frontend for migrations. It provides commands to manage migrate repository and database selection aswell as script versioning.

3.1.1 Project setup

Create a change repository

To begin, we'll need to create a *repository* for our project.

All work with repositories is done using the *migrate* command. Let's create our project's repository:

```
$ migrate create my_repository "Example project"
```

This creates an initially empty repository relative to current directory at my_repository/ named *Example project*.

The repository directory contains a sub directory `versions` that will store the *schema versions*, a configuration file `migrate.cfg` that contains *repository configuration* and a script *manage.py* that has the same functionality as the *migrate* command but is preconfigured with repository specific parameters.

Note: Repositories are associated with a single database schema, and store collections of change scripts to manage that schema. The scripts in a repository may be applied to any number of databases. Each repository has an unique name. This name is used to identify the repository we're working with.

Version control a database

Next we need to declare database to be under version control. Information on a database's version is stored in the database itself; declaring a database to be under version control creates a table named **migrate_version** and associates it with your repository.

The database is specified as a SQLAlchemy database url.

```
$ python my_repository/manage.py version_control sqlite:///project.db
```

We can have any number of databases under this repository's version control.

Each schema has a version that SQLAlchemy Migrate manages. Each change script applied to the database increments this version number. You can see a database's current version:

```
$ python my_repository/manage.py db_version sqlite:///project.db
0
```

A freshly versioned database begins at version 0 by default. This assumes the database is empty. (If this is a bad assumption, you can specify the version at the time the database is declared under version control, with the "version_control" command.) We'll see that creating and applying change scripts changes the database's version number.

Similarly, we can also see the latest version available in a repository with the command:

```
$ python my_repository/manage.py version
0
```

We've entered no changes so far, so our repository cannot upgrade a database past version 0.

Project management script

Many commands need to know our project's database url and repository path - typing them each time is tedious. We can create a script for our project that remembers the database and repository we're using, and use it to perform commands:

```
$ migrate manage manage.py --repository=my_repository --url=sqlite:///project.db
$ python manage.py db_version
0
```

The script manage.py was created. All commands we perform with it are the same as those performed with the *migrate* tool, using the repository and database connection entered above. The difference between the script `manage.py` in the current directory and the script inside the repository is, that the one in the current directory has the database URL preconfigured.

Note: Parameters specified in manage.py should be the same as in *versioning api*. Preconfigured parameter should just be omitted from *migrate* command.

3.1.2 Making schema changes

All changes to a database schema under version control should be done via change scripts - you should avoid schema modifications (creating tables, etc.) outside of change scripts. This allows you to determine what the schema looks like based on the version number alone, and helps ensure multiple databases you're working with are consistent.

Create a change script

Our first change script will create a simple table

```
account = Table('account', meta,
                Column('id', Integer, primary_key=True),
                Column('login', String(40)),
```

```
                Column('passwd', String(40)),
)
```

This table should be created in a change script. Let's create one:

```
$ python manage.py script "Add account table"
```

This creates an empty change script at `my_repository/versions/001_Add_account_table.py`. Next, we'll edit this script to create our table.

Edit the change script

Our change script predefines two functions, currently empty: `upgrade()` and `downgrade()`. We'll fill those in

```
from sqlalchemy import *
from migrate import *

meta = MetaData()

account = Table('account', meta,
    Column('id', Integer, primary_key=True),
    Column('login', String(40)),
    Column('passwd', String(40)),
)

def upgrade(migrate_engine):
    meta.bind = migrate_engine
    account.create()

def downgrade(migrate_engine):
    meta.bind = migrate_engine
    account.drop()
```

As you might have guessed, `upgrade()` upgrades the database to the next version. This function should contain the *schema changes* we want to perform (in our example we're creating a table).

`downgrade()` should reverse changes made by `upgrade()`. You'll need to write both functions for every change script. (Well, you don't *have* to write downgrade, but you won't be able to revert to an older version of the database or test your scripts without it.)

Note: As you can see, **migrate_engine** is passed to both functions. You should use this in your change scripts, rather than creating your own engine.

> **Warning:** You should be very careful about importing files from the rest of your application, as your change scripts might break when your application changes. More about writing scripts with consistent behavior.

Test the change script

Change scripts should be tested before they are committed. Testing a script will run its `upgrade()` and `downgrade()` functions on a specified database; you can ensure the script runs without error. You should be testing on a test database - if something goes wrong here, you'll need to correct it by hand. If the test is successful, the database should appear unchanged after `upgrade()` and `downgrade()` run.

To test the script:

```
$ python manage.py test
Upgrading... done
Downgrading... done
Success
```

Our script runs on our database (`sqlite:///project.db`, as specified in manage.py) without any errors.

Our repository's version is:

```
$ python manage.py version
1
```

> **Warning:** test command executes actual script, be sure you are NOT doing this on production database.

Upgrade the database

Now, we can apply this change script to our database:

```
$ python manage.py upgrade
0 -> 1... done
```

This upgrades the database (`sqlite:///project.db`, as specified when we created manage.py above) to the latest available version. (We could also specify a version number if we wished, using the `--version` option.) We can see the database's version number has changed, and our table has been created:

```
$ python manage.py db_version
1
$ sqlite3 project.db
sqlite> .tables
account migrate_version
```

Our account table was created - success! As our application evolves, we can create more change scripts using a similar process.

3.1.3 Writing change scripts

By default, change scripts may do anything any other SQLAlchemy program can do.

SQLAlchemy Migrate extends SQLAlchemy with several operations used to change existing schemas - ie. `ALTER TABLE` stuff. See *changeset* documentation for details.

Writing scripts with consistent behavior

Normally, it's important to write change scripts in a way that's independent of your application - the same SQL should be generated every time, despite any changes to your app's source code. You don't want your change scripts' behavior changing when your source code does.

> **Warning: Consider the following example of what NOT to do**

Let's say your application defines a table in the `model.py` file:

```
from sqlalchemy import *

meta = MetaData()
table = Table('mytable', meta,
    Column('id', Integer, primary_key=True),
)
```

... and uses this file to create a table in a change script:

```
from sqlalchemy import *
from migrate import *
import model

def upgrade(migrate_engine):
    model.meta.bind = migrate_engine

def downgrade(migrate_engine):
    model.meta.bind = migrate_engine
    model.table.drop()
```

This runs successfully the first time. But what happens if we change the table definition in `model.py`?

```
from sqlalchemy import *

meta = MetaData()
table = Table('mytable', meta,
    Column('id', Integer, primary_key=True),
    Column('data', String(42)),
)
```

We'll create a new column with a matching change script

```
from sqlalchemy import *
from migrate import *
import model

def upgrade(migrate_engine):
    model.meta.bind = migrate_engine
    model.table.create()

def downgrade(migrate_engine):
    model.meta.bind = migrate_engine
    model.table.drop()
```

This appears to run fine when upgrading an existing database - but the first script's behavior changed! Running all our change scripts on a new database will result in an error - the first script creates the table based on the new definition, with both columns; the second cannot add the column because it already exists.

To avoid the above problem, you should copy-paste your table definition into each change script rather than importing parts of your application.

Note: Sometimes it is enough to just reflect tables with SQLAlchemy instead of copy-pasting - but remember, explicit is better than implicit!

Writing for a specific database

Sometimes you need to write code for a specific database. Migrate scripts can run under any database, however - the engine you're given might belong to any database. Use engine.name to get the name of the database you're working with

```
>>> from sqlalchemy import *
>>> from migrate import *
>>>
>>> engine = create_engine('sqlite:///:memory:')
>>> engine.name
'sqlite'
```

Writings .sql scripts

You might prefer to write your change scripts in SQL, as .sql files, rather than as Python scripts. SQLAlchemy-migrate can work with that:

```
$ python manage.py version
1
$ python manage.py script_sql postgres
```

This creates two scripts `my_repository/versions/002_postgresql_upgrade.sql` and `my_repository/versions/002_postgresql_downgrade.sql`, one for each *operation*, or function defined in a Python change script - upgrade and downgrade. Both are specified to run with Postgres databases - we can add more for different databases if we like. Any database defined by SQLAlchemy may be used here - ex. sqlite, postgres, oracle, mysql...

3.1.4 Command line usage

migrate command is used for API interface. For list of commands and help use:

```
$ migrate --help
```

migrate command exectues `main()` function. For ease of usage, generate your own *project management script*, which calls `main()` function with keywords arguments. You may want to specify *url* and *repository* arguments which almost all API functions require.

If api command looks like:

```
$ migrate downgrade URL REPOSITORY VERSION [--preview_sql|--preview_py]
```

and you have a project management script that looks like

```
from migrate.versioning.shell import main

main(url='sqlite://', repository='./project/migrations/')
```

you have first two slots filed, and command line usage would look like:

```
# preview Python script
$ migrate downgrade 2 --preview_py

# downgrade to version 2
$ migrate downgrade 2
```

Changed in version 0.5.4: Command line parsing refactored: positional parameters usage Whole command line parsing was rewriten from scratch with use of OptionParser. Options passed as kwargs to `main()` are now parsed correctly. Options are passed to commands in the following priority (starting from highest):

- optional (given by `--some_option` in commandline)
- positional arguments
- kwargs passed to migrate.versioning.shell.main

3.1.5 Python API

All commands available from the command line are also available for your Python scripts by importing `migrate.versioning.api`. See the `migrate.versioning.api` documentation for a list of functions; function names match equivalent shell commands. You can use this to help integrate SQLAlchemy Migrate with your existing update process.

For example, the following commands are similar:

From the command line:

```
$ migrate help help
/usr/bin/migrate help COMMAND

    Displays help on a given command.
```

From Python

```
import migrate.versioning.api
migrate.versioning.api.help('help')
# Output:
# %prog help COMMAND
#
#     Displays help on a given command.
```

3.1.6 Experimental commands

Some interesting new features to create SQLAlchemy db models from existing databases and vice versa were developed by Christian Simms during the development of SQLAlchemy-migrate 0.4.5. These features are roughly documented in a thread in migrate-users.

Here are the commands' descriptions as given by `migrate help <command>`:

- `compare_model_to_db`: Compare the current model (assumed to be a module level variable of type sqlalchemy.MetaData) against the current database.
- `create_model`: Dump the current database as a Python model to stdout.
- `make_update_script_for_model`: Create a script changing the old Python model to the new (current) Python model, sending to stdout.

- `upgrade_db_from_model`: Modify the database to match the structure of the current Python model. This also sets the db_version number to the latest in the repository.

As this sections headline says: These features are EXPERIMENTAL. Take the necessary arguments to the commands from the output of `migrate help <command>`.

3.1.7 Repository configuration

SQLAlchemy-migrate repositories can be configured in their migrate.cfg files. The initial configuration is performed by the *migrate create* call explained in *Create a change repository*. The following options are available currently:

- *repository_id* Used to identify which repository this database is versioned under. You can use the name of your project.
- *version_table* The name of the database table used to track the schema version. This name shouldn't already be used by your project. If this is changed once a database is under version control, you'll need to change the table name in each database too.
- *required_dbs* When committing a change script, SQLAlchemy-migrate will attempt to generate the sql for all supported databases; normally, if one of them fails - probably because you don't have that database installed - it is ignored and the commit continues, perhaps ending successfully. Databases in this list MUST compile successfully during a commit, or the entire commit will fail. List the databases your application will actually be using to ensure your updates to that database work properly. This must be a list; example: *['postgres', 'sqlite']*

3.1.8 Customize templates

Users can pass `templates_path` to API functions to provide customized templates path. Path should be a collection of templates, like `migrate.versioning.templates` package directory.

One may also want to specify custom themes. API functions accept `templates_theme` for this purpose (which defaults to *default*)

Example:

```
/home/user/templates/manage $ ls
default.py_tmpl
pylons.py_tmpl

/home/user/templates/manage $ migrate manage manage.py --templates_path=/home/user/templates --templa
```

New in version 0.6.0.

3.2 Database schema migrations

Importing `migrate.changeset` adds some new methods to existing SQLAlchemy objects, as well as creating functions of its own. Most operations can be done either by a method or a function. Methods match SQLAlchemy's existing API and are more intuitive when the object is available; functions allow one to make changes when only the name of an object is available (for example, adding a column to a table in the database without having to load that table into Python).

Changeset operations can be used independently of SQLAlchemy Migrate's *versioning*.

For more information, see the generated documentation for `migrate.changeset`. Summary of supported actions:

- `Create a column`

- Drop a column
- Alter a column (follow a link for list of supported changes)
- Rename a table
- Rename an index
- Create primary key constraint
- Drop primary key constraint
- Create foreign key contraint
- Drop foreign key constraint
- Create unique key contraint
- Drop unique key constraint
- Create check key contraint
- Drop check key constraint

Note: alter_metadata keyword defaults to True.

3.2.1 Column

Given a standard SQLAlchemy table:

```
table = Table('mytable', meta,
   Column('id', Integer, primary_key=True),
)
table.create()
```

Create a column:

```
col = Column('col1', String, default='foobar')
col.create(table, populate_default=True)

# Column is added to table based on its name
assert col is table.c.col1

# col1 is populated with 'foobar' because of 'populate_default'
```

Drop a column:

```
col.drop()
```

Alter a column:

```
col.alter(name='col2')

# Renaming a column affects how it's accessed by the table object
assert col is table.c.col2

# Other properties can be modified as well
col.alter(type=String(42), default="life, the universe, and everything", nullable=False)

# Given another column object, col1.alter(col2), col1 will be changed to match col2
col.alter(Column('col3', String(77), nullable=True))
```

```
assert col.nullable
assert table.c.col3 is col
```

> **Warning:** Since version `0.6.0` passing column into `ChangesetColumn.alter()` is deprecated. Pass in explicit parameters instead.

Note: Since version `0.6.0` you can pass primary_key_name, index_name and unique_name to column.create method to issue ALTER TABLE ADD CONSTRAINT after changing the column. Note for multi columns constraints and other advanced configuration, check *constraint tutorial*.

3.2.2 Table

SQLAlchemy supports table create/drop.

`Rename a table`:

```
table.rename('newtablename')
```

3.2.3 Index

SQLAlchemy supports index create/drop.

`Rename an index`, given an SQLAlchemy `Index` object:

```
index.rename('newindexname')
```

3.2.4 Constraint

SQLAlchemy supports creating/dropping constraints at the same time a table is created/dropped. SQLAlchemy Migrate adds support for creating/dropping `PrimaryKeyConstraint`/`ForeignKeyConstraint`/`CheckConstraint`/`UniqueConstraint` constraints independently. (as ALTER TABLE statements).

The following rundowns are true for all constraints classes:

1. Make sure you do `from migrate.changeset import *` after SQLAlchemy imports since *migrate* does not patch SA's Constraints.
2. You can also use Constraints as in SQLAlchemy. In this case passing table argument explicitly is required:

   ```
   cons = PrimaryKeyConstraint('id', 'num', table=self.table)

   # Create the constraint
   cons.create()

   # Drop the constraint
   cons.drop()
   ```

   or you can pass in column objects (and table argument can be left out):

   ```
   cons = PrimaryKeyConstraint(col1, col2)
   ```

3. Some dialects support CASCADE option when dropping constraints:

```
    cons = PrimaryKeyConstraint(col1, col2)

    # Create the constraint
    cons.create()

    # Drop the constraint
    cons.drop(cascade=True)
```

Note: SQLAlchemy Migrate will try to guess the name of the constraints for databases, but if it's something other than the default, you'll need to give its name. Best practice is to always name your constraints. Note that Oracle requires that you state the name of the constraint to be created/dropped.

Examples

Primary key constraints:

```
from migrate.changeset import *

cons = PrimaryKeyConstraint(col1, col2)

# Create the constraint
cons.create()

# Drop the constraint
cons.drop()
```

Foreign key constraints:

```
from migrate.changeset import *

cons = ForeignKeyConstraint([table.c.fkey], [othertable.c.id])

# Create the constraint
cons.create()

# Drop the constraint
cons.drop()
```

Check constraints:

```
from migrate.changeset import *

cons = CheckConstraint('id > 3', columns=[table.c.id])

# Create the constraint
cons.create()

# Drop the constraint
cons.drop()
```

Unique constraints:

```
from migrate.changeset import *

cons = UniqueConstraint('id', 'age', table=self.table)
```

```
# Create the constraint
cons.create()

# Drop the constraint
cons.drop()
```

3.3 Repository migration (0.4.5 -> 0.5.4)

migrate_repository.py should be used to migrate your repository from a version before 0.4.5 of SQLAlchemy migrate to the current version. Running **migrate_repository.py** is as easy as:

'`migrate_repository.py repository_directory`'

CHAPTER

FOUR

API DOCUMENTATION

4.1 Module `migrate.changeset` – Schema migration API

This module extends SQLAlchemy and provides additional DDL [1] support.

4.1.1 Module `ansisql` – Standard SQL implementation

Extensions to SQLAlchemy for altering existing tables.

At the moment, this isn't so much based off of ANSI as much as things that just happen to work with multiple databases.

class `ANSIColumnDropper` (*dialect, connection, **kw*)
Extends ANSI SQL dropper for column dropping (`ALTER TABLE DROP COLUMN`).

`visit_column` (*column*)
Drop a column from its table.

Parameter *column* (`sqlalchemy.Column`) – the column object

class `ANSIColumnGenerator` (*dialect, connection, **kw*)
Extends ansisql generator for column creation (alter table add col)

`visit_column` (*column*)
Create a column (table already exists).

Parameter *column* (`sqlalchemy.Column` instance) – column object

class `ANSIConstraintCommon` (*dialect, connection, **kw*)
Migrate's constraints require a separate creation function from SA's: Migrate's constraints are created independently of a table; SA's are created at the same time as the table.

`get_constraint_name` (*cons*)
Gets a name for the given constraint.

If the name is already set it will be used otherwise the constraint's `autoname` method is used.

Parameter *cons* – constraint object

class `ANSISchemaChanger` (*dialect, connection, **kw*)
Manages changes to existing schema elements.

Note that columns are schema elements; `ALTER TABLE ADD COLUMN` is in SchemaGenerator.

All items may be renamed. Columns can also have many of their properties - type, for example - changed.

[1] SQL Data Definition Language

Each function is passed a tuple, containing (object, name); where object is a type of object you'd expect for that function (ie. table for visit_table) and name is the object's new name. NONE means the name is unchanged.

start_alter_column(*table, col_name*)
Starts ALTER COLUMN

visit_column(*delta*)
Rename/change a column.

visit_index(*index*)
Rename an index

visit_table(*table*)
Rename a table. Other ops aren't supported.

class AlterTableVisitor(*dialect, connection, **kw*)
Common operations for `ALTER TABLE` statements.

append(*s*)
Append content to the SchemaIterator's query buffer.

execute()
Execute the contents of the SchemaIterator's buffer.

start_alter_table(*param*)
Returns the start of an `ALTER TABLE` SQL-Statement.

Use the param object to determine the table name and use it for building the SQL statement.

Parameter *param* (`sqlalchemy.Column`, `sqlalchemy.Index`, `sqlalchemy.schema.Constraint`, `sqlalchemy.Table`, or string (table name)) – object to determine the table from

4.1.2 Module `constraint` – Constraint schema migration API

This module defines standalone schema constraint classes.

class CheckConstraint(*sqltext, *args, **kwargs*)
Bases: `migrate.changeset.constraint.ConstraintChangeset`, `sqlalchemy.schema.CheckConstraint`

Construct CheckConstraint

Migrate's additional parameters:

Parameters

- *sqltext* (string) – Plain SQL text to check condition
- *columns* (list of Columns instances) – If not name is applied, you must supply this kw to autoname constraint
- *table* (Table instance) – If columns are passed as strings, this kw is required

create(**a, **kw*)
Create the constraint in the database.

Parameters

- *engine* (`sqlalchemy.engine.base.Engine`) – the database engine to use. If this is `None` the instance's engine will be used
- *connection* (`sqlalchemy.engine.base.Connection` instance) – reuse connection istead of creating new one.

drop(**a, **kw*)
Drop the constraint from the database.

Parameters

- *engine* (sqlalchemy.engine.base.Engine) – the database engine to use. If this is None the instance's engine will be used
- *cascade* (bool) – Issue CASCADE drop if database supports it
- *connection* (sqlalchemy.engine.base.Connection instance) – reuse connection istead of creating new one.

Returns Instance with cleared columns

get_children(***kwargs*)
used to allow SchemaVisitor access

class ConstraintChangeset()
Bases: object

Base class for Constraint classes.

create(**a, **kw*)
Create the constraint in the database.

Parameters

- *engine* (sqlalchemy.engine.base.Engine) – the database engine to use. If this is None the instance's engine will be used
- *connection* (sqlalchemy.engine.base.Connection instance) – reuse connection istead of creating new one.

drop(**a, **kw*)
Drop the constraint from the database.

Parameters

- *engine* (sqlalchemy.engine.base.Engine) – the database engine to use. If this is None the instance's engine will be used
- *cascade* (bool) – Issue CASCADE drop if database supports it
- *connection* (sqlalchemy.engine.base.Connection instance) – reuse connection istead of creating new one.

Returns Instance with cleared columns

class ForeignKeyConstraint(*columns, refcolumns, *args, **kwargs*)
Bases: migrate.changeset.constraint.ConstraintChangeset, sqlalchemy.schema.ForeignKeyConstraint

Construct ForeignKeyConstraint

Migrate's additional parameters:

Parameters

- *columns* (list of strings or Column instances) – Columns in constraint
- *refcolumns* (list of strings or Column instances) – Columns that this FK reffers to in another table.
- *table* (Table instance) – If columns are passed as strings, this kw is required

autoname()
Mimic the database's automatic constraint names

create(**a, **kw*)
Create the constraint in the database.

Parameters

- *engine* (sqlalchemy.engine.base.Engine) – the database engine to use. If this is None the instance's engine will be used
- *connection* (sqlalchemy.engine.base.Connection instance) – reuse connection istead of creating new one.

drop(**a, **kw*)
Drop the constraint from the database.

Parameters

- *engine* (sqlalchemy.engine.base.Engine) – the database engine to use. If this is None the instance's engine will be used
- *cascade* (bool) – Issue CASCADE drop if database supports it
- *connection* (sqlalchemy.engine.base.Connection instance) – reuse connection istead of creating new one.

Returns Instance with cleared columns

get_children(***kwargs*)
used to allow SchemaVisitor access

class **PrimaryKeyConstraint**(**cols, **kwargs*)
Bases: migrate.changeset.constraint.ConstraintChangeset, sqlalchemy.schema.PrimaryKeyConstraint

Construct PrimaryKeyConstraint

Migrate's additional parameters:

Parameters

- *cols* (strings or Column instances) – Columns in constraint.
- *table* (Table instance) – If columns are passed as strings, this kw is required

autoname()
Mimic the database's automatic constraint names

create(**a, **kw*)
Create the constraint in the database.

Parameters

- *engine* (sqlalchemy.engine.base.Engine) – the database engine to use. If this is None the instance's engine will be used
- *connection* (sqlalchemy.engine.base.Connection instance) – reuse connection istead of creating new one.

drop(**a, **kw*)
Drop the constraint from the database.

Parameters

- *engine* (sqlalchemy.engine.base.Engine) – the database engine to use. If this is None the instance's engine will be used
- *cascade* (bool) – Issue CASCADE drop if database supports it
- *connection* (sqlalchemy.engine.base.Connection instance) – reuse connection istead of creating new one.

Returns Instance with cleared columns

get_children(**kwargs)
used to allow SchemaVisitor access

class **UniqueConstraint**(*cols, **kwargs)
Bases: migrate.changeset.constraint.ConstraintChangeset, sqlalchemy.schema.UniqueConstraint

Construct UniqueConstraint

Migrate's additional parameters:

Parameters

- *cols* (strings or Column instances) – Columns in constraint.
- *table* (Table instance) – If columns are passed as strings, this kw is required

New in version 0.6.0.

autoname()
Mimic the database's automatic constraint names

create(*a, **kw)
Create the constraint in the database.

Parameters

- *engine* (sqlalchemy.engine.base.Engine) – the database engine to use. If this is None the instance's engine will be used
- *connection* (sqlalchemy.engine.base.Connection instance) – reuse connection istead of creating new one.

drop(*a, **kw)
Drop the constraint from the database.

Parameters

- *engine* (sqlalchemy.engine.base.Engine) – the database engine to use. If this is None the instance's engine will be used
- *cascade* (bool) – Issue CASCADE drop if database supports it
- *connection* (sqlalchemy.engine.base.Connection instance) – reuse connection istead of creating new one.

Returns Instance with cleared columns

get_children(**kwargs)
used to allow SchemaVisitor access

4.1.3 Module databases – Database specific schema migration

This module contains database dialect specific changeset implementations.

Module mysql

MySQL database specific implementations of changeset classes.

Module firebird

Firebird database specific implementations of changeset classes.

class FBColumnDropper(*dialect, connection, **kw*)
Firebird column dropper implementation.

visit_column(*column*)
Firebird supports 'DROP col' instead of 'DROP COLUMN col' syntax

Drop primary key and unique constraints if dropped column is referencing it.

class FBColumnGenerator(*dialect, connection, **kw*)
Firebird column generator implementation.

class FBConstraintDropper(*dialect, connection, **kw*)
Firebird constaint dropper implementation.

cascade_constraint(*constraint*)
Cascading constraints is not supported

class FBConstraintGenerator(*dialect, connection, **kw*)
Firebird constraint generator implementation.

class FBSchemaChanger(*dialect, connection, **kw*)
Firebird schema changer implementation.

visit_table(*table*)
Rename table not supported

Module oracle

Oracle database specific implementations of changeset classes.

Module postgres

PostgreSQL database specific implementations of changeset classes.

class PGColumnDropper(*dialect, connection, **kw*)
PostgreSQL column dropper implementation.

class PGColumnGenerator(*dialect, connection, **kw*)
PostgreSQL column generator implementation.

class PGConstraintDropper(*dialect, connection, **kw*)
PostgreSQL constaint dropper implementation.

class PGConstraintGenerator(*dialect, connection, **kw*)
PostgreSQL constraint generator implementation.

class PGSchemaChanger(*dialect, connection, **kw*)
PostgreSQL schema changer implementation.

Module `sqlite`

SQLite database specific implementations of changeset classes.

class `SQLiteColumnDropper`(*dialect, connection, **kw*)
SQLite ColumnDropper

class `SQLiteColumnGenerator`(*dialect, connection, **kw*)
SQLite ColumnGenerator

`add_foreignkey`(*constraint*)
Does not support ALTER TABLE ADD FOREIGN KEY

class `SQLiteSchemaChanger`(*dialect, connection, **kw*)
SQLite SchemaChanger

`visit_index`(*index*)
Does not support ALTER INDEX

Module `visitor`

Module for visitor class mapping.

`get_dialect_visitor`(*sa_dialect, name*)
Get the visitor implementation for the given dialect.

Finds the visitor implementation based on the dialect class and returns and instance initialized with the given name.

Binds dialect specific preparer to visitor.

`get_engine_visitor`(*engine, name*)
Get the visitor implementation for the given database engine.

Parameters

- *engine* (Engine) – SQLAlchemy Engine
- *name* (string) – Name of the visitor

Returns visitor

`run_single_visitor`(*engine, visitorcallable, element, connection=None, **kwargs*)
Taken from `sqlalchemy.engine.base.Engine._run_single_visitor()` with support for migrate visitors.

4.1.4 Module `exceptions` – Exception definitions

This module provides exception classes.

exception `Error`
Changeset error.

exception `InvalidConstraintError`
Invalid constraint error.

exception `MigrateDeprecationWarning`
Warning for deprecated features in Migrate

exception `NotSupportedError`
Not supported error.

4.1.5 Module `schema` – Additional API to SQLAlchemy for migrations

Schema module providing common schema operations.

create_column(*column, table=None, *p, **kw*)
Create a column, given the table.

API to `ChangesetColumn.create()`.

drop_column(*column, table=None, *p, **kw*)
Drop a column, given the table.

API to `ChangesetColumn.drop()`.

alter_column(**p, **k*)
Alter a column.

Direct API to `ColumnDelta`.

Parameters

- *table* – Table or table name (will issue reflection).
- *engine* – Will be used for reflection.
- *alter_metadata* – Defaults to True. It will alter changes also to objects.

Returns `Columndelta` instance

rename_table(*table, name, engine=None, **kw*)
Rename a table.

If Table instance is given, engine is not used.

API to `ChangesetTable.rename()`.

Parameters

- *table* (string or Table instance) – Table to be renamed.
- *name* (string) – New name for Table.
- *engine* (obj) – Engine instance.

rename_index(*index, name, table=None, engine=None, **kw*)
Rename an index.

If Index instance is given, table and engine are not used.

API to `ChangesetIndex.rename()`.

Parameters

- *index* (string or Index instance) – Index to be renamed.
- *name* (string) – New name for index.
- *table* (string or Table instance) – Table to which Index is reffered.
- *engine* (obj) – Engine instance.

class **ChangesetTable**()
Changeset extensions to SQLAlchemy tables.

create_column(*column, *p, **kw*)
Creates a column.

The column parameter may be a column definition or the name of a column in this table.

API to `ChangesetColumn.create()`

Parameter *column* (Column instance or string) – Column to be created

deregister()
Remove this table from its metadata

drop_column(*column, *p, **kw*)
Drop a column, given its name or definition.

API to `ChangesetColumn.drop()`

Parameter *column* (Column instance or string) – Column to be droped

rename(*name, connection=None, **kwargs*)
Rename this table.

Parameters

- *name* (string) – New name of the table.
- *alter_metadata* (bool) – If True, table will be removed from metadata
- *connection* (`sqlalchemy.engine.base.Connection` instance) – reuse connection istead of creating new one.

class ChangesetColumn()
Changeset extensions to SQLAlchemy columns.

alter(*p, **k*)
Alter a column's definition: `ALTER TABLE ALTER COLUMN`.

Column name, type, server_default, and nullable may be changed here.

Direct API to `alter_column()`

Example:

```
col.alter(name='foobar', type=Integer(), server_default=text("a"))
```

Supported parameters: name, type, primary_key, nullable, server_onupdate, server_default, autoincrement

copy_fixed(*kw*)
Create a copy of this `Column`, with all attributes.

create(*table=None, index_name=None, unique_name=None, primary_key_name=None, populate_default=True, connection=None, **kwargs*)
Create this column in the database.

Assumes the given table exists. `ALTER TABLE ADD COLUMN`, for most databases.

Parameters

- *table* (Table instance) – Table instance to create on.
- *index_name* (string) – Creates `ChangesetIndex` on this column.
- *unique_name* (string) – Creates `UniqueConstraint` on this column.
- *primary_key_name* (string) – Creates `PrimaryKeyConstraint` on this column.
- *alter_metadata* (bool) – If True, column will be added to table object.
- *populate_default* (bool) – If True, created column will be populated with defaults
- *connection* (`sqlalchemy.engine.base.Connection` instance) – reuse connection istead of creating new one.

Returns self

drop(*table=None, connection=None, **kwargs*)
Drop this column from the database, leaving its table intact.

`ALTER TABLE DROP COLUMN`, for most databases.

Parameters

- *alter_metadata* (bool) – If True, column will be removed from table object.
- *connection* (`sqlalchemy.engine.base.Connection` instance) – reuse connection istead of creating new one.

class ChangesetIndex()
Changeset extensions to SQLAlchemy Indexes.

rename(*name, connection=None, **kwargs*)
Change the name of an index.

Parameters

- *name* (string) – New name of the Index.
- *alter_metadata* (bool) – If True, Index object will be altered.
- *connection* (`sqlalchemy.engine.base.Connection` instance) – reuse connection istead of creating new one.

class ChangesetDefaultClause()
Implements comparison between `DefaultClause` instances

class ColumnDelta(**p, **kw*)
Extracts the differences between two columns/column-parameters

May receive parameters arranged in several different ways:

- **current_column, new_column, *p, **kw** Additional parameters can be specified to override column differences.
- **current_column, *p, **kw** Additional parameters alter current_column. Table name is extracted from current_column object. Name is changed to current_column.name from current_name, if current_name is specified.
- **current_col_name, *p, **kw** Table kw must specified.

Parameters

- *table* (string or Table instance) – Table at which current Column should be bound to. If table name is given, reflection will be used.
- *alter_metadata* (bool) – If True, it will apply changes to metadata.
- *metadata* (`MetaData` instance) – If *alter_metadata* is true, metadata is used to reflect table names into
- *engine* (`Engine` instance) – When reflecting tables, either engine or metadata must be specified to acquire engine object.

Returns `ColumnDelta` instance provides interface for altered attributes to *result_column* through `dict()` alike object.

- `ColumnDelta`.result_column is altered column with new attributes
- `ColumnDelta`.current_name is current name of column in db

apply_diffs(*diffs*)
Populate dict and column object with new values

are_column_types_eq(*old_type, new_type*)
Compares two types to be equal

compare_1_column(*col, *p, **k*)
Compares one Column object

compare_2_columns(*old_col, new_col, *p, **k*)
Compares two Column objects

compare_parameters(*current_name, *p, **k*)
Compares Column objects with reflection

process_column(*column*)
Processes default values for column

4.2 Module migrate.versioning – Database versioning and repository management

This package provides functionality to create and manage repositories of database schema changesets and to apply these changesets to databases.

4.2.1 Module api – Python API commands

This module provides an external API to the versioning system. Changed in version 0.6.0: migrate.versioning.api.test() and schema diff functions changed order of positional arguments so all accept *url* and *repository* as first arguments.Changed in version 0.5.4: --preview_sql displays source file when using SQL scripts. If Python script is used, it runs the action with mocked engine and returns captured SQL statements.Changed in version 0.5.4: Deprecated --echo parameter in favour of new migrate.versioning.util.construct_engine() behavior.

db_version(*url, repository, **opts*)
%prog db_version URL REPOSITORY_PATH

Show the current version of the repository with the given connection string, under version control of the specified repository.

The url should be any valid SQLAlchemy connection string.

upgrade(*url, repository, version=None, **opts*)
%prog upgrade URL REPOSITORY_PATH [VERSION] [–preview_py|–preview_sql]

Upgrade a database to a later version.

This runs the upgrade() function defined in your change scripts.

By default, the database is updated to the latest available version. You may specify a version instead, if you wish.

You may preview the Python or SQL code to be executed, rather than actually executing it, using the appropriate 'preview' option.

drop_version_control(*url, repository, **opts*)
%prog drop_version_control URL REPOSITORY_PATH

Removes version control from a database.

help(*cmd=None, **opts*)
%prog help COMMAND

Displays help on a given command.

script(*description, repository, **opts*)
%prog script DESCRIPTION REPOSITORY_PATH

Create an empty change script using the next unused version number appended with the given description.

For instance, manage.py script "Add initial tables" creates: repository/versions/001_Add_initial_tables.py

test(*url, repository, **opts*)
%prog test URL REPOSITORY_PATH [VERSION]

Performs the upgrade and downgrade option on the given database. This is not a real test and may leave the database in a bad state. You should therefore better run the test on a copy of your database.

create(*repository, name, **opts*)
%prog create REPOSITORY_PATH NAME [–table=TABLE]

Create an empty repository at the specified path.

You can specify the version_table to be used; by default, it is 'migrate_version'. This table is created in all version-controlled databases.

manage(*file, **opts*)
%prog manage FILENAME [VARIABLES...]

Creates a script that runs Migrate with a set of default values.

For example:

```
%prog manage manage.py --repository=/path/to/repository --url=sqlite:///project.db
```

would create the script manage.py. The following two commands would then have exactly the same results:

```
python manage.py version
%prog version --repository=/path/to/repository
```

update_db_from_model(*url, repository, model, **opts*)
%prog update_db_from_model URL REPOSITORY_PATH MODEL

Modify the database to match the structure of the current Python model. This also sets the db_version number to the latest in the repository.

NOTE: This is EXPERIMENTAL.

create_model(*url, repository, **opts*)
%prog create_model URL REPOSITORY_PATH [DECLERATIVE=True]

Dump the current database as a Python model to stdout.

NOTE: This is EXPERIMENTAL.

source(*version, dest=None, repository=None, **opts*)
%prog source VERSION [DESTINATION] –repository=REPOSITORY_PATH

Display the Python code for a particular version in this repository. Save it to the file at DESTINATION or, if omitted, send to stdout.

version(*repository, **opts*)
%prog version REPOSITORY_PATH

Display the latest version available in a repository.

make_update_script_for_model (*url, repository, oldmodel, model, **opts*)
%prog make_update_script_for_model URL OLDMODEL MODEL REPOSITORY_PATH

Create a script changing the old Python model to the new (current) Python model, sending to stdout.

NOTE: This is EXPERIMENTAL.

compare_model_to_db (*url, repository, model, **opts*)
%prog compare_model_to_db URL REPOSITORY_PATH MODEL

Compare the current model (assumed to be a module level variable of type sqlalchemy.MetaData) against the current database.

NOTE: This is EXPERIMENTAL.

downgrade (*url, repository, version, **opts*)
%prog downgrade URL REPOSITORY_PATH VERSION [–preview_py|–preview_sql]

Downgrade a database to an earlier version.

This is the reverse of upgrade; this runs the downgrade() function defined in your change scripts.

You may preview the Python or SQL code to be executed, rather than actually executing it, using the appropriate 'preview' option.

version_control (*url, repository, version=None, **opts*)
%prog version_control URL REPOSITORY_PATH [VERSION]

Mark a database as under this repository's version control.

Once a database is under version control, schema changes should only be done via change scripts in this repository.

This creates the table version_table in the database.

The url should be any valid SQLAlchemy connection string.

By default, the database begins at version 0 and is assumed to be empty. If the database is not empty, you may specify a version at which to begin instead. No attempt is made to verify this version's correctness - the database schema is expected to be identical to what it would be if the database were created from scratch.

script_sql (*database, repository, **opts*)
%prog script_sql DATABASE REPOSITORY_PATH

Create empty change SQL scripts for given DATABASE, where DATABASE is either specific ('postgres', 'mysql', 'oracle', 'sqlite', etc.) or generic ('default').

For instance, manage.py script_sql postgres creates: repository/versions/001_postgres_upgrade.sql and repository/versions/001_postgres_postgres.sql

4.2.2 Module `exceptions` – Exception definitions

Provide exception classes for `migrate.versioning`

exception ApiError
Base class for API errors.

exception ControlledSchemaError
Base class for controlled schema errors.

exception DatabaseAlreadyControlledError
Database shouldn't be under version control, but it is

exception DatabaseNotControlledError
Database should be under version control, but it's not.

exception Error
Error base class.

exception InvalidRepositoryError
Invalid repository error.

exception InvalidScriptError
Invalid script error.

exception InvalidVersionError
Invalid version error.

exception KnownError
A known error condition.

exception NoSuchTableError
The table does not exist.

exception PathError
Base class for path errors.

exception PathFoundError
A path with a file was required; found no file.

exception PathNotFoundError
A path with no file was required; found a file.

exception RepositoryError
Base class for repository errors.

exception ScriptError
Base class for script errors.

exception UsageError
A known error condition where help should be displayed.

exception WrongRepositoryError
This database is under version control by another repository.

4.2.3 Module genmodel – ORM Model generator

Code to generate a Python model from a database or differences between a model and database.

Some of this is borrowed heavily from the AutoCode project at: http://code.google.com/p/sqlautocode/

4.2.4 Module pathed – Path utilities

A path/directory class.

class Pathed(*path*)
A class associated with a path/directory tree.

Only one instance of this class may exist for a particular file; __new__ will return an existing instance if possible

class **require_found**(*path*)
Ensures a given path already exists

class **require_notfound**(*path*)
Ensures a given path does not already exist

4.2.5 Module `repository` – Repository management

SQLAlchemy migrate repository management.

class Changeset(*start, *changes, **k*)
A collection of changes to be applied to a database.

Changesets are bound to a repository and manage a set of scripts from that repository.

Behaves like a dict, for the most part. Keys are ordered based on step value.

add(*change*)
Add new change to changeset

keys()
In a series of upgrades x -> y, keys are version x. Sorted.

run(*p, **k*)
Run the changeset scripts

class Repository(*path*)
A project's change script repository

changeset(*database, start, end=None*)
Create a changeset to migrate this database from ver. start to end/latest.

Parameters

- *database* (string) – name of database to generate changeset
- *start* (int) – version to start at
- *end* (int) – version to end at (latest if None given)

Returns `Changeset instance`

class **create**(*path, name, **opts*)
Create a repository at a specified path

class **create_manage_file**(*file_, **opts*)
Create a project management script (manage.py)

Parameters

- *file* – Destination file to be written
- *opts* – Options that are passed to `migrate.versioning.shell.main()`

create_script(*description, **k*)
API to `migrate.versioning.version.Collection.create_new_python_version()`

create_script_sql(*database, **k*)
API to `migrate.versioning.version.Collection.create_new_sql_version()`

class **prepare_config**(*tmpl_dir, name, options=None*)
Prepare a project configuration file for a new project.

Parameters

- *tmpl_dir* (string) – Path to Repository template
- *config_file* (string) – Name of the config file in Repository template

- *name* (string) – Repository name

Returns Populated config file

class **verify**(*path*)
Ensure the target path is a valid repository.

Raises `InvalidRepositoryError`

version(**p, **k*)
API to `migrate.versioning.version.Collection.version`

id
Returns repository id specified in config

latest
API to `migrate.versioning.version.Collection.latest`

version_table
Returns version_table name specified in config

4.2.6 Module `schema` – Migration upgrade/downgrade

Database schema version management.

class ControlledSchema(*engine, repository*)
A database under version control

changeset(*version=None*)
API to Changeset creation.

Uses self.version for start version and engine.name to get database name.

class **compare_model_to_db**(*engine, model, repository*)
Compare the current model against the current database.

class **create**(*engine, repository, version=None*)
Declare a database to be under a repository's version control.

Raises `DatabaseAlreadyControlledError`

Returns `ControlledSchema`

class **create_model**(*engine, repository, declarative=False*)
Dump the current database as a Python model.

drop()
Remove version control from a database.

load()
Load controlled schema version info from DB

update_db_from_model(*model*)
Modify the database to match the structure of the current Python model.

update_repository_table(*startver, endver*)
Update version_table with new information

upgrade(*version=None*)
Upgrade (or downgrade) to a specified version, or latest version.

4.2.7 Module `schemadiff` – ORM Model differencing

Schema differencing support.

class `SchemaDiff`(*model, conn, excludeTables=None, oldmodel=None*)
Differences of model against database.

`compareModelToDatabase`()
Do actual comparison.

`getDiffOfModelAgainstDatabase`(*model, conn, excludeTables=None*)
Return differences of model against database.

Returns object which will evaluate to `True` if there are differences else `False`.

`getDiffOfModelAgainstModel`(*oldmodel, model, conn, excludeTables=None*)
Return differences of model against another model.

Returns object which will evaluate to `True` if there are differences else `False`.

4.2.8 Module `script` – Script actions

class `BaseScript`(*path*)
Base class for other types of scripts. All scripts have the following properties:

source (script.source()) The source code of the script

version (script.version()) The version number of the script

operations (script.operations()) The operations defined by the script: upgrade(), downgrade() or both. Returns a tuple of operations. Can also check for an operation with ex. script.operation(Script.ops.up)

`run`(*engine*)
Core of each BaseScript subclass. This method executes the script.

`source`()

Returns source code of the script.

Return type string

class `verify`(*path*)
Ensure this is a valid script This version simply ensures the script file's existence

Raises `InvalidScriptError`

class `PythonScript`(*path*)
Bases: `migrate.versioning.script.base.BaseScript`

Base for Python scripts

class `create`(*path, **opts*)
Create an empty migration script at specified path

Returns `PythonScript instance`

class `make_update_script_for_model`(*engine, oldmodel, model, repository, **opts*)
Create a migration script based on difference between two SA models.

Parameters

- *repository* (string or `Repository instance`) – path to migrate repository
- *oldmodel* (string or Class) – dotted.module.name:SAClass or SAClass object

- *model* (string or Class) – dotted.module.name:SAClass or SAClass object
- *engine* (Engine instance) – SQLAlchemy engine

Returns Upgrade / Downgrade script

Return type string

preview_sql(*url, step, **args*)
Mocks SQLAlchemy Engine to store all executed calls in a string and runs `PythonScript.run`

Returns SQL file

class **require_found**(*path*)
Ensures a given path already exists

class **require_notfound**(*path*)
Ensures a given path does not already exist

run(*engine, step*)
Core method of Script file. Exectues `update()` or `downgrade()` functions

Parameters

- *engine* (string) – SQLAlchemy Engine
- *step* (int) – Operation to run

source()

Returns source code of the script.

Return type string

class **verify**(*path*)
Ensure this is a valid script This version simply ensures the script file's existence

Raises `InvalidScriptError`

class **verify_module**(*path*)
Ensure path is a valid script

Parameter *path* (string) – Script location

Raises `InvalidScriptError`

Returns Python module

module
Calls `migrate.versioning.script.py.verify_module()` and returns it.

class SqlScript(*path*)
Bases: `migrate.versioning.script.base.BaseScript`

A file containing plain SQL statements.

class **create**(*path, **opts*)
Create an empty migration script at specified path

Returns `SqlScript instance`

class **require_found**(*path*)
Ensures a given path already exists

class **require_notfound**(*path*)
Ensures a given path does not already exist

run(*engine, step=None, executemany=True*)
Runs SQL script through raw dbapi execute call

source()

Returns source code of the script.

Return type string

class **verify**(*path*)
Ensure this is a valid script This version simply ensures the script file's existence

Raises `InvalidScriptError`

4.2.9 Module `shell` – CLI interface

The migrate command-line tool.

main(*argv=None, **kwargs*)
Shell interface to `migrate.versioning.api`.

kwargs are default options that can be overriden with passing –some_option as command line option

Parameter *disable_logging* (bool) – Let migrate configure logging

4.2.10 Module `util` – Various utility functions

class Memoize(*fn*)
Memoize(fn) - an instance which acts like fn but memoizes its arguments Will only work on functions with non-mutable arguments

ActiveState Code 52201

asbool(*obj*)
Do everything to use object as bool

catch_known_errors(*f*)
Decorator that catches known api errors

construct_engine(*engine, **opts*)
New in version 0.5.4. Constructs and returns SQLAlchemy engine.

Currently, there are 2 ways to pass create_engine options to `migrate.versioning.api` functions:

Parameters

- *engine* (string or Engine instance) – connection string or a existing engine
- *engine_dict* (dict) – python dictionary of options to pass to *create_engine*
- *engine_arg_** (string) – keyword parameters to pass to *create_engine* (evaluated with `migrate.versioning.util.guess_obj_type()`)

Returns SQLAlchemy Engine

Note: keyword parameters override `engine_dict` values.

guess_obj_type(*obj*)
Do everything to guess object type from string

Tries to convert to *int*, *bool* and finally returns if not succeded.

load_model(*dotted_name*)
Import module and use module-level variable".

Parameter *dotted_name* – path to model in form of string: `some.python.module:Class`

Changed in version 0.5.4.

with_engine(*f*)
Decorator for `migrate.versioning.api` functions to safely close resources after function usage.

Passes engine parameters to `construct_engine()` and resulting parameter is available as kw['engine'].

Engine is disposed after wrapped function is executed.

4.2.11 Module `version` – Versioning management

class Collection(*path*)
A collection of versioning scripts in a repository

create_new_python_version(*description, **k*)
Create Python files for new version

create_new_sql_version(*database, **k*)
Create SQL files for new version

version(*vernum=None*)
Returns latest Version if vernum is not given. Otherwise, returns wanted version

latest

Returns Latest version in Collection

class Extensions()
A namespace for file extensions

class VerNum(*value*)
A version number that behaves like a string and int at the same time

class Version(*vernum, path, filelist*)
A single version in a collection :param vernum: Version Number :param path: Path to script files :param filelist: List of scripts :type vernum: int, VerNum :type path: string :type filelist: list

add_script(*path*)
Add script to Collection/Version

script(*database=None, operation=None*)
Returns SQL or Python Script

str_to_filename(*s*)
Replaces spaces, (double and single) quotes and double underscores to underscores

CHAPTER

FIVE

CHANGELOG

5.1 0.6 (11.07.2010)

Warning: Backward incompatible changes:

- `api.test()` and schema comparison functions now all accept *url* as first parameter and *repository* as second.
- python upgrade/downgrade scripts do not import *migrate_engine* magically, but recieve engine as the only parameter to function (eg. `def upgrade(migrate_engine):`)
- `Column.alter` does not accept *current_name* anymore, it extracts name from the old column.

5.1.1 Features

- added support for *firebird*
- added option to define custom templates through option `--templates_path` and `--templates_theme`, read more in *tutorial section*
- use Python logging for output, can be shut down by passing `--disable_logging` to `migrate.versioning.shell.main()`
- deprecated *alter_column* comparing of columns. Just use explicit parameter change.
- added support for SQLAlchemy 0.6.x by Michael Bayer
- Constraint classes have *cascade=True* keyword argument to issue `DROP CASCADE` where supported
- added `UniqueConstraint`/`CheckConstraint` and corresponding create/drop methods
- API *url* parameter can also be an `Engine` instance (this usage is discouraged though sometimes necessary)
- code coverage is up to 80% with more than 100 tests
- alter, create, drop column / rename table / rename index constructs now accept *alter_metadata* parameter. If True, it will modify Column/Table objects according to changes. Otherwise, everything will be untouched.
- added *populate_default* bool argument to `Column.create` which issues corresponding UPDATE statements to set defaults after column creation
- `Column.create` accepts *primary_key_name*, *unique_name* and *index_name* as string value which is used as contraint name when adding a column

5.1.2 Bug fixes

- ORM methods now accept *connection* parameter commonly used for transactions
- *server_defaults* passed to `Column.create` are now issued correctly
- use SQLAlchemy quoting system to avoid name conflicts (for issue 32)
- complete refactoring of `ColumnDelta` (fixes issue 23)
- partial refactoring of `changeset` package
- fixed bug when `Column.alter`(server_default='string') was not properly set
- constraints passed to `Column.create` are correctly interpreted (`ALTER TABLE ADD CONSTRAINT` is issued after `ATLER TABLE ADD COLUMN`)
- script names don't break with dot in the name

5.1.3 Documentation

- *dialect support* table was added to documentation
- majoy update to documentation

5.2 0.5.4

- fixed preview_sql parameter for downgrade/upgrade. Now it prints SQL if the step is SQL script and runs step with mocked engine to only print SQL statements if ORM is used. [Domen Kozar]
- use entrypoints terminology to specify dotted model names (module.model:User) [Domen Kozar]
- added engine_dict and engine_arg_* parameters to all api functions (deprecated echo) [Domen Kozar]
- make –echo parameter a bit more forgivable (better Python API support) [Domen Kozar]
- apply patch to refactor cmd line parsing for Issue 54 by Domen Kozar

5.3 0.5.3

- apply patch for Issue 29 by Jonathan Ellis
- fix Issue 52 by removing needless parameters from object.__new__ calls

5.4 0.5.2

- move sphinx and nose dependencies to extras_require and tests_require
- integrate patch for Issue 36 by Kumar McMillan
- fix unit tests
- mark ALTER TABLE ADD COLUMN with FOREIGN KEY as not supported by SQLite

5.5 0.5.1.2

- corrected build

5.6 0.5.1.1

- add documentation in tarball
- add a MANIFEST.in

5.7 0.5.1

- SA 0.5.x support. SQLAlchemy < 0.5.1 not supported anymore.
- use nose instead of py.test for testing
- Added –echo=True option for all commands, which will make the sqlalchemy connection echo SQL statements.
- Better PostgreSQL support, especially for schemas.
- modification to the downgrade command to simplify the calling (old way still works just fine)
- improved support for SQLite
- add support for check constraints (EXPERIMENTAL)
- print statements removed from APIs
- improved sphinx based documentation
- removal of old commented code
- PEP-8 clean code

5.8 0.4.5

- work by Christian Simms to compare metadata against databases
- new repository format
- a repository format migration tool is in migrate/versioning/migrate_repository.py
- support for default SQL scripts
- EXPERIMENTAL support for dumping database to model

5.9 0.4.4

- patch by pwannygoodness for Issue #15
- fixed unit tests to work with py.test 0.9.1
- fix for a SQLAlchemy deprecation warning

5.10 0.4.3

- patch by Kevin Dangoor to handle database versions as packages and ignore their __init__.py files in version.py
- fixed unit tests and Oracle changeset support by Christian Simms

5.11 0.4.2

- package name is sqlalchemy-migrate again to make pypi work
- make import of sqlalchemy's SchemaGenerator work regardless of previous imports

5.12 0.4.1

- setuptools patch by Kevin Dangoor
- re-rename module to migrate

5.13 0.4.0

- SA 0.4.0 compatibility thanks to Christian Simms
- all unit tests are working now (with sqlalchemy >= 0.3.10)

5.14 0.3

- SA 0.3.10 compatibility

5.15 0.2.3

- Removed lots of SA monkeypatching in Migrate's internals
- SA 0.3.3 compatibility
- Removed logsql (#75)
- Updated py.test version from 0.8 to 0.9; added a download link to setup.py
- Fixed incorrect "function not defined" error (#88)
- Fixed SQLite and .sql scripts (#87)

5.16 0.2.2

- Deprecated driver(engine) in favor of engine.name (#80)
- Deprecated logsql (#75)
- Comments in .sql scripts don't make things fail silently now (#74)

- Errors while downgrading (and probably other places) are shown on their own line
- Created mailing list and announcements list, updated documentation accordingly
- Automated tests now require py.test (#66)
- Documentation fix to .sql script commits (#72)
- Fixed a pretty major bug involving logengine, dealing with commits/tests (#64)
- Fixes to the online docs - default DB versioning table name (#68)
- Fixed the engine name in the scripts created by the command 'migrate script' (#69)
- Added Evan's email to the online docs

5.17 0.2.1

- Created this changelog
- Now requires (and is now compatible with) SA 0.3
- Commits across filesystems now allowed (shutil.move instead of os.rename) (#62)

CHAPTER

SIX

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

M

migrate, 1
migrate.changeset, 21
migrate.changeset.ansisql, 21
migrate.changeset.constraint, 22
migrate.changeset.databases, 25
migrate.changeset.databases.firebird, 26
migrate.changeset.databases.mysql, 26
migrate.changeset.databases.oracle, 26
migrate.changeset.databases.postgres, 26
migrate.changeset.databases.sqlite, 27
migrate.changeset.databases.visitor, 27
migrate.changeset.exceptions, 27
migrate.changeset.schema, 28
migrate.versioning, 31
migrate.versioning.api, 31
migrate.versioning.exceptions, 33
migrate.versioning.genmodel, 34
migrate.versioning.migrate_repository, 20
migrate.versioning.pathed, 34
migrate.versioning.repository, 35
migrate.versioning.schema, 36
migrate.versioning.schemadiff, 37
migrate.versioning.script.base, 37
migrate.versioning.script.py, 37
migrate.versioning.script.sql, 38
migrate.versioning.shell, 39
migrate.versioning.util, 39
migrate.versioning.version, 40

INDEX

A

add() (migrate.versioning.repository.Changeset method), 35
add_foreignkey() (migrate.changeset.databases.sqlite.SQLiteColumnGenerator method), 27
add_script() (migrate.versioning.version.Version method), 40
alter() (migrate.changeset.schema.ChangesetColumn method), 29
alter_column() (in module migrate.changeset.schema), 28
AlterTableVisitor (class in migrate.changeset.ansisql), 22
ANSIColumnDropper (class in migrate.changeset.ansisql), 21
ANSIColumnGenerator (class in migrate.changeset.ansisql), 21
ANSIConstraintCommon (class in migrate.changeset.ansisql), 21
ANSISchemaChanger (class in migrate.changeset.ansisql), 21
ApiError, 33
append() (migrate.changeset.ansisql.AlterTableVisitor method), 22
apply_diffs() (migrate.changeset.schema.ColumnDelta method), 30
are_column_types_eq() (migrate.changeset.schema.ColumnDelta method), 31
asbool() (in module migrate.versioning.util), 39
autoname() (migrate.changeset.constraint.ForeignKeyConstraint method), 23
autoname() (migrate.changeset.constraint.PrimaryKeyConstraint method), 24
autoname() (migrate.changeset.constraint.UniqueConstraint method), 25

B

BaseScript (class in migrate.versioning.script.base), 37

C

cascade_constraint() (migrate.changeset.databases.firebird.FBConstraintDropper method), 26
catch_known_errors() (in module migrate.versioning.util), 39
Changeset (class in migrate.versioning.repository), 35
changeset() (migrate.versioning.repository.Repository method), 35
changeset() (migrate.versioning.schema.ControlledSchema method), 36
ChangesetColumn (class in migrate.changeset.schema), 29
ChangesetDefaultClause (class in migrate.changeset.schema), 30
ChangesetIndex (class in migrate.changeset.schema), 30
ChangesetTable (class in migrate.changeset.schema), 28
CheckConstraint (class in migrate.changeset.constraint), 22
Collection (class in migrate.versioning.version), 40
ColumnDelta (class in migrate.changeset.schema), 30
compare_1_column() (migrate.changeset.schema.ColumnDelta method), 31
compare_2_columns() (migrate.changeset.schema.ColumnDelta method), 31
compare_model_to_db() (in module migrate.versioning.api), 33
compare_model_to_db() (migrate.versioning.schema.ControlledSchema class method), 36
compare_parameters() (migrate.changeset.schema.ColumnDelta method), 31
compareModelToDatabase() (migrate.versioning.schemadiff.SchemaDiff method), 37
ConstraintChangeset (class in migrate.changeset.constraint), 23
construct_engine() (in module migrate.versioning.util), 39
ControlledSchema (class in migrate.versioning.schema), 36
ControlledSchemaError, 33

copy_fixed() (migrate.changeset.schema.ChangesetColumn method), 29
create() (in module migrate.versioning.api), 32
create() (migrate.changeset.constraint.CheckConstraint method), 22
create() (migrate.changeset.constraint.ConstraintChangeset method), 23
create() (migrate.changeset.constraint.ForeignKeyConstraint method), 24
create() (migrate.changeset.constraint.PrimaryKeyConstraint method), 24
create() (migrate.changeset.constraint.UniqueConstraint method), 25
create() (migrate.changeset.schema.ChangesetColumn method), 29
create() (migrate.versioning.repository.Repository class method), 35
create() (migrate.versioning.schema.ControlledSchema class method), 36
create() (migrate.versioning.script.py.PythonScript class method), 37
create() (migrate.versioning.script.sql.SqlScript class method), 38
create_column() (in module migrate.changeset.schema), 28
create_column() (migrate.changeset.schema.ChangesetTable method), 28
create_manage_file() (migrate.versioning.repository.Repository class method), 35
create_model() (in module migrate.versioning.api), 32
create_model() (migrate.versioning.schema.ControlledSchema class method), 36
create_new_python_version() (migrate.versioning.version.Collection method), 40
create_new_sql_version() (migrate.versioning.version.Collection method), 40
create_script() (migrate.versioning.repository.Repository method), 35
create_script_sql() (migrate.versioning.repository.Repository method), 35

D

DatabaseAlreadyControlledError, 33
DatabaseNotControlledError, 33
db_version() (in module migrate.versioning.api), 31
deregister() (migrate.changeset.schema.ChangesetTable method), 29
downgrade() (in module migrate.versioning.api), 33
drop() (migrate.changeset.constraint.CheckConstraint method), 23
drop() (migrate.changeset.constraint.ConstraintChangeset method), 23
drop() (migrate.changeset.constraint.ForeignKeyConstraint method), 24
drop() (migrate.changeset.constraint.PrimaryKeyConstraint method), 24
drop() (migrate.changeset.constraint.UniqueConstraint method), 25
drop() (migrate.changeset.schema.ChangesetColumn method), 30
drop() (migrate.versioning.schema.ControlledSchema method), 36
drop_column() (in module migrate.changeset.schema), 28
drop_column() (migrate.changeset.schema.ChangesetTable method), 29
drop_version_control() (in module migrate.versioning.api), 31

E

Error, 27, 34
execute() (migrate.changeset.ansisql.AlterTableVisitor method), 22
Extensions (class in migrate.versioning.version), 40

F

FBColumnDropper (class in migrate.changeset.databases.firebird), 26
FBColumnGenerator (class in migrate.changeset.databases.firebird), 26
FBConstraintDropper (class in migrate.changeset.databases.firebird), 26
FBConstraintGenerator (class in migrate.changeset.databases.firebird), 26
FBSchemaChanger (class in migrate.changeset.databases.firebird), 26
ForeignKeyConstraint (class in migrate.changeset.constraint), 23

G

get_children() (migrate.changeset.constraint.CheckConstraint method), 23
get_children() (migrate.changeset.constraint.ForeignKeyConstraint method), 24
get_children() (migrate.changeset.constraint.PrimaryKeyConstraint method), 25
get_children() (migrate.changeset.constraint.UniqueConstraint method), 25
get_constraint_name() (migrate.changeset.ansisql.ANSIConstraintCommon method), 21
get_dialect_visitor() (in module migrate.changeset.databases.visitor), 27
get_engine_visitor() (in module migrate.changeset.databases.visitor), 27

getDiffOfModelAgainstDatabase() (in module migrate.versioning.schemadiff), 37
getDiffOfModelAgainstModel() (in module migrate.versioning.schemadiff), 37
guess_obj_type() (in module migrate.versioning.util), 39

H

help() (in module migrate.versioning.api), 31

I

id (migrate.versioning.repository.Repository attribute), 36
InvalidConstraintError, 27
InvalidRepositoryError, 34
InvalidScriptError, 34
InvalidVersionError, 34

K

keys() (migrate.versioning.repository.Changeset method), 35
KnownError, 34

L

latest (migrate.versioning.repository.Repository attribute), 36
latest (migrate.versioning.version.Collection attribute), 40
load() (migrate.versioning.schema.ControlledSchema method), 36
load_model() (in module migrate.versioning.util), 39

M

main() (in module migrate.versioning.shell), 39
make_update_script_for_model() (in module migrate.versioning.api), 33
make_update_script_for_model() (migrate.versioning.script.py.PythonScript class method), 37
manage() (in module migrate.versioning.api), 32
Memoize (class in migrate.versioning.util), 39
migrate (module), 1
migrate.changeset (module), 21
migrate.changeset.ansisql (module), 21
migrate.changeset.constraint (module), 22
migrate.changeset.databases (module), 25
migrate.changeset.databases.firebird (module), 26
migrate.changeset.databases.mysql (module), 26
migrate.changeset.databases.oracle (module), 26
migrate.changeset.databases.postgres (module), 26
migrate.changeset.databases.sqlite (module), 27
migrate.changeset.databases.visitor (module), 27
migrate.changeset.exceptions (module), 27
migrate.changeset.schema (module), 28
migrate.versioning (module), 31
migrate.versioning.api (module), 31
migrate.versioning.exceptions (module), 33
migrate.versioning.genmodel (module), 34
migrate.versioning.migrate_repository (module), 20
migrate.versioning.pathed (module), 34
migrate.versioning.repository (module), 35
migrate.versioning.schema (module), 36
migrate.versioning.schemadiff (module), 37
migrate.versioning.script.base (module), 37
migrate.versioning.script.py (module), 37
migrate.versioning.script.sql (module), 38
migrate.versioning.shell (module), 39
migrate.versioning.util (module), 39
migrate.versioning.version (module), 40
MigrateDeprecationWarning, 27
module (migrate.versioning.script.py.PythonScript attribute), 38

N

NoSuchTableError, 34
NotSupportedError, 27

P

Pathed (class in migrate.versioning.pathed), 34
PathError, 34
PathFoundError, 34
PathNotFoundError, 34
PGColumnDropper (class in migrate.changeset.databases.postgres), 26
PGColumnGenerator (class in migrate.changeset.databases.postgres), 26
PGConstraintDropper (class in migrate.changeset.databases.postgres), 26
PGConstraintGenerator (class in migrate.changeset.databases.postgres), 26
PGSchemaChanger (class in migrate.changeset.databases.postgres), 26
prepare_config() (migrate.versioning.repository.Repository class method), 35
preview_sql() (migrate.versioning.script.py.PythonScript method), 38
PrimaryKeyConstraint (class in migrate.changeset.constraint), 24
process_column() (migrate.changeset.schema.ColumnDelta method), 31
PythonScript (class in migrate.versioning.script.py), 37

R

rename() (migrate.changeset.schema.ChangesetIndex method), 30
rename() (migrate.changeset.schema.ChangesetTable method), 29
rename_index() (in module migrate.changeset.schema), 28

rename_table() (in module migrate.changeset.schema), 28
Repository (class in migrate.versioning.repository), 35
repository migration, 20
RepositoryError, 34
require_found() (migrate.versioning.pathed.Pathed class method), 34
require_found() (migrate.versioning.script.py.PythonScript class method), 38
require_found() (migrate.versioning.script.sql.SqlScript class method), 38
require_notfound() (migrate.versioning.pathed.Pathed class method), 34
require_notfound() (migrate.versioning.script.py.PythonScript class method), 38
require_notfound() (migrate.versioning.script.sql.SqlScript class method), 38
run() (migrate.versioning.repository.Changeset method), 35
run() (migrate.versioning.script.base.BaseScript method), 37
run() (migrate.versioning.script.py.PythonScript method), 38
run() (migrate.versioning.script.sql.SqlScript method), 38
run_single_visitor() (in module migrate.changeset.databases.visitor), 27

S

SchemaDiff (class in migrate.versioning.schemadiff), 37
script() (in module migrate.versioning.api), 32
script() (migrate.versioning.version.Version method), 40
script_sql() (in module migrate.versioning.api), 33
ScriptError, 34
source() (in module migrate.versioning.api), 32
source() (migrate.versioning.script.base.BaseScript method), 37
source() (migrate.versioning.script.py.PythonScript method), 38
source() (migrate.versioning.script.sql.SqlScript method), 39
SQLiteColumnDropper (class in migrate.changeset.databases.sqlite), 27
SQLiteColumnGenerator (class in migrate.changeset.databases.sqlite), 27
SQLiteSchemaChanger (class in migrate.changeset.databases.sqlite), 27
SqlScript (class in migrate.versioning.script.sql), 38
start_alter_column() (migrate.changeset.ansisql.ANSISchemaChanger method), 22
start_alter_table() (migrate.changeset.ansisql.AlterTableVisitor method), 22
str_to_filename() (in module migrate.versioning.version), 40

T

test() (in module migrate.versioning.api), 32

U

UniqueConstraint (class in migrate.changeset.constraint), 25
update_db_from_model() (in module migrate.versioning.api), 32
update_db_from_model() (migrate.versioning.schema.ControlledSchema method), 36
update_repository_table() (migrate.versioning.schema.ControlledSchema method), 36
upgrade() (in module migrate.versioning.api), 31
upgrade() (migrate.versioning.schema.ControlledSchema method), 36
UsageError, 34

V

verify() (migrate.versioning.repository.Repository class method), 36
verify() (migrate.versioning.script.base.BaseScript class method), 37
verify() (migrate.versioning.script.py.PythonScript class method), 38
verify() (migrate.versioning.script.sql.SqlScript class method), 39
verify_module() (migrate.versioning.script.py.PythonScript class method), 38
VerNum (class in migrate.versioning.version), 40
Version (class in migrate.versioning.version), 40
version() (in module migrate.versioning.api), 32
version() (migrate.versioning.repository.Repository method), 36
version() (migrate.versioning.version.Collection method), 40
version_control() (in module migrate.versioning.api), 33
version_table (migrate.versioning.repository.Repository attribute), 36
visit_column() (migrate.changeset.ansisql.ANSIColumnDropper method), 21
visit_column() (migrate.changeset.ansisql.ANSIColumnGenerator method), 21
visit_column() (migrate.changeset.ansisql.ANSISchemaChanger method), 22
visit_column() (migrate.changeset.databases.firebird.FBColumnDropper method), 26
visit_index() (migrate.changeset.ansisql.ANSISchemaChanger method), 22

visit_index() (migrate.changeset.databases.sqlite.SQLiteSchemaChanger method), 27
visit_table() (migrate.changeset.ansisql.ANSISchemaChanger method), 22
visit_table() (migrate.changeset.databases.firebird.FBSchemaChanger method), 26

W

with_engine() (in module migrate.versioning.util), 40
WrongRepositoryError, 34