SQLAlchemy-Utils Documentation

Release 0.32.18

Konsta Vesterinen

Oct 06, 2017

Contents

SQLAlchemy-Utils provides custom data types and various utility functions for SQLAlchemy.

CHAPTER 1

Installation

This part of the documentation covers the installation of SQLAlchemy-Utils.

Supported platforms

SQLAlchemy-Utils has been tested against the following Python platforms.

- cPython 2.6 (unsupported since 0.32)
- cPython 2.7
- cPython 3.3
- cPython 3.4
- cPython 3.5

Installing an official release

You can install the most recent official SQLAlchemy-Utils version using pip:

```
pip install sqlalchemy-utils
```

Installing the development version

To install the latest version of SQLAlchemy-Utils, you need first obtain a copy of the source. You can do that by cloning the git repository:

```
git clone git://github.com/kvesteri/sqlalchemy-utils.git
```

Then you can install the source distribution using the `setup.py` script:

```
cd sqlalchemy-utils
python setup.py install
```

Checking the installation

To check that SQLAlchemy-Utils has been properly installed, type `python` from your shell. Then at the Python prompt, try to import SQLAlchemy-Utils, and check the installed version:

```
>>> import sqlalchemy_utils
>>> sqlalchemy_utils.__version__
0.32.18
```

CHAPTER 2

Listeners

Automatic data coercion

sqlalchemy_utils.listeners.**force_auto_coercion**(*mapper=None*)

Function that assigns automatic data type coercion for all classes which are of type of given mapper. The coercion is applied to all coercion capable properties. By default coercion is applied to all SQLAlchemy mappers.

Before initializing your models you need to call force_auto_coercion.

```
from sqlalchemy_utils import force_auto_coercion


force_auto_coercion()
```

Then define your models the usual way:

```
class Document(Base):
    __tablename__ = 'document'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(50))
    background_color = sa.Column(ColorType)
```

Now scalar values for coercion capable data types will convert to appropriate value objects:

```
document = Document()
document.background_color = 'F5F5F5'
document.background_color  # Color object
session.commit()
```

A useful side-effect of this is that additional validation of data will be done on the moment it is being assigned to model objects. For example without auto coerction set, an invalid `sqlalchemy_utils.types.IPAddressType` (eg. `10.0.0 255.255`) would get through without an exception being raised. The database wouldn't notice this (as most databases don't have a native type for an IP address, so they're usually just stored as a string), and the `ipaddress/ipaddr` package uses a string field as well.

Parameters **mapper** – The mapper which the automatic data type coercion should be applied to

Instant defaults

sqlalchemy_utils.listeners.**force_instant_defaults**(*mapper=None*)

Function that assigns object column defaults on object initialization time. By default calling this function applies instant defaults to all your models.

Setting up instant defaults:

```
from sqlalchemy_utils import force_instant_defaults


force_instant_defaults()
```

Example usage:

```
class Document(Base):
    __tablename__ = 'document'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(50))
    created_at = sa.Column(sa.DateTime, default=datetime.now)


document = Document()
document.created_at  # datetime object
```

Parameters **mapper** – The mapper which the automatic instant defaults forcing should be applied to

Many-to-many orphan deletion

sqlalchemy_utils.listeners.**auto_delete_orphans**(*attr*)

Delete orphans for given SQLAlchemy model attribute. This function can be used for deleting many-to-many associated orphans easily. For more information see https://bitbucket.org/zzzeek/sqlalchemy/wiki/UsageRecipes/ManyToManyOrphan.

Consider the following model definition:

```
from sqlalchemy.ext.associationproxy import association_proxy
from sqlalchemy import *
from sqlalchemy.orm import *
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import event


Base = declarative_base()

tagging = Table(
    'tagging',
    Base.metadata,
    Column(
        'tag_id',
        Integer,
```

```
        ForeignKey('tag.id', ondelete='CASCADE'),
        primary_key=True
    ),
    Column(
        'entry_id',
        Integer,
        ForeignKey('entry.id', ondelete='CASCADE'),
        primary_key=True
    )
)

class Tag(Base):
    __tablename__ = 'tag'
    id = Column(Integer, primary_key=True)
    name = Column(String(100), unique=True, nullable=False)

    def __init__(self, name=None):
        self.name = name

class Entry(Base):
    __tablename__ = 'entry'

    id = Column(Integer, primary_key=True)

    tags = relationship(
        'Tag',
        secondary=tagging,
        backref='entries'
    )
```

Now lets say we want to delete the tags if all their parents get deleted (all Entry objects get deleted). This can be achieved as follows:

```
from sqlalchemy_utils import auto_delete_orphans


auto_delete_orphans(Entry.tags)
```

After we've set up this listener we can see it in action.

```
e = create_engine('sqlite://')

Base.metadata.create_all(e)

s = Session(e)

r1 = Entry()
r2 = Entry()
r3 = Entry()
t1, t2, t3, t4 = Tag('t1'), Tag('t2'), Tag('t3'), Tag('t4')

r1.tags.extend([t1, t2])
r2.tags.extend([t2, t3])
r3.tags.extend([t4])
s.add_all([r1, r2, r3])

assert s.query(Tag).count() == 4
```

```
r2.tags.remove(t2)

assert s.query(Tag).count() == 4

r1.tags.remove(t2)

assert s.query(Tag).count() == 3

r1.tags.remove(t1)

assert s.query(Tag).count() == 2
```

Parameters **attr** – Association relationship attribute to auto delete orphans from

CHAPTER 3

Data types

SQLAlchemy-Utils provides various new data types for SQLAlchemy. In order to gain full advantage of these datatypes you should use automatic data coercion. See `force_auto_coercion()` for how to set up this feature.

ArrowType

class `sqlalchemy_utils.types.arrow.`**`ArrowType`**(**args*, ***kwargs*)

ArrowType provides way of saving Arrow objects into database. It automatically changes Arrow objects to datetime objects on the way in and datetime objects back to Arrow objects on the way out (when querying database). ArrowType needs Arrow library installed.

```
from datetime import datetime
from sqlalchemy_utils import ArrowType
import arrow


class Article(Base):
    __tablename__ = 'article'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    created_at = sa.Column(ArrowType)



article = Article(created_at=arrow.utcnow())
```

As you may expect all the arrow goodies come available:

```
article.created_at = article.created_at.replace(hours=-1)

article.created_at.humanize()
# 'an hour ago'
```

ChoiceType

class sqlalchemy_utils.types.choice.**ChoiceType**(*choices*, *impl=None*)

ChoiceType offers way of having fixed set of choices for given column. It could work with a list of tuple (a collection of key-value pairs), or integrate with `enum` in the standard library of Python 3.4+ (the enum34 backported package on PyPI is compatible too for `< 3.4`).

Columns with ChoiceTypes are automatically coerced to Choice objects while a list of tuple been passed to the constructor. If a subclass of `enum.Enum` is passed, columns will be coerced to `enum.Enum` objects instead.

```
class User(Base):
    TYPES = [
        (u'admin', u'Admin'),
        (u'regular-user', u'Regular user')
    ]

    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    type = sa.Column(ChoiceType(TYPES))


user = User(type=u'admin')
user.type  # Choice(type='admin', value=u'Admin')
```

Or:

```
import enum


class UserType(enum.Enum):
    admin = 1
    regular = 2


class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    type = sa.Column(ChoiceType(UserType, impl=sa.Integer()))


user = User(type=1)
user.type  # <UserType.admin: 1>
```

ChoiceType is very useful when the rendered values change based on user's locale:

```
from babel import lazy_gettext as _


class User(Base):
    TYPES = [
        (u'admin', _(u'Admin')),
        (u'regular-user', _(u'Regular user'))
    ]

    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
```

```
    name = sa.Column(sa.Unicode(255))
    type = sa.Column(ChoiceType(TYPES))


user = User(type=u'admin')
user.type  # Choice(type='admin', value=u'Admin')

print user.type  # u'Admin'
```

Or:

```
from enum import Enum
from babel import lazy_gettext as _


class UserType(Enum):
    admin = 1
    regular = 2


UserType.admin.label = _(u'Admin')
UserType.regular.label = _(u'Regular user')


class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    type = sa.Column(ChoiceType(UserType, impl=sa.Integer()))


user = User(type=UserType.admin)
user.type  # <UserType.admin: 1>

print user.type.label  # u'Admin'
```

ColorType

class sqlalchemy_utils.types.color.**ColorType**(*max_length=20*, **args*, ***kwargs*)

ColorType provides a way for saving Color (from colour package) objects into database. ColorType saves Color objects as strings on the way in and converts them back to objects when querying the database.

```
from colour import Color
from sqlalchemy_utils import ColorType


class Document(Base):
    __tablename__ = 'document'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(50))
    background_color = sa.Column(ColorType)


document = Document()
document.background_color = Color('#F5F5F5')
```

```
session.commit()
```

Querying the database returns Color objects:

```
document = session.query(Document).first()

document.background_color.hex
# '#f5f5f5'
```

CompositeType

CompositeType provides means to interact with PostgreSQL composite types. Currently this type features:

- Easy attribute access to composite type fields
- Supports SQLAlchemy TypeDecorator types
- Ability to include composite types as part of PostgreSQL arrays
- Type creation and dropping

Installation

CompositeType automatically attaches *before_create* and *after_drop* DDL listeners. These listeners create and drop the composite type in the database. This means it works out of the box in your test environment where you create the tables on each test run.

When you already have your database set up you should call `register_composites()` after you've set up all models.

```
register_composites(conn)
```

Usage

```
from collections import OrderedDict

import sqlalchemy as sa
from sqlalchemy_utils import CompositeType, CurrencyType


class Account(Base):
    __tablename__ = 'account'
    id = sa.Column(sa.Integer, primary_key=True)
    balance = sa.Column(
        CompositeType(
            'money_type',
            [
                sa.Column('currency', CurrencyType),
                sa.Column('amount', sa.Integer)
            ]
        )
    )
```

Accessing fields

CompositeType provides attribute access to underlying fields. In the following example we find all accounts with balance amount more than 5000.

```
session.query(Account).filter(Account.balance.amount > 5000)
```

Arrays of composites

```
from sqlalchemy_utils import CompositeArray


class Account(Base):
    __tablename__ = 'account'
    id = sa.Column(sa.Integer, primary_key=True)
    balances = sa.Column(
        CompositeArray(
            CompositeType(
                'money_type',
                [
                    sa.Column('currency', CurrencyType),
                    sa.Column('amount', sa.Integer)
                ]
            )
        )
    )
```

Related links:

http://schinckel.net/2014/09/24/using-postgres-composite-types-in-django/

class sqlalchemy_utils.types.pg_composite.**CompositeType**(*name*, *columns*)
Represents a PostgreSQL composite type.

Parameters

- **name** – Name of the composite type.
- **columns** – List of columns that this composite type consists of

CountryType

class sqlalchemy_utils.types.country.**CountryType**(**args*, ***kwargs*)
Changes *Country* objects to a string representation on the way in and changes them back to :class:‘.Country objects on the way out.

In order to use CountryType you need to install Babel first.

```
from sqlalchemy_utils import CountryType, Country


class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(255))
    country = sa.Column(CountryType)
```

```
user = User()
user.country = Country('FI')
session.add(user)
session.commit()

user.country  # Country('FI')
user.country.name  # Finland

print user.country  # Finland
```

CountryType is scalar coercible:

```
user.country = 'US'
user.country  # Country('US')
```

class sqlalchemy_utils.primitives.country.**Country**(*code_or_country*)

Country class wraps a 2 to 3 letter country code. It provides various convenience properties and methods.

```
from babel import Locale
from sqlalchemy_utils import Country, i18n


# First lets add a locale getter for testing purposes
i18n.get_locale = lambda: Locale('en')


Country('FI').name  # Finland
Country('FI').code  # FI

Country(Country('FI')).code  # 'FI'
```

Country always validates the given code.

```
Country(None)  # raises TypeError

Country('UnknownCode')  # raises ValueError
```

Country supports equality operators.

```
Country('FI') == Country('FI')
Country('FI') != Country('US')
```

Country objects are hashable.

```
assert hash(Country('FI')) == hash('FI')
```

CurrencyType

class sqlalchemy_utils.types.currency.**CurrencyType**(**args*, ***kwargs*)

Changes *Currency* objects to a string representation on the way in and changes them back to *Currency* objects on the way out.

In order to use CurrencyType you need to install Babel first.

```
from sqlalchemy_utils import CurrencyType, Currency


class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(255))
    currency = sa.Column(CurrencyType)


user = User()
user.currency = Currency('USD')
session.add(user)
session.commit()

user.currency  # Currency('USD')
user.currency.name  # US Dollar

str(user.currency)  # US Dollar
user.currency.symbol  # $
```

CurrencyType is scalar coercible:

```
user.currency = 'US'
user.currency  # Currency('US')
```

class sqlalchemy_utils.primitives.currency.**Currency**(*code*)

Currency class wraps a 3-letter currency code. It provides various convenience properties and methods.

```
from babel import Locale
from sqlalchemy_utils import Currency, i18n


# First lets add a locale getter for testing purposes
i18n.get_locale = lambda: Locale('en')


Currency('USD').name  # US Dollar
Currency('USD').symbol  # $

Currency(Currency('USD')).code  # 'USD'
```

Currency always validates the given code.

```
Currency(None)  # raises TypeError

Currency('UnknownCode')  # raises ValueError
```

Currency supports equality operators.

```
Currency('USD') == Currency('USD')
Currency('USD') != Currency('EUR')
```

Currencies are hashable.

```
len(set([Currency('USD'), Currency('USD')]))  # 1
```

EmailType

class sqlalchemy_utils.types.email.**EmailType**(*length=255*, **args*, ***kwargs*)

Provides a way for storing emails in a lower case.

Example:

```
from sqlalchemy_utils import EmailType


class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    email = sa.Column(EmailType)


user = User()
user.email = 'John.Smith@foo.com'
user.name = 'John Smith'
session.add(user)
session.commit()
# Notice - email in filter() is lowercase.
user = (session.query(User)
               .filter(User.email == 'john.smith@foo.com')
               .one())
assert user.name == 'John Smith'
```

EncryptedType

class sqlalchemy_utils.types.encrypted.**EncryptedType**(*type_in=None*, *key=None*, *engine=None*, ***kwargs*)

EncryptedType provides a way to encrypt and decrypt values, to and from databases, that their type is a basic SQLAlchemy type. For example Unicode, String or even Boolean. On the way in, the value is encrypted and on the way out the stored value is decrypted.

EncryptedType needs Cryptography library in order to work. A simple example is given below.

```
import sqlalchemy as sa
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy_utils import EncryptedType


secret_key = 'secretkey1234'
# setup
engine = create_engine('sqlite:///:memory:')
connection = engine.connect()
Base = declarative_base()

class User(Base):
    __tablename__ = "user"
    id = sa.Column(sa.Integer, primary_key=True)
    username = sa.Column(EncryptedType(sa.Unicode, secret_key))
    access_token = sa.Column(EncryptedType(sa.String, secret_key))
```

```
    is_active = sa.Column(EncryptedType(sa.Boolean, secret_key))
    number_of_accounts = sa.Column(EncryptedType(sa.Integer,
                                                 secret_key))

sa.orm.configure_mappers()
Base.metadata.create_all(connection)

# create a configured "Session" class
Session = sessionmaker(bind=connection)

# create a Session
session = Session()

# example
user_name = u'secret_user'
test_token = 'atesttoken'
active = True
num_of_accounts = 2

user = User(username=user_name, access_token=test_token,
            is_active=active, accounts_num=accounts)
session.add(user)
session.commit()

print('id: {}'.format(user.id))
print('username: {}'.format(user.username))
print('token: {}'.format(user.access_token))
print('active: {}'.format(user.is_active))
print('accounts: {}'.format(user.accounts_num))

# teardown
session.close_all()
Base.metadata.drop_all(connection)
connection.close()
engine.dispose()
```

The key parameter accepts a callable to allow for the key to change per-row instead of be fixed for the whole table.

::

def get_key(): return 'dynamic-key'

class User(Base): __tablename__ = 'user' id = sa.Column(sa.Integer, primary_key=True) username = sa.Column(EncryptedType(

sa.Unicode, get_key))

JSONType

class sqlalchemy_utils.types.json.**JSONType**(**args*, ***kwargs*)

JSONType offers way of saving JSON data structures to database. On PostgreSQL the underlying implementation of this data type is 'json' while on other databases its simply 'text'.

```
from sqlalchemy_utils import JSONType
```

```
class Product(Base):
    __tablename__ = 'product'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(50))
    details = sa.Column(JSONType)


product = Product()
product.details = {
    'color': 'red',
    'type': 'car',
    'max-speed': '400 mph'
}
session.commit()
```

LocaleType

class sqlalchemy_utils.types.locale.**LocaleType**

LocaleType saves Babel Locale objects into database. The Locale objects are converted to string on the way in and back to object on the way out.

In order to use LocaleType you need to install Babel first.

```
from sqlalchemy_utils import LocaleType
from babel import Locale


class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(50))
    locale = sa.Column(LocaleType)


user = User()
user.locale = Locale('en_US')
session.add(user)
session.commit()
```

Like many other types this type also supports scalar coercion:

```
user.locale = 'de_DE'
user.locale  # Locale('de', territory='DE')
```

LtreeType

class sqlalchemy_utils.types.ltree.**LtreeType**

Postgresql LtreeType type.

The LtreeType datatype can be used for representing labels of data stored in hierarchial tree-like structure. For more detailed information please refer to http://www.postgresql.org/docs/current/static/ltree.html

```
from sqlalchemy_utils import LtreeType


class DocumentSection(Base):
    __tablename__ = 'document_section'
    id = sa.Column(sa.Integer, autoincrement=True)
    path = sa.Column(LtreeType)


section = DocumentSection(name='Countries.Finland')
session.add(section)
session.commit()

section.path  # Ltree('Countries.Finland')
```

Note: Using *LtreeType*, LQUERY and LTXTQUERY types may require installation of Postgresql ltree extension on the server side. Please visit http://www.postgres.org for details.

class sqlalchemy_utils.primitives.ltree.**Ltree**(*path_or_ltree*)

Ltree class wraps a valid string label path. It provides various convenience properties and methods.

```
from sqlalchemy_utils import Ltree

Ltree('1.2.3').path  # '1.2.3'
```

Ltree always validates the given path.

```
Ltree(None)  # raises TypeError

Ltree('..')  # raises ValueError
```

Validator is also available as class method.

```
Ltree.validate('1.2.3')
Ltree.validate(None)  # raises ValueError
```

Ltree supports equality operators.

```
Ltree('Countries.Finland') == Ltree('Countries.Finland')
Ltree('Countries.Germany') != Ltree('Countries.Finland')
```

Ltree objects are hashable.

```
assert hash(Ltree('Finland')) == hash('Finland')
```

Ltree objects have length.

```
assert len(Ltree('1.2'))  2
assert len(Ltree('some.one.some.where'))  # 4
```

You can easily find subpath indexes.

```
assert Ltree('1.2.3').index('2.3') == 1
assert Ltree('1.2.3.4.5').index('3.4') == 2
```

Ltree objects can be sliced.

```
assert Ltree('1.2.3')[0:2] == Ltree('1.2')
assert Ltree('1.2.3')[1:] == Ltree('2.3')
```

Finding longest common ancestor.

```
assert Ltree('1.2.3.4.5').lca('1.2.3', '1.2.3.4', '1.2.3') == '1.2'
assert Ltree('1.2.3.4.5').lca('1.2', '1.2.3') == '1'
```

Ltree objects can be concatenated.

```
assert Ltree('1.2') + Ltree('1.2') == Ltree('1.2.1.2')
```

IPAddressType

class sqlalchemy_utils.types.ip_address.**IPAddressType**(*max_length=50*, **args*, ***kwargs*)

Changes IPAddress objects to a string representation on the way in and changes them back to IPAddress objects on the way out.

IPAddressType uses ipaddress package on Python >= 3 and ipaddr package on Python 2. In order to use IPAddressType with python you need to install ipaddr first.

```
from sqlalchemy_utils import IPAddressType


class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(255))
    ip_address = sa.Column(IPAddressType)


user = User()
user.ip_address = '123.123.123.123'
session.add(user)
session.commit()

user.ip_address  # IPAddress object
```

PasswordType

class sqlalchemy_utils.types.password.**PasswordType**(*max_length=None*, ***kwargs*)

PasswordType hashes passwords as they come into the database and allows verifying them using a Pythonic interface. This Pythonic interface relies on setting up automatic data type coercion using the *force_auto_coercion()* function.

All keyword arguments (aside from max_length) are forwarded to the construction of a *passlib.context.LazyCryptContext* object, which also supports deferred configuration via the *onload* callback.

The following usage will create a password column that will automatically hash new passwords as *pbkdf2_sha512* but still compare passwords against pre-existing *md5_crypt* hashes. As passwords are compared; the password hash in the database will be updated to be *pbkdf2_sha512*.

```
class Model(Base):
    password = sa.Column(PasswordType(
        schemes=[
            'pbkdf2_sha512',
            'md5_crypt'
        ],

        deprecated=['md5_crypt']
    ))
```

Verifying password is as easy as:

```
target = Model()
target.password = 'b'
# '$5$rounds=80000$H............'

target.password == 'b'
# True
```

Lazy configuration of the type with Flask config:

```
import flask
from sqlalchemy_utils import PasswordType, force_auto_coercion

force_auto_coercion()

class User(db.Model):
    __tablename__ = 'user'

    password = db.Column(
        PasswordType(
            # The returned dictionary is forwarded to the CryptContext
            onload=lambda **kwargs: dict(
                schemes=flask.current_app.config['PASSWORD_SCHEMES'],
                **kwargs
            ),
        ),
        unique=False,
        nullable=False,
    )
```

PhoneNumberType

class sqlalchemy_utils.types.phone_number.**PhoneNumber**(*raw_number*, *region=None*)

Extends a PhoneNumber class from Python phonenumbers library. Adds different phone number formats to attributes, so they can be easily used in templates. Phone number validation method is also implemented.

Takes the raw phone number and country code as params and parses them into a PhoneNumber object.

```
from sqlalchemy_utils import PhoneNumber


class User(self.Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, autoincrement=True, primary_key=True)
```

```
    name = sa.Column(sa.Unicode(255))
    _phone_number = sa.Column(sa.Unicode(20))
    country_code = sa.Column(sa.Unicode(8))

    phonenumber = sa.orm.composite(
        PhoneNumber,
        _phone_number,
        country_code
    )


user = User(phone_number=PhoneNumber('0401234567', 'FI'))

user.phone_number.e164  # u'+358401234567'
user.phone_number.international  # u'+358 40 1234567'
user.phone_number.national  # u'040 1234567'
user.country_code  # 'FI'
```

Parameters

- **raw_number** – String representation of the phone number.
- **region** – Region of the phone number.

class sqlalchemy_utils.types.phone_number.**PhoneNumberType**(*region='US'*, *max_length=20*, **args*, ***kwargs*)

Changes PhoneNumber objects to a string representation on the way in and changes them back to PhoneNumber objects on the way out. If E164 is used as storing format, no country code is needed for parsing the database value to PhoneNumber object.

```
class User(self.Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, autoincrement=True, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    phone_number = sa.Column(PhoneNumberType())


user = User(phone_number='+358401234567')

user.phone_number.e164  # u'+358401234567'
user.phone_number.international  # u'+358 40 1234567'
user.phone_number.national  # u'040 1234567'
```

ScalarListType

class sqlalchemy_utils.types.scalar_list.**ScalarListType**(*coerce_func=<type 'unicode'>*, *separator=u','*)

ScalarListType type provides convenient way for saving multiple scalar values in one column. ScalarListType works like list on python side and saves the result as comma-separated list in the database (custom separators can also be used).

Example

```
from sqlalchemy_utils import ScalarListType


class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, autoincrement=True)
    hobbies = sa.Column(ScalarListType())


user = User()
user.hobbies = [u'football', u'ice_hockey']
session.commit()
```

You can easily set up integer lists too:

```
from sqlalchemy_utils import ScalarListType


class Player(Base):
    __tablename__ = 'player'
    id = sa.Column(sa.Integer, autoincrement=True)
    points = sa.Column(ScalarListType(int))


player = Player()
player.points = [11, 12, 8, 80]
session.commit()
```

TimezoneType

class sqlalchemy_utils.types.timezone.**TimezoneType**(*backend='dateutil'*)

TimezoneType provides a way for saving timezones (from either the pytz or the dateutil package) objects into database. TimezoneType saves timezone objects as strings on the way in and converts them back to objects when querying the database.

```
from sqlalchemy_utils import TimezoneType

class User(Base):
    __tablename__ = 'user'

    # Pass backend='pytz' to change it to use pytz (dateutil by
    # default)
    timezone = sa.Column(TimezoneType(backend='pytz'))
```

TSVectorType

class sqlalchemy_utils.types.ts_vector.**TSVectorType**(**args*, ***kwargs*)

Note: This type is PostgreSQL specific and is not supported by other dialects.

Provides additional functionality for SQLAlchemy PostgreSQL dialect's TSVECTOR type. This additional functionality includes:

- Vector concatenation
- regconfig constructor parameter which is applied to match function if no postgresql_regconfig parameter is given
- Provides extensible base for extensions such as SQLAlchemy-Searchable

```
from sqlalchemy_utils import TSVectorType


class Article(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String(100))
    search_vector = sa.Column(TSVectorType)


# Find all articles whose name matches 'finland'
session.query(Article).filter(Article.search_vector.match('finland'))
```

TSVectorType also supports vector concatenation.

```
class Article(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String(100))
    name_vector = sa.Column(TSVectorType)
    content = sa.Column(sa.String)
    content_vector = sa.Column(TSVectorType)

# Find all articles whose name or content matches 'finland'
session.query(Article).filter(
    (Article.name_vector | Article.content_vector).match('finland')
)
```

You can configure TSVectorType to use a specific regconfig.

```
class Article(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String(100))
    search_vector = sa.Column(
        TSVectorType(regconfig='pg_catalog.simple')
    )
```

Now expression such as:

```
Article.search_vector.match('finland')
```

Would be equivalent to SQL:

```
search_vector @@ to_tsquery('pg_catalog.simgle', 'finland')
```

URLType

class sqlalchemy_utils.types.url.**URLType**(**args*, ***kwargs*)

URLType stores furl objects into database.

```
from sqlalchemy_utils import URLType
from furl import furl


class User(Base):
    __tablename__ = 'user'

    id = sa.Column(sa.Integer, primary_key=True)
    website = sa.Column(URLType)


user = User(website=u'www.example.com')

# website is coerced to furl object, hence all nice furl operations
# come available
user.website.args['some_argument'] = '12'

print user.website
# www.example.com?some_argument=12
```

UUIDType

class sqlalchemy_utils.types.uuid.**UUIDType**(*binary=True*, *native=True*)

Stores a UUID in the database natively when it can and falls back to a BINARY(16) or a CHAR(32) when it can't.

```
from sqlalchemy_utils import UUIDType
import uuid

class User(Base):
    __tablename__ = 'user'

    # Pass `binary=False` to fallback to CHAR instead of BINARY
    id = sa.Column(UUIDType(binary=False), primary_key=True)
```

WeekDaysType

class sqlalchemy_utils.types.weekdays.**WeekDaysType**(**args*, ***kwargs*)

WeekDaysType offers way of saving WeekDays objects into database. The WeekDays objects are converted to bit strings on the way in and back to WeekDays objects on the way out.

In order to use WeekDaysType you need to install Babel first.

```
from sqlalchemy_utils import WeekDaysType, WeekDays
from babel import Locale
```

```
class Schedule(Base):
    __tablename__ = 'schedule'
    id = sa.Column(sa.Integer, autoincrement=True)
    working_days = sa.Column(WeekDaysType)


schedule = Schedule()
schedule.working_days = WeekDays('0001111')
session.add(schedule)
session.commit()

print schedule.working_days  # Thursday, Friday, Saturday, Sunday
```

WeekDaysType also supports scalar coercion:

```
schedule.working_days = '1110000'
schedule.working_days  # WeekDays object
```

CHAPTER 4

Range data types

SQLAlchemy-Utils provides wide variety of range data types. All range data types return Interval objects of intervals package. In order to use range data types you need to install intervals with:

```
pip install intervals
```

Intervals package provides good chunk of additional interval operators that for example psycopg2 range objects do not support.

Some good reading for practical interval implementations:

http://wiki.postgresql.org/images/f/f0/Range-types.pdf

Range type initialization

```
from sqlalchemy_utils import IntRangeType


class Event(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(255))
    estimated_number_of_persons = sa.Column(IntRangeType)
```

You can also set a step parameter for range type. The values that are not multipliers of given step will be rounded up to nearest step multiplier.

```
from sqlalchemy_utils import IntRangeType


class Event(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(255))
```

```
    estimated_number_of_persons = sa.Column(IntRangeType(step=1000))


event = Event(estimated_number_of_persons=[100, 1200])
event.estimated_number_of_persons.lower  # 0
event.estimated_number_of_persons.upper  # 1000
```

Range type operators

SQLAlchemy-Utils supports many range type operators. These operators follow the *intervals* package interval coercion rules.

So for example when we make a query such as:

```
session.query(Car).filter(Car.price_range == 300)
```

It is essentially the same as:

```
session.query(Car).filter(Car.price_range == DecimalInterval([300, 300]))
```

Comparison operators

All range types support all comparison operators (>, >=, ==, !=, <=, <).

```
Car.price_range < [12, 300]

Car.price_range == [12, 300]

Car.price_range < 300

Car.price_range > (300, 500)

# Whether or not range is strictly left of another range
Car.price_range << [300, 500]

# Whether or not range is strictly right of another range
Car.price_range >> [300, 500]
```

Membership operators

```
Car.price_range.contains([300, 500])

Car.price_range.contained_by([300, 500])

Car.price_range.in_([[300, 500], [800, 900]])

~ Car.price_range.in_([[300, 400], [700, 800]])
```

Length

SQLAlchemy-Utils provides length property for all range types. The implementation of this property varies on different range types.

In the following example we find all cars whose price range's length is more than 500.

```
session.query(Car).filter(
    Car.price_range.length > 500
)
```

DateRangeType

class sqlalchemy_utils.types.range.**DateRangeType**(**args*, ***kwargs*)

DateRangeType provides way for saving ranges of dates into database. On PostgreSQL this type maps to native DATERANGE type while on other drivers this maps to simple string column.

Example:

```
from sqlalchemy_utils import DateRangeType


class Reservation(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, autoincrement=True)
    room_id = sa.Column(sa.Integer))
    during = sa.Column(DateRangeType)
```

DateTimeRangeType

class sqlalchemy_utils.types.range.**DateTimeRangeType**(**args*, ***kwargs*)

IntRangeType

class sqlalchemy_utils.types.range.**IntRangeType**(**args*, ***kwargs*)

IntRangeType provides way for saving ranges of integers into database. On PostgreSQL this type maps to native INT4RANGE type while on other drivers this maps to simple string column.

Example:

```
from sqlalchemy_utils import IntRangeType


class Event(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(255))
    estimated_number_of_persons = sa.Column(IntRangeType)


party = Event(name=u'party')
```

```
# we estimate the party to contain minium of 10 persons and at max
# 100 persons
party.estimated_number_of_persons = [10, 100]

print party.estimated_number_of_persons
# '10-100'
```

IntRangeType returns the values as IntInterval objects. These objects support many arithmetic operators:

```
meeting = Event(name=u'meeting')

meeting.estimated_number_of_persons = [20, 40]

total = (
    meeting.estimated_number_of_persons +
    party.estimated_number_of_persons
)
print total
# '30-140'
```

NumericRangeType

class sqlalchemy_utils.types.range.**NumericRangeType**(**args*, ***kwargs*)

NumericRangeType provides way for saving ranges of decimals into database. On PostgreSQL this type maps to native NUMRANGE type while on other drivers this maps to simple string column.

Example:

```
from sqlalchemy_utils import NumericRangeType


class Car(Base):
    __tablename__ = 'car'
    id = sa.Column(sa.Integer, autoincrement=True)
    name = sa.Column(sa.Unicode(255)))
    price_range = sa.Column(NumericRangeType)
```

RangeComparator

class sqlalchemy_utils.types.range.**RangeComparator**(*expr*)

CHAPTER 5

Aggregated attributes

SQLAlchemy-Utils provides way of automatically calculating aggregate values of related models and saving them to parent model.

This solution is inspired by RoR counter cache, counter_culture and stackoverflow reply by Michael Bayer.

Why?

Many times you may have situations where you need to calculate dynamically some aggregate value for given model. Some simple examples include:

- Number of products in a catalog
- Average rating for movie
- Latest forum post
- Total price of orders for given customer

Now all these aggregates can be elegantly implemented with SQLAlchemy column_property function. However when your data grows calculating these values on the fly might start to hurt the performance of your application. The more aggregates you are using the more performance penalty you get.

This module provides way of calculating these values automatically and efficiently at the time of modification rather than on the fly.

Features

- Automatically updates aggregate columns when aggregated values change
- Supports aggregate values through arbitrary number levels of relations
- Highly optimized: uses single query per transaction per aggregate column
- Aggregated columns can be of any data type and use any selectable scalar expression

Simple aggregates

```
from sqlalchemy_utils import aggregated


class Thread(Base):
    __tablename__ = 'thread'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))

    @aggregated('comments', sa.Column(sa.Integer))
    def comment_count(self):
        return sa.func.count('1')

    comments = sa.orm.relationship(
        'Comment',
        backref='thread'
    )


class Comment(Base):
    __tablename__ = 'comment'
    id = sa.Column(sa.Integer, primary_key=True)
    content = sa.Column(sa.UnicodeText)
    thread_id = sa.Column(sa.Integer, sa.ForeignKey(Thread.id))


thread = Thread(name=u'SQLAlchemy development')
thread.comments.append(Comment(u'Going good!'))
thread.comments.append(Comment(u'Great new features!'))

session.add(thread)
session.commit()

thread.comment_count  # 2
```

Custom aggregate expressions

Aggregate expression can be virtually any SQL expression not just a simple function taking one parameter. You can try things such as subqueries and different kinds of functions.

In the following example we have a Catalog of products where each catalog knows the net worth of its products.

```
from sqlalchemy_utils import aggregated


class Catalog(Base):
    __tablename__ = 'catalog'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))

    @aggregated('products', sa.Column(sa.Integer))
    def net_worth(self):
        return sa.func.sum(Product.price)
```

```
    products = sa.orm.relationship('Product')


class Product(Base):
    __tablename__ = 'product'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    price = sa.Column(sa.Numeric)

    catalog_id = sa.Column(sa.Integer, sa.ForeignKey(Catalog.id))
```

Now the net_worth column of Catalog model will be automatically whenever:

- A new product is added to the catalog
- A product is deleted from the catalog
- The price of catalog product is changed

```
from decimal import Decimal


product1 = Product(name='Some product', price=Decimal(1000))
product2 = Product(name='Some other product', price=Decimal(500))


catalog = Catalog(
    name=u'My first catalog',
    products=[
        product1,
        product2
    ]
)
session.add(catalog)
session.commit()

session.refresh(catalog)
catalog.net_worth  # 1500

session.delete(product2)
session.commit()
session.refresh(catalog)

catalog.net_worth  # 1000

product1.price = 2000
session.commit()
session.refresh(catalog)

catalog.net_worth  # 2000
```

Multiple aggregates per class

Sometimes you may need to define multiple aggregate values for same class. If you need to define lots of relationships pointing to same class, remember to define the relationships as viewonly when possible.

```
from sqlalchemy_utils import aggregated


class Customer(Base):
    __tablename__ = 'customer'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))

    @aggregated('orders', sa.Column(sa.Integer))
    def orders_sum(self):
        return sa.func.sum(Order.price)

    @aggregated('invoiced_orders', sa.Column(sa.Integer))
    def invoiced_orders_sum(self):
        return sa.func.sum(Order.price)

    orders = sa.orm.relationship('Order')

    invoiced_orders = sa.orm.relationship(
        'Order',
        primaryjoin=
            'sa.and_(Order.customer_id == Customer.id, Order.invoiced)',
        viewonly=True
    )


class Order(Base):
    __tablename__ = 'order'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    price = sa.Column(sa.Numeric)
    invoiced = sa.Column(sa.Boolean, default=False)
    customer_id = sa.Column(sa.Integer, sa.ForeignKey(Customer.id))
```

Many-to-Many aggregates

Aggregate expressions also support many-to-many relationships. The usual use scenarios includes things such as:

1. Friend count of a user
2. Group count where given user belongs to

```
user_group = sa.Table('user_group', Base.metadata,
    sa.Column('user_id', sa.Integer, sa.ForeignKey('user.id')),
    sa.Column('group_id', sa.Integer, sa.ForeignKey('group.id'))
)


class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))

    @aggregated('groups', sa.Column(sa.Integer, default=0))
    def group_count(self):
        return sa.func.count('1')
```

```
    groups = sa.orm.relationship(
        'Group',
        backref='users',
        secondary=user_group
    )


class Group(Base):
    __tablename__ = 'group'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))



user = User(name=u'John Matrix')
user.groups = [Group(name=u'Group A'), Group(name=u'Group B')]

session.add(user)
session.commit()

session.refresh(user)
user.group_count  # 2
```

Multi-level aggregates

Aggregates can span across multiple relationships. In the following example each Catalog has a net_worth which is the sum of all products in all categories.

```
from sqlalchemy_utils import aggregated


class Catalog(Base):
    __tablename__ = 'catalog'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))

    @aggregated('categories.products', sa.Column(sa.Integer))
    def net_worth(self):
        return sa.func.sum(Product.price)

    categories = sa.orm.relationship('Category')


class Category(Base):
    __tablename__ = 'category'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))

    catalog_id = sa.Column(sa.Integer, sa.ForeignKey(Catalog.id))

    products = sa.orm.relationship('Product')


class Product(Base):
```

```
    __tablename__ = 'product'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    price = sa.Column(sa.Numeric)

    category_id = sa.Column(sa.Integer, sa.ForeignKey(Category.id))
```

Examples

Average movie rating

```
from sqlalchemy_utils import aggregated


class Movie(Base):
    __tablename__ = 'movie'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))

    @aggregated('ratings', sa.Column(sa.Numeric))
    def avg_rating(self):
        return sa.func.avg(Rating.stars)

    ratings = sa.orm.relationship('Rating')


class Rating(Base):
    __tablename__ = 'rating'
    id = sa.Column(sa.Integer, primary_key=True)
    stars = sa.Column(sa.Integer)

    movie_id = sa.Column(sa.Integer, sa.ForeignKey(Movie.id))


movie = Movie('Terminator 2')
movie.ratings.append(Rating(stars=5))
movie.ratings.append(Rating(stars=4))
movie.ratings.append(Rating(stars=3))
session.add(movie)
session.commit()

movie.avg_rating  # 4
```

TODO

- Special consideration should be given to deadlocks.

sqlalchemy_utils.aggregates.**aggregated**(*relationship*, *column*)

Decorator that generates an aggregated attribute. The decorated function should return an aggregate select expression.

Parameters

- **relationship** – Defines the relationship of which the aggregate is calculated from. The class needs to have given relationship in order to calculate the aggregate.
- **column** – SQLAlchemy Column object. The column definition of this aggregate attribute.

CHAPTER 6

Observers

This module provides a decorator function for observing changes in a given property. Internally the decorator is implemented using SQLAlchemy event listeners. Both column properties and relationship properties can be observed.

Property observers can be used for pre-calculating aggregates and automatic real-time data denormalization.

Simple observers

At the heart of the observer extension is the *observes()* decorator. You mark some property path as being observed and the marked method will get notified when any changes are made to given path.

Consider the following model structure:

```
class Director(Base):
    __tablename__ = 'director'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String)
    date_of_birth = sa.Column(sa.Date)

class Movie(Base):
    __tablename__ = 'movie'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String)
    director_id = sa.Column(sa.Integer, sa.ForeignKey(Director.id))
    director = sa.orm.relationship(Director, backref='movies')
```

Now consider we want to show movies in some listing ordered by director id first and movie id secondly. If we have many movies then using joins and ordering by Director.name will be very slow. Here is where denormalization and *observes()* comes to rescue the day. Let's add a new column called director_name to Movie which will get automatically copied from associated Director.

```
from sqlalchemy_utils import observes
```

```
class Movie(Base):
    # same as before..
    director_name = sa.Column(sa.String)

    @observes('director')
    def director_observer(self, director):
        self.director_name = director.name
```

Note: This example could be done much more efficiently using a compound foreign key from director_name, director_id to Director.name, Director.id but for the sake of simplicity we added this as an example.

Observes vs aggregated

observes() and *aggregates.aggregated()* can be used for similar things. However performance wise you should take the following things into consideration:

- *observes()* works always inside transaction and deals with objects. If the relationship observer is observing has a large number of objects it's better to use *aggregates.aggregated()*.
- *aggregates.aggregated()* always executes one additional query per aggregate so in scenarios where the observed relationship has only a handful of objects it's better to use *observes()* instead.

Example 1. Movie with many ratings

Let's say we have a Movie object with potentially thousands of ratings. In this case we should always use *aggregates.aggregated()* since iterating through thousands of objects is slow and very memory consuming.

Example 2. Product with denormalized catalog name

Each product belongs to one catalog. Here it is natural to use *observes()* for data denormalization.

Deeply nested observing

Consider the following model structure where Catalog has many Categories and Category has many Products.

```
class Catalog(Base):
    __tablename__ = 'catalog'
    id = sa.Column(sa.Integer, primary_key=True)
    product_count = sa.Column(sa.Integer, default=0)

    @observes('categories.products')
    def product_observer(self, products):
        self.product_count = len(products)

    categories = sa.orm.relationship('Category', backref='catalog')

class Category(Base):
    __tablename__ = 'category'
    id = sa.Column(sa.Integer, primary_key=True)
    catalog_id = sa.Column(sa.Integer, sa.ForeignKey('catalog.id'))

    products = sa.orm.relationship('Product', backref='category')
```

```
class Product(Base):
    __tablename__ = 'product'
    id = sa.Column(sa.Integer, primary_key=True)
    price = sa.Column(sa.Numeric)

    category_id = sa.Column(sa.Integer, sa.ForeignKey('category.id'))
```

observes() is smart enough to:

- Notify catalog objects of any changes in associated Product objects
- Notify catalog objects of any changes in Category objects that affect products (for example if Category gets deleted, or a new Category is added to Catalog with any number of Products)

```
category = Category(
    products=[Product(), Product()]
)
category2 = Category(
    product=[Product()]
)

catalog = Catalog(
    categories=[category, category2]
)
session.add(catalog)
session.commit()
catalog.product_count  # 2

session.delete(category)
session.commit()
catalog.product_count  # 1
```

Observing multiple columns

You can also observe multiple columns by specifying all the observable columns in the decorator.

```
class Order(Base):
    __tablename__ = 'order'
    id = sa.Column(sa.Integer, primary_key=True)
    unit_price = sa.Column(sa.Integer)
    amount = sa.Column(sa.Integer)
    total_price = sa.Column(sa.Integer)

    @observes('amount', 'unit_price')
    def total_price_observer(self, amount, unit_price):
        self.total_price = amount * unit_price
```

sqlalchemy_utils.observer.**observes**(**paths*, ***observer_kw*)

Mark method as property observer for the given property path. Inside transaction observer gathers all changes made in given property path and feeds the changed objects to observer-marked method at the before flush phase.

```
from sqlalchemy_utils import observes


class Catalog(Base):
    __tablename__ = 'catalog'
```

```
    id = sa.Column(sa.Integer, primary_key=True)
    category_count = sa.Column(sa.Integer, default=0)

    @observes('categories')
    def category_observer(self, categories):
        self.category_count = len(categories)

class Category(Base):
    __tablename__ = 'category'
    id = sa.Column(sa.Integer, primary_key=True)
    catalog_id = sa.Column(sa.Integer, sa.ForeignKey('catalog.id'))


catalog = Catalog(categories=[Category(), Category()])
session.add(catalog)
session.commit()

catalog.category_count  # 2
```

Parameters

- ***paths** – One or more dot-notated property paths, eg. 'categories.products.price'
- ****observer** – A dictionary where value for key 'observer' contains `PropertyObserver()` object

CHAPTER 7

Internationalization

SQLAlchemy-Utils provides a way for modeling translatable models. Model is translatable if one or more of its columns can be displayed in various languages.

Note: The implementation is currently highly PostgreSQL specific since it needs a dict-compatible column type (PostgreSQL HSTORE and JSON are such types). If you want database-agnostic way of modeling i18n see SQLAlchemy-i18n.

TranslationHybrid vs SQLAlchemy-i18n

Compared to SQLAlchemy-i18n the TranslationHybrid has the following pros and cons:

- Usually faster since no joins are needed for fetching the data
- Less magic
- Easier to understand data model
- Only PostgreSQL supported for now

Quickstart

Let's say we have an Article model with translatable name and content. First we need to define the TranslationHybrid.

```
from sqlalchemy_utils import TranslationHybrid


# For testing purposes we define this as simple function which returns
# locale 'fi'. Usually you would define this function as something that
# returns the user's current locale.
def get_locale():
```

```
    return 'fi'


translation_hybrid = TranslationHybrid(
    current_locale=get_locale,
    default_locale='en'
)
```

Then we can define the model.:

```
from sqlalchemy import *
from sqlalchemy.dialects.postgresql import HSTORE


class Article(Base):
    __tablename__ = 'article'

    id = Column(Integer, primary_key=True)
    name_translations = Column(HSTORE)
    content_translations = Column(HSTORE)

    name = translation_hybrid(name_translations)
    content = translation_hybrid(content_translations)
```

Now we can start using our translatable model. By assigning things to translatable hybrids you are assigning them to the locale returned by the *current_locale*.

```
article = Article(name='Joku artikkeli')
article.name_translations['fi']  # Joku artikkeli
article.name  # Joku artikkeli
```

If you access the hybrid with a locale that doesn't exist the hybrid tries to fetch a the locale returned by *default_locale*.

```
article = Article(name_translations={'en': 'Some article'})
article.name  # Some article
article.name_translations['fi'] = 'Joku artikkeli'
article.name  # Joku artikkeli
```

Translation hybrids can also be used as expressions.

```
session.query(Article).filter(Article.name['en'] == 'Some article')
```

By default if no value is found for either current or default locale the translation hybrid returns *None*. You can customize this value with *default_value* parameter of translation_hybrid. In the following example we make translation hybrid fallback to empty string instead of *None*.

```
translation_hybrid = TranslationHybrid(
    current_locale=get_locale,
    default_locale='en',
    default_value=''
)


class Article(Base):
    __tablename__ = 'article'

    id = Column(Integer, primary_key=True)
    name_translations = Column(HSTORE)
```

```
    name = translation_hybrid(name_translations, default)


Article().name  # ''
```

Dynamic locales

Sometimes locales need to be dynamic. The following example illustrates how to setup dynamic locales.

```
translation_hybrid = TranslationHybrid(
    current_locale=get_locale,
    default_locale=lambda obj: obj.locale,
)


class Article(Base):
    __tablename__ = 'article'

    id = Column(Integer, primary_key=True)
    name_translations = Column(HSTORE)

    name = translation_hybrid(name_translations, default)
    locale = Column(String)


article = Article(name_translations={'en': 'Some article'})
session.add(article)
session.commit()

article.name  # Some article (even if current locale is other than 'en')
```

CHAPTER 8

Generic relationships

Generic relationship is a form of relationship that supports creating a 1 to many relationship to any target model.

```
from sqlalchemy_utils import generic_relationship

class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)

class Customer(Base):
    __tablename__ = 'customer'
    id = sa.Column(sa.Integer, primary_key=True)

class Event(Base):
    __tablename__ = 'event'
    id = sa.Column(sa.Integer, primary_key=True)

    # This is used to discriminate between the linked tables.
    object_type = sa.Column(sa.Unicode(255))

    # This is used to point to the primary key of the linked row.
    object_id = sa.Column(sa.Integer)

    object = generic_relationship(object_type, object_id)


# Some general usage to attach an event to a user.
user = User()
customer = Customer()

session.add_all([user, customer])
session.commit()

ev = Event()
ev.object = user
```

```
session.add(ev)
session.commit()

# Find the event we just made.
session.query(Event).filter_by(object=user).first()

# Find any events that are bound to users.
session.query(Event).filter(Event.object.is_type(User)).all()
```

Inheritance

```
class Employee(self.Base):
    __tablename__ = 'employee'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String(50))
    type = sa.Column(sa.String(20))

    __mapper_args__ = {
        'polymorphic_on': type,
        'polymorphic_identity': 'employee'
    }

class Manager(Employee):
    __mapper_args__ = {
        'polymorphic_identity': 'manager'
    }

class Engineer(Employee):
    __mapper_args__ = {
        'polymorphic_identity': 'engineer'
    }

class Activity(self.Base):
    __tablename__ = 'event'
    id = sa.Column(sa.Integer, primary_key=True)

    object_type = sa.Column(sa.Unicode(255))
    object_id = sa.Column(sa.Integer, nullable=False)

    object = generic_relationship(object_type, object_id)
```

Now same as before we can add some objects:

```
manager = Manager()

session.add(manager)
session.commit()

activity = Activity()
activity.object = manager

session.add(activity)
session.commit()

# Find the activity we just made.
```

```
session.query(Event).filter_by(object=manager).first()
```

We can even test super types:

```
session.query(Activity).filter(Event.object.is_type(Employee)).all()
```

Abstract base classes

Generic relationships also allows using string arguments. When using generic_relationship with abstract base classes you need to set up the relationship using declared_attr decorator and string arguments.

```
class Building(self.Base):
    __tablename__ = 'building'
    id = sa.Column(sa.Integer, primary_key=True)

class User(self.Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)

class EventBase(self.Base):
    __abstract__ = True

    object_type = sa.Column(sa.Unicode(255))
    object_id = sa.Column(sa.Integer, nullable=False)

    @declared_attr
    def object(cls):
        return generic_relationship('object_type', 'object_id')

class Event(EventBase):
    __tablename__ = 'event'
    id = sa.Column(sa.Integer, primary_key=True)
```

Composite keys

For some very rare cases you may need to use generic_relationships with composite primary keys. There is a limitation here though: you can only set up generic_relationship for similar composite primary key types. In other words you can't mix generic relationship to both composite keyed objects and single keyed objects.

```
from sqlalchemy_utils import generic_relationship


class Customer(Base):
    __tablename__ = 'customer'
    code1 = sa.Column(sa.Integer, primary_key=True)
    code2 = sa.Column(sa.Integer, primary_key=True)


class Event(Base):
    __tablename__ = 'event'
    id = sa.Column(sa.Integer, primary_key=True)
```

```
    # This is used to discriminate between the linked tables.
    object_type = sa.Column(sa.Unicode(255))

    object_code1 = sa.Column(sa.Integer)

    object_code2 = sa.Column(sa.Integer)

    object = generic_relationship(
        object_type, (object_code1, object_code2)
    )
```

CHAPTER 9

Database helpers

analyze

sqlalchemy_utils.functions.**analyze**(*conn*, *query*)

Analyze query using given connection and return `QueryAnalysis` object. Analysis is performed using database specific EXPLAIN ANALYZE construct and then examining the results into structured format. Currently only PostgreSQL is supported.

Getting query runtime (in database level)

```
from sqlalchemy_utils import analyze


analysis = analyze(conn, 'SELECT * FROM article')
analysis.runtime  # runtime as milliseconds
```

Analyze can be very useful when testing that query doesn't issue a sequential scan (scanning all rows in table). You can for example write simple performance tests this way.:

```
query = (
    session.query(Article.name)
    .order_by(Article.name)
    .limit(10)
)
analysis = analyze(self.connection, query)
analysis.node_types  # [u'Limit', u'Index Only Scan']

assert 'Seq Scan' not in analysis.node_types
```

Parameters

- **conn** – SQLAlchemy Connection object
- **query** – SQLAlchemy Query object or query as a string

database_exists

sqlalchemy_utils.functions.**database_exists**(*url*)

Check if a database exists.

Parameters **url** – A SQLAlchemy engine URL.

Performs backend-specific testing to quickly determine if a database exists on the server.

```
database_exists('postgres://postgres@localhost/name')  #=> False
create_database('postgres://postgres@localhost/name')
database_exists('postgres://postgres@localhost/name')  #=> True
```

Supports checking against a constructed URL as well.

```
engine = create_engine('postgres://postgres@localhost/name')
database_exists(engine.url)  #=> False
create_database(engine.url)
database_exists(engine.url)  #=> True
```

create_database

sqlalchemy_utils.functions.**create_database**(*url*, *encoding='utf8'*, *template=None*)

Issue the appropriate CREATE DATABASE statement.

Parameters

- **url** – A SQLAlchemy engine URL.
- **encoding** – The encoding to create the database as.
- **template** – The name of the template from which to create the new database. At the moment only supported by PostgreSQL driver.

To create a database, you can pass a simple URL that would have been passed to `create_engine`.

```
create_database('postgres://postgres@localhost/name')
```

You may also pass the url from an existing engine.

```
create_database(engine.url)
```

Has full support for mysql, postgres, and sqlite. In theory, other database engines should be supported.

drop_database

sqlalchemy_utils.functions.**drop_database**(*url*)

Issue the appropriate DROP DATABASE statement.

Parameters **url** – A SQLAlchemy engine URL.

Works similar to the create_database method in that both url text and a constructed url are accepted.

```
drop_database('postgres://postgres@localhost/name')
drop_database(engine.url)
```

has_index

sqlalchemy_utils.functions.**has_index**(*column_or_constraint*)

Return whether or not given column or the columns of given foreign key constraint have an index. A column has an index if it has a single column index or it is the first column in compound column index.

A foreign key constraint has an index if the constraint columns are the first columns in compound column index.

Parameters **column_or_constraint** – SQLAlchemy Column object or SA ForeignKeyConstraint object

```
from sqlalchemy_utils import has_index


class Article(Base):
    __tablename__ = 'article'
    id = sa.Column(sa.Integer, primary_key=True)
    title = sa.Column(sa.String(100))
    is_published = sa.Column(sa.Boolean, index=True)
    is_deleted = sa.Column(sa.Boolean)
    is_archived = sa.Column(sa.Boolean)

    __table_args__ = (
        sa.Index('my_index', is_deleted, is_archived),
    )


table = Article.__table__

has_index(table.c.is_published)  # True
has_index(table.c.is_deleted)    # True
has_index(table.c.is_archived)   # False
```

Also supports primary key indexes

```
from sqlalchemy_utils import has_index


class ArticleTranslation(Base):
    __tablename__ = 'article_translation'
    id = sa.Column(sa.Integer, primary_key=True)
    locale = sa.Column(sa.String(10), primary_key=True)
    title = sa.Column(sa.String(100))


table = ArticleTranslation.__table__

has_index(table.c.locale)    # False
has_index(table.c.id)        # True
```

This function supports foreign key constraints as well

```
class User(Base):
    __tablename__ = 'user'
    first_name = sa.Column(sa.Unicode(255), primary_key=True)
    last_name = sa.Column(sa.Unicode(255), primary_key=True)

class Article(Base):
```

```
    __tablename__ = 'article'
    id = sa.Column(sa.Integer, primary_key=True)
    author_first_name = sa.Column(sa.Unicode(255))
    author_last_name = sa.Column(sa.Unicode(255))
    __table_args__ = (
        sa.ForeignKeyConstraint(
            [author_first_name, author_last_name],
            [User.first_name, User.last_name]
        ),
        sa.Index(
            'my_index',
            author_first_name,
            author_last_name
        )
    )

table = Article.__table__
constraint = list(table.foreign_keys)[0].constraint

has_index(constraint)  # True
```

has_unique_index

sqlalchemy_utils.functions.**has_unique_index**(*column_or_constraint*)

Return whether or not given column or given foreign key constraint has a unique index.

A column has a unique index if it has a single column primary key index or it has a single column UniqueConstraint.

A foreign key constraint has a unique index if the columns of the constraint are the same as the columns of table primary key or the coluns of any unique index or any unique constraint of the given table.

Parameters **column** – SQLAlchemy Column object

```
from sqlalchemy_utils import has_unique_index


class Article(Base):
    __tablename__ = 'article'
    id = sa.Column(sa.Integer, primary_key=True)
    title = sa.Column(sa.String(100))
    is_published = sa.Column(sa.Boolean, unique=True)
    is_deleted = sa.Column(sa.Boolean)
    is_archived = sa.Column(sa.Boolean)


table = Article.__table__

has_unique_index(table.c.is_published) # True
has_unique_index(table.c.is_deleted)   # False
has_unique_index(table.c.id)           # True
```

This function supports foreign key constraints as well

```
class User(Base):
    __tablename__ = 'user'
```

```
    first_name = sa.Column(sa.Unicode(255), primary_key=True)
    last_name = sa.Column(sa.Unicode(255), primary_key=True)

class Article(Base):
    __tablename__ = 'article'
    id = sa.Column(sa.Integer, primary_key=True)
    author_first_name = sa.Column(sa.Unicode(255))
    author_last_name = sa.Column(sa.Unicode(255))
    __table_args__ = (
        sa.ForeignKeyConstraint(
            [author_first_name, author_last_name],
            [User.first_name, User.last_name]
        ),
        sa.Index(
            'my_index',
            author_first_name,
            author_last_name,
            unique=True
        )
    )

table = Article.__table__
constraint = list(table.foreign_keys)[0].constraint

has_unique_index(constraint)  # True
```

Raises **TypeError** – if given column does not belong to a Table object

json_sql

sqlalchemy_utils.functions.**json_sql**(*value*, *scalars_to_json=True*)

Convert python data structures to PostgreSQL specific SQLAlchemy JSON constructs. This function is extremly useful if you need to build PostgreSQL JSON on python side.

Note: This function needs PostgreSQL >= 9.4

Scalars are converted to to_json SQLAlchemy function objects

```
json_sql(1)     # Equals SQL: to_json(1)

json_sql('a')   # to_json('a')
```

Mappings are converted to json_build_object constructs

```
json_sql({'a': 'c', '2': 5})  # json_build_object('a', 'c', '2', 5)
```

Sequences (other than strings) are converted to json_build_array constructs

```
json_sql([1, 2, 3])  # json_build_array(1, 2, 3)
```

You can also nest these data structures

```
json_sql({'a': [1, 2, 3]})
# json_build_object('a', json_build_array[1, 2, 3])
```

Parameters **value** – value to be converted to SQLAlchemy PostgreSQL function constructs

render_expression

sqlalchemy_utils.functions.**render_expression**(*expression*, *bind*, *stream=None*)

Generate a SQL expression from the passed python expression.

Only the global variable, *engine*, is available for use in the expression. Additional local variables may be passed in the context parameter.

Note this function is meant for convenience and protected usage. Do NOT blindly pass user input to this function as it uses exec.

Parameters

- **bind** – A SQLAlchemy engine or bind URL.
- **stream** – Render all DDL operations to the stream.

render_statement

sqlalchemy_utils.functions.**render_statement**(*statement*, *bind=None*)

Generate an SQL expression string with bound parameters rendered inline for the given SQLAlchemy statement.

Parameters

- **statement** – SQLAlchemy Query object.
- **bind** – Optional SQLAlchemy bind, if None uses the bind of the given query object.

CHAPTER 10

Foreign key helpers

dependent_objects

sqlalchemy_utils.functions.**dependent_objects**(*obj*, *foreign_keys=None*)

Return a *QueryChain* that iterates through all dependent objects for given SQLAlchemy object.

Consider a User object is referenced in various articles and also in various orders. Getting all these dependent objects is as easy as:

```
from sqlalchemy_utils import dependent_objects


dependent_objects(user)
```

If you expect an object to have lots of dependent_objects it might be good to limit the results:

```
dependent_objects(user).limit(5)
```

The common use case is checking for all restrict dependent objects before deleting parent object and inform the user if there are dependent objects with ondelete='RESTRICT' foreign keys. If this kind of checking is not used it will lead to nasty IntegrityErrors being raised.

In the following example we delete given user if it doesn't have any foreign key restricted dependent objects:

```
from sqlalchemy_utils import get_referencing_foreign_keys


user = session.query(User).get(some_user_id)


deps = list(
    dependent_objects(
        user,
        (
            fk for fk in get_referencing_foreign_keys(User)
```

```
            # On most databases RESTRICT is the default mode hence we
            # check for None values also
            if fk.ondelete == 'RESTRICT' or fk.ondelete is None
        )
    ).limit(5)
)

if deps:
    # Do something to inform the user
    pass
else:
    session.delete(user)
```

Parameters

- **obj** – SQLAlchemy declarative model object
- **foreign_keys** – A sequence of foreign keys to use for searching the dependent_objects for given object. By default this is None, indicating that all foreign keys referencing the object will be used.

Note: This function does not support exotic mappers that use multiple tables

---

See also:

get_referencing_foreign_keys()

See also:

merge_references()

get_referencing_foreign_keys

sqlalchemy_utils.functions.**get_referencing_foreign_keys**(*mixed*)

Returns referencing foreign keys for given Table object or declarative class.

Parameters **mixed** – SA Table object or SA declarative class

```
get_referencing_foreign_keys(User)  # set([ForeignKey('user.id')])

get_referencing_foreign_keys(User.__table__)
```

This function also understands inheritance. This means it returns all foreign keys that reference any table in the class inheritance tree.

Let's say you have three classes which use joined table inheritance, namely TextItem, Article and BlogPost with Article and BlogPost inheriting TextItem.

```
# This will check all foreign keys that reference either article table
# or textitem table.
get_referencing_foreign_keys(Article)
```

See also:

get_tables()

group_foreign_keys

sqlalchemy_utils.functions.**group_foreign_keys**(*foreign_keys*)

Return a groupby iterator that groups given foreign keys by table.

Parameters **foreign_keys** – a sequence of foreign keys

```
foreign_keys = get_referencing_foreign_keys(User)

for table, fks in group_foreign_keys(foreign_keys):
    # do something
    pass
```

See also:

get_referencing_foreign_keys()

is_indexed_foreign_key

merge_references

sqlalchemy_utils.functions.**merge_references**(*from_*, *to*, *foreign_keys=None*)

Merge the references of an entity into another entity.

Consider the following models:

```
class User(self.Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String(255))

    def __repr__(self):
        return 'User(name=%r)' % self.name

class BlogPost(self.Base):
    __tablename__ = 'blog_post'
    id = sa.Column(sa.Integer, primary_key=True)
    title = sa.Column(sa.String(255))
    author_id = sa.Column(sa.Integer, sa.ForeignKey('user.id'))

    author = sa.orm.relationship(User)
```

Now lets add some data:

```
john = self.User(name='John')
jack = self.User(name='Jack')
post = self.BlogPost(title='Some title', author=john)
post2 = self.BlogPost(title='Other title', author=jack)
self.session.add_all([
    john,
    jack,
    post,
    post2
])
self.session.commit()
```

If we wanted to merge all John's references to Jack it would be as easy as

```
merge_references(john, jack)
self.session.commit()

post.author      # User(name='Jack')
post2.author     # User(name='Jack')
```

Parameters

- **from** – an entity to merge into another entity
- **to** – an entity to merge another entity into
- **foreign_keys** – A sequence of foreign keys. By default this is None indicating all referencing foreign keys should be used.

non_indexed_foreign_keys

sqlalchemy_utils.functions.**non_indexed_foreign_keys**(*metadata*, *engine=None*)

Finds all non indexed foreign keys from all tables of given MetaData.

Very useful for optimizing postgresql database and finding out which foreign keys need indexes.

Parameters **metadata** – MetaData object to inspect tables from

CHAPTER 11

ORM helpers

cast_if

sqlalchemy_utils.functions.**cast_if**(*expression*, *type_*)

Produce a CAST expression but only if given expression is not of given type already.

Assume we have a model with two fields id (Integer) and name (String).

```
import sqlalchemy as sa
from sqlalchemy_utils import cast_if


cast_if(User.id, sa.Integer)     # "user".id
cast_if(User.name, sa.String)    # "user".name
cast_if(User.id, sa.String)      # CAST("user".id AS TEXT)
```

This function supports scalar values as well.

```
cast_if(1, sa.Integer)           # 1
cast_if('text', sa.String)       # 'text'
cast_if(1, sa.String)            # CAST(1 AS TEXT)
```

Parameters

- **expression** – A SQL expression, such as a ColumnElement expression or a Python string which will be coerced into a bound literal value.
- **type** – A TypeEngine class or instance indicating the type to which the CAST should apply.

escape_like

sqlalchemy_utils.functions.**escape_like**(*string*, *escape_char='*'*)

Escape the string paremeter used in SQL LIKE expressions.

```
from sqlalchemy_utils import escape_like


query = session.query(User).filter(
    User.name.ilike(escape_like('John'))
)
```

Parameters

- **string** – a string to escape
- **escape_char** – escape character

get_bind

sqlalchemy_utils.functions.**get_bind**(*obj*)

Return the bind for given SQLAlchemy Engine / Connection / declarative model object.

Parameters **obj** – SQLAlchemy Engine / Connection / declarative model object

```
from sqlalchemy_utils import get_bind


get_bind(session)  # Connection object

get_bind(user)
```

get_class_by_table

sqlalchemy_utils.functions.**get_class_by_table**(*base*, *table*, *data=None*)

Return declarative class associated with given table. If no class is found this function returns *None*. If multiple classes were found (polymorphic cases) additional *data* parameter can be given to hint which class to return.

```
class User(Base):
    __tablename__ = 'entity'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String)


get_class_by_table(Base, User.__table__)  # User class
```

This function also supports models using single table inheritance. Additional data paratemer should be provided in these case.

```
class Entity(Base):
    __tablename__ = 'entity'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String)
    type = sa.Column(sa.String)
    __mapper_args__ = {
        'polymorphic_on': type,
        'polymorphic_identity': 'entity'
    }
```

```
class User(Entity):
    __mapper_args__ = {
        'polymorphic_identity': 'user'
    }


# Entity class
get_class_by_table(Base, Entity.__table__, {'type': 'entity'})

# User class
get_class_by_table(Base, Entity.__table__, {'type': 'user'})
```

Parameters

- **base** – Declarative model base
- **table** – SQLAlchemy Table object
- **data** – Data row to determine the class in polymorphic scenarios

Returns Declarative class or None.

get_column_key

sqlalchemy_utils.functions.**get_column_key**(*model*, *column*)

Return the key for given column in given model.

Parameters **model** – SQLAlchemy declarative model object

```
class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column('_name', sa.String)


get_column_key(User, User.__table__.c._name)  # 'name'
```

get_columns

sqlalchemy_utils.functions.**get_columns**(*mixed*)

Return a collection of all Column objects for given SQLAlchemy object.

The type of the collection depends on the type of the object to return the columns from.

```
get_columns(User)

get_columns(User())

get_columns(User.__table__)

get_columns(User.__mapper__)

get_columns(sa.orm.aliased(User))
```

```
get_columns(sa.orm.alised(User.__table__))
```

Parameters **mixed** – SA Table object, SA Mapper, SA declarative class, SA declarative class instance or an alias of any of these objects

get_declarative_base

sqlalchemy_utils.functions.**get_declarative_base**(*model*)

Returns the declarative base for given model class.

Parameters **model** – SQLAlchemy declarative model

get_hybrid_properties

sqlalchemy_utils.functions.**get_hybrid_properties**(*model*)

Returns a dictionary of hybrid property keys and hybrid properties for given SQLAlchemy declarative model / mapper.

Consider the following model

```
from sqlalchemy.ext.hybrid import hybrid_property


class Category(Base):
    __tablename__ = 'category'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))

    @hybrid_property
    def lowercase_name(self):
        return self.name.lower()

    @lowercase_name.expression
    def lowercase_name(cls):
        return sa.func.lower(cls.name)
```

You can now easily get a list of all hybrid property names

```
from sqlalchemy_utils import get_hybrid_properties


get_hybrid_properties(Category).keys()  # ['lowercase_name']
```

This function also supports aliased classes

```
get_hybrid_properties(
    sa.orm.aliased(Category)
).keys()  # ['lowercase_name']
```

Parameters **model** – SQLAlchemy declarative model or mapper

get_mapper

sqlalchemy_utils.functions.**get_mapper**(*mixed*)

Return related SQLAlchemy Mapper for given SQLAlchemy object.

Parameters **mixed** – SQLAlchemy Table / Alias / Mapper / declarative model object

```
from sqlalchemy_utils import get_mapper


get_mapper(User)

get_mapper(User())

get_mapper(User.__table__)

get_mapper(User.__mapper__)

get_mapper(sa.orm.aliased(User))

get_mapper(sa.orm.aliased(User.__table__))
```

Raises: ValueError: if multiple mappers were found for given argument

get_query_entities

sqlalchemy_utils.functions.**get_query_entities**(*query*)

Return a list of all entities present in given SQLAlchemy query object.

Examples:

```
from sqlalchemy_utils import get_query_entities


query = session.query(Category)

get_query_entities(query)  # [<Category>]


query = session.query(Category.id)

get_query_entities(query)  # [<Category>]
```

This function also supports queries with joins.

```
query = session.query(Category).join(Article)

get_query_entities(query)  # [<Category>, <Article>]
```

Parameters **query** – SQLAlchemy Query object

get_primary_keys

sqlalchemy_utils.functions.**get_primary_keys**(*mixed*)

Return an OrderedDict of all primary keys for given Table object, declarative class or declarative class instance.

Parameters **mixed** – SA Table object, SA declarative class or SA declarative class instance

```
get_primary_keys(User)

get_primary_keys(User())

get_primary_keys(User.__table__)

get_primary_keys(User.__mapper__)

get_primary_keys(sa.orm.aliased(User))

get_primary_keys(sa.orm.aliased(User.__table__))
```

See also:

get_columns()

get_tables

sqlalchemy_utils.functions.**get_tables**(*mixed*)

Return a set of tables associated with given SQLAlchemy object.

Let's say we have three classes which use joined table inheritance TextItem, Article and BlogPost. Article and BlogPost inherit TextItem.

```
get_tables(Article)  # set([Table('article', ...), Table('text_item')])

get_tables(Article())

get_tables(Article.__mapper__)
```

If the TextItem entity is using with_polymorphic='*' then this function returns all child tables (article and blog_post) as well.

```
get_tables(TextItem)  # set([Table('text_item', ...)], ...])
```

Parameters **mixed** – SQLAlchemy Mapper, Declarative class, Column, InstrumentedAttribute or a SA Alias object wrapping any of these objects.

get_type

sqlalchemy_utils.functions.**get_type**(*expr*)

Return the associated type with given Column, InstrumentedAttribute, ColumnProperty, RelationshipProperty or other similar SQLAlchemy construct.

For constructs wrapping columns this is the column type. For relationships this function returns the relationship mapper class.

Parameters **expr** – SQLAlchemy Column, InstrumentedAttribute, ColumnProperty or other similar SA construct.

```
class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String)


class Article(Base):
    __tablename__ = 'article'
    id = sa.Column(sa.Integer, primary_key=True)
    author_id = sa.Column(sa.Integer, sa.ForeignKey(User.id))
    author = sa.orm.relationship(User)


get_type(User.__table__.c.name)  # sa.String()
get_type(User.name)  # sa.String()
get_type(User.name.property)  # sa.String()

get_type(Article.author)  # User
```

has_changes

sqlalchemy_utils.functions.**has_changes**(*obj*, *attrs=None*, *exclude=None*)

Simple shortcut function for checking if given attributes of given declarative model object have changed during the session. Without parameters this checks if given object has any modificiations. Additionally exclude parameter can be given to check if given object has any changes in any attributes other than the ones given in exclude.

```
from sqlalchemy_utils import has_changes


user = User()

has_changes(user, 'name')  # False

user.name = u'someone'

has_changes(user, 'name')  # True

has_changes(user)  # True
```

You can check multiple attributes as well.

```
has_changes(user, ['age'])  # True

has_changes(user, ['name', 'age'])  # True
```

This function also supports excluding certain attributes.

```
has_changes(user, exclude=['name'])  # False

has_changes(user, exclude=['age'])  # True
```

Parameters

- **obj** – SQLAlchemy declarative model object
- **attrs** – Names of the attributes
- **exclude** – Names of the attributes to exclude

identity

sqlalchemy_utils.functions.**identity**(*obj_or_class*)

Return the identity of given sqlalchemy declarative model class or instance as a tuple. This differs from obj._sa_instance_state.identity in a way that it always returns the identity even if object is still in transient state (new object that is not yet persisted into database). Also for classes it returns the identity attributes.

```
from sqlalchemy import inspect
from sqlalchemy_utils import identity


user = User(name=u'John Matrix')
session.add(user)
identity(user)  # None
inspect(user).identity  # None

session.flush()  # User now has id but is still in transient state

identity(user)  # (1,)
inspect(user).identity  # None

session.commit()

identity(user)  # (1,)
inspect(user).identity  # (1, )
```

You can also use identity for classes:

```
identity(User)  # (User.id, )
```

Parameters **obj** – SQLAlchemy declarative model object

is_loaded

sqlalchemy_utils.functions.**is_loaded**(*obj*, *prop*)

Return whether or not given property of given object has been loaded.

```
class Article(Base):
    __tablename__ = 'article'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String)
    content = sa.orm.deferred(sa.Column(sa.String))


article = session.query(Article).get(5)
```

```
# name gets loaded since its not a deferred property
assert is_loaded(article, 'name')

# content has not yet been loaded since its a deferred property
assert not is_loaded(article, 'content')
```

Parameters

- **obj** – SQLAlchemy declarative model object
- **prop** – Name of the property or InstrumentedAttribute

make_order_by_deterministic

sqlalchemy_utils.functions.**make_order_by_deterministic**(*query*)

Make query order by deterministic (if it isn't already). Order by is considered deterministic if it contains column that is unique index (either it is a primary key or has a unique index). Many times it is design flaw to order by queries in nondeterministic manner.

Consider a User model with three fields: id (primary key), favorite color and email (unique).:

```
from sqlalchemy_utils import make_order_by_deterministic


query = session.query(User).order_by(User.favorite_color)

query = make_order_by_deterministic(query)
print query  # 'SELECT ... ORDER BY "user".favorite_color, "user".id'


query = session.query(User).order_by(User.email)

query = make_order_by_deterministic(query)
print query  # 'SELECT ... ORDER BY "user".email'


query = session.query(User).order_by(User.id)

query = make_order_by_deterministic(query)
print query  # 'SELECT ... ORDER BY "user".id'
```

naturally_equivalent

sqlalchemy_utils.functions.**naturally_equivalent**(*obj*, *obj2*)

Returns whether or not two given SQLAlchemy declarative instances are naturally equivalent (all their non primary key properties are equivalent).

```
from sqlalchemy_utils import naturally_equivalent


user = User(name=u'someone')
user2 = User(name=u'someone')
```

```
user == user2  # False

naturally_equivalent(user, user2)  # True
```

Parameters

- **obj** – SQLAlchemy declarative model object
- **obj2** – SQLAlchemy declarative model object to compare with *obj*

quote

sqlalchemy_utils.functions.**quote**(*mixed*, *ident*)

Conditionally quote an identifier.

```
from sqlalchemy_utils import quote


engine = create_engine('sqlite:///:memory:')

quote(engine, 'order')
# '"order"'

quote(engine, 'some_other_identifier')
# 'some_other_identifier'
```

Parameters

- **mixed** – SQLAlchemy Session / Connection / Engine / Dialect object.
- **ident** – identifier to conditionally quote

sort_query

sqlalchemy_utils.functions.**sort_query**(*query*, **args*, ***kwargs*)

Applies an sql ORDER BY for given query. This function can be easily used with user-defined sorting.

The examples use the following model definition:

```
import sqlalchemy as sa
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy_utils import sort_query


engine = create_engine(
    'sqlite:///'
)
Base = declarative_base()
Session = sessionmaker(bind=engine)
session = Session()

class Category(Base):
```

```
    __tablename__ = 'category'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))

class Article(Base):
    __tablename__ = 'article'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    category_id = sa.Column(sa.Integer, sa.ForeignKey(Category.id))

    category = sa.orm.relationship(
        Category, primaryjoin=category_id == Category.id
    )
```

1. Applying simple ascending sort

```
query = session.query(Article)
query = sort_query(query, 'name')
```

2. Applying descending sort

```
query = sort_query(query, '-name')
```

3. Applying sort to custom calculated label

```
query = session.query(
    Category, sa.func.count(Article.id).label('articles')
)
query = sort_query(query, 'articles')
```

4. Applying sort to joined table column

```
query = session.query(Article).join(Article.category)
query = sort_query(query, 'category-name')
```

Parameters

- **query** – query to be modified
- **sort** – string that defines the label or column to sort the query by
- **silent** – Whether or not to raise exceptions if unknown sort column is passed. By default this is *True* indicating that no errors should be raised for unknown columns.

CHAPTER 12

Utility classes

QueryChain

QueryChain is a wrapper for sequence of queries.

Features:

- Easy iteration for sequence of queries
- Limit, offset and count which are applied to all queries in the chain
- Smart __getitem__ support

Initialization

QueryChain takes iterable of queries as first argument. Additionally limit and offset parameters can be given

```
chain = QueryChain([session.query(User), session.query(Article)])

chain = QueryChain(
    [session.query(User), session.query(Article)],
    limit=4
)
```

Simple iteration

```
chain = QueryChain([session.query(User), session.query(Article)])

for obj in chain:
    print obj
```

Limit and offset

Lets say you have 5 blog posts, 5 articles and 5 news items in your database.

```
chain = QueryChain(
    [
        session.query(BlogPost),
        session.query(Article),
        session.query(NewsItem)
    ],
    limit=5
)

list(chain)  # all blog posts but not articles and news items


chain = chain.offset(4)
list(chain)  # last blog post, and first four articles
```

Just like with original query object the limit and offset can be chained to return a new QueryChain.

```
chain = chain.limit(5).offset(7)
```

Chain slicing

```
chain = QueryChain(
    [
        session.query(BlogPost),
        session.query(Article),
        session.query(NewsItem)
    ]
)

chain[3:6]   # New QueryChain with offset=3 and limit=6
```

Count

Let's assume that there are five blog posts, five articles and five news items in the database, and you have the following query chain:

```
chain = QueryChain(
    [
        session.query(BlogPost),
        session.query(Article),
        session.query(NewsItem)
    ]
)
```

You can then get the total number rows returned by the query chain with *count()*:

```
>>> chain.count()
15
```

API

class sqlalchemy_utils.query_chain.**QueryChain**(*queries*, *limit=None*, *offset=None*)

QueryChain can be used as a wrapper for sequence of queries.

Parameters

- **queries** – A sequence of SQLAlchemy Query objects
- **limit** – Similar to normal query limit this parameter can be used for limiting the number of results for the whole query chain.
- **offset** – Similar to normal query offset this parameter can be used for offsetting the query chain as a whole.

count()

Return the total number of rows this QueryChain's queries would return.

CHAPTER 13

Model mixins

Timestamp

class sqlalchemy_utils.models.**Timestamp**

Adds *created* and *updated* columns to a derived declarative model.

The *created* column is handled through a default and the *updated* column is handled through a *before_update* event that propagates for all derived declarative models.

```
import sqlalchemy as sa
from sqlalchemy_utils import Timestamp


class SomeModel(Base, Timestamp):
    __tablename__ = 'somemodel'
    id = sa.Column(sa.Integer, primary_key=True)
```

generic_repr

sqlalchemy_utils.models.**generic_repr**(**fields*)

Adds generic `__repr__()` method to a decalrative SQLAlchemy model.

In case if some fields are not loaded from a database, it doesn't force their loading and instead repesents them as `<not loaded>`.

In addition, user can provide field names as arguments to the decorator to specify what fields should present in the string representation and in what order.

Example:

```
import sqlalchemy as sa
from sqlalchemy_utils import generic_repr
```

```
@generic_repr
class MyModel(Base):
    __tablename__ = 'mymodel'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String)
    category = sa.Column(sa.String)

session.add(MyModel(name='Foo', category='Bar'))
session.commit()
foo = session.query(MyModel).options(sa.orm.defer('category')).one(s)

assert repr(foo) == 'MyModel(id=1, name='Foo', category=<not loaded>)'
```

CHAPTER 14

Testing

The functions in this module can be used for testing that the constraints of your models. Each assert function runs SQL UPDATEs that check for the existence of given constraint. Consider the following model:

```
class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.String(200), nullable=True)
    email = sa.Column(sa.String(255), nullable=False)


user = User(name='John Doe', email='john@example.com')
session.add(user)
session.commit()
```

We can easily test the constraints by assert_* functions:

```
from sqlalchemy_utils import (
    assert_nullable,
    assert_non_nullable,
    assert_max_length
)

assert_nullable(user, 'name')
assert_non_nullable(user, 'email')
assert_max_length(user, 'name', 200)

# raises AssertionError because the max length of email is 255
assert_max_length(user, 'email', 300)
```

assert_min_value

sqlalchemy_utils.asserts.**assert_min_value**(*obj*, *column*, *min_value*)

Assert that the given column must have a minimum value of *min_value*.

Parameters

- **obj** – SQLAlchemy declarative model object
- **column** – Name of the column
- **min_value** – The minimum allowed value for given column

assert_max_length

sqlalchemy_utils.asserts.**assert_max_length**(*obj*, *column*, *max_length*)

Assert that the given column is of given max length. This function supports string typed columns as well as PostgreSQL array typed columns.

In the following example we add a check constraint that user can have a maximum of 5 favorite colors and then test this.:

```
class User(Base):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    favorite_colors = sa.Column(ARRAY(sa.String), nullable=False)
    __table_args__ = (
        sa.CheckConstraint(
            sa.func.array_length(favorite_colors, 1) <= 5
        )
    )


user = User(name='John Doe', favorite_colors=['red', 'blue'])
session.add(user)
session.commit()


assert_max_length(user, 'favorite_colors', 5)
```

Parameters

- **obj** – SQLAlchemy declarative model object
- **column** – Name of the column
- **max_length** – Maximum length of given column

assert_max_value

sqlalchemy_utils.asserts.**assert_max_value**(*obj*, *column*, *min_value*)

Assert that the given column must have a minimum value of *max_value*.

Parameters

- **obj** – SQLAlchemy declarative model object

- **column** – Name of the column
- **max_value** – The maximum allowed value for given column

assert_nullable

sqlalchemy_utils.asserts.**assert_nullable**(*obj*, *column*)

Assert that given column is nullable. This is checked by running an SQL update that assigns given column as None.

Parameters

- **obj** – SQLAlchemy declarative model object
- **column** – Name of the column

assert_non_nullable

sqlalchemy_utils.asserts.**assert_non_nullable**(*obj*, *column*)

Assert that given column is not nullable. This is checked by running an SQL update that assigns given column as None.

Parameters

- **obj** – SQLAlchemy declarative model object
- **column** – Name of the column

CHAPTER 15

License

Python Module Index

S

Index

U

W