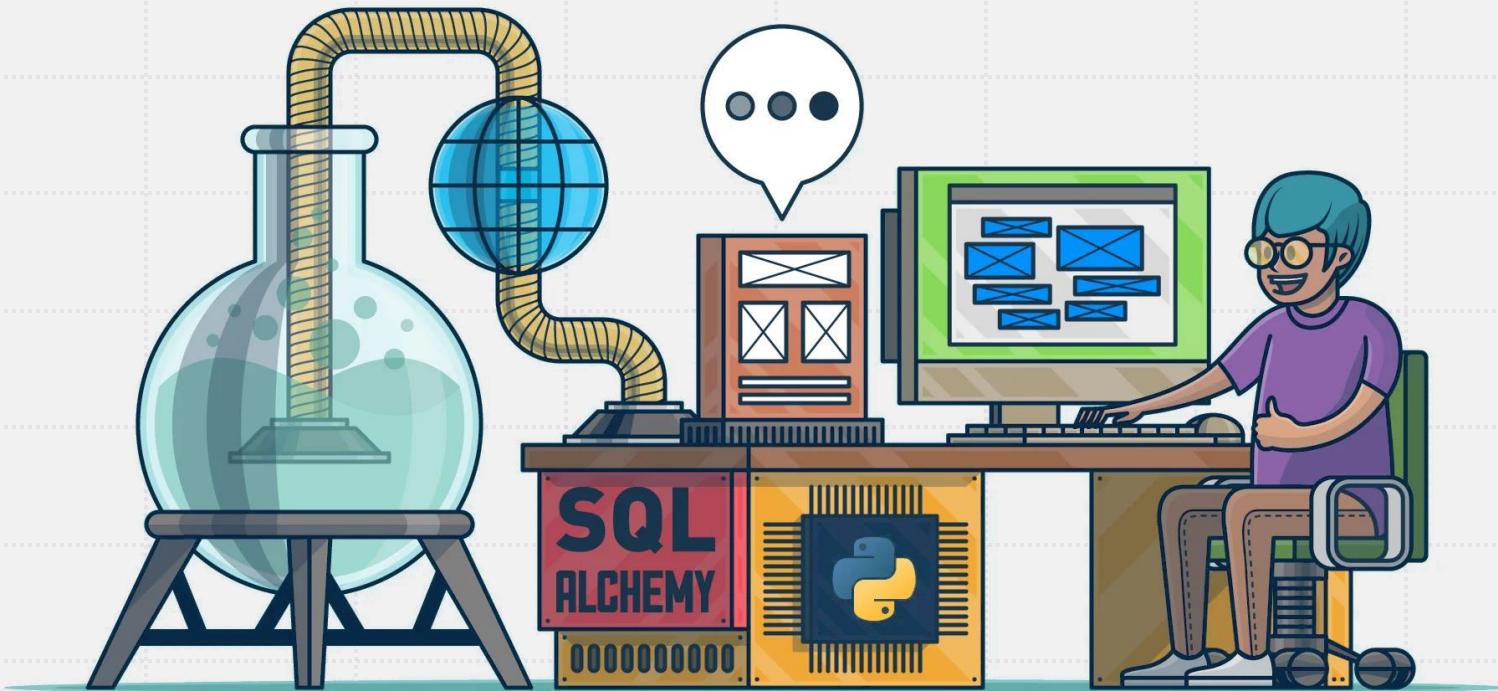


Programowanie baz danych

sqlalchemy 1



Real Python

Plan na dzisiaj

- Przegląd narzędzi do pracy z bazami w Pythonie
- Wstęp do SQLAlchemy

Przegląd narzędzi bazodanowych dla Pythona

Connectors





The world's most popular open source database

Contact MySQL | Login | Register

[f](#) [Twitter](#) [in](#) [YouTube](#)

Search this Manual

Abstract

mysql-connector-Python

- Preface and Legal Notices
 - Introduction to MySQL Connector/Python
 - Guidelines for Python Developers
 - Connector/Python Versions
 - ▶ Connector/Python Installation
 - ▶ Connector/Python Coding Examples
 - ▶ Connector/Python Tutorials
 - ▶ Connector/Python Connection Establishment
 - ▶ The Connector/Python C Extension
 - ▶ Connector/Python Other Topics
 - ▶ Connector/Python API Reference
 - ▶ Connector/Python C Extension API Reference

The latest MySQL Connector/Python version is recommended for use with MySQL Server version 5.7 and higher.

For notes detailing the changes in each release of Connector/Python, see MySQL Connector/Python Release Notes.

For legal information, see the Legal Notices.

For help with using MySQL, please visit the [MySQL Forums](#), where you can discuss your issues with other MySQL users.

Licensing information. This product may include third-party software, used under license. If you are using a Commercial release of MySQL Connector/Python, see the [MySQL Connector/Python 8.3 Commercial License Information User Manual](#) for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a Community release of MySQL Connector/Python, see the [MySQL Connector/Python 8.3 Community License Information User Manual](#) for licensing information, including licensing information relating to third-party software that may be included in this Community release.

Document generated on: 2024-03-11 (revision: 78035)

Welcome to MySQLdb's documentation

https://mysqlclient.readthedocs.io

mysqlclient latest

Search docs

MySQLdb User's Guide

MySQLdb Package

MySQLdb Frequently Asked Questions

Contents:

- MySQLdb User's Guide
 - Introduction
 - Installation
 - MySQLdb._mysql
 - MySQL C API translation
 - MySQL C API function mapping
 - Some _mysql examples
 - MySQLdb
 - Functions and attributes
 - Connection Objects
 - Cursor Objects
 - Some examples
 - Using and extending
- MySQLdb Package
 - MySQLdb Package
 - MySQLdb.connections Module
 - MySQLdb.converters Module
 - MySQLdb.cursors Module
 - MySQLdb.times Module
 - MySQLdb._mysql Module
 - MySQLdb._exceptions Module

Welcome to MySQLdb's documentation!

Edit on GitHub

mysqlclient

Read the Docs v: latest

Psycopg – PostgreSQL database adapter for Python

Psycopg is the most popular PostgreSQL database adapter for the Python programming language. Its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety (several threads can share the same connection). It was designed for heavily multi-threaded applications that create and destroy lots of cursors and make a large number of concurrent INSERTS OR UPDATES.

Psycopg 2 is mostly implemented in C as a C API wrapper, resulting in being both efficient and secure. It features client-side and server-side cursors, asynchronous communication and notifications, COPY support. Many Python types are supported out-of-the-box and adapted to matching PostgreSQL data types; adaptation can be extended and customized thanks to a flexible objects adaptation system.

Psycopg 2 is both Unicode and Python 3 friendly.

Contents

- Installation
 - Quick Install
 - Prerequisites
 - Non-standard builds
 - Running the test suite
 - If you still have problems
- Basic module usage
 - Passing parameters to SQL queries
 - Adaptation of Python values to SQL types
 - Transactions control

ORMs

SQLAlchemy - The Database Toolkit for Python

https://www.sqlalchemy.org

SQLAlchemy THE DATABASE TOOLKIT FOR PYTHON

home features blog library community download

The Python SQL Toolkit and Object Relational Mapper

SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL. It provides a full suite of well known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple and Pythonic domain language.

SQLALCHEMY'S PHILOSOPHY

SQL databases behave less like object collections the more size and performance start to matter; object collections behave less like tables and rows the more abstraction starts to matter. SQLAlchemy aims to accommodate both of these principles.

SQLAlchemy considers the database to be a relational algebra engine, not just a collection of tables. Rows can be selected from not only tables but also joins and other select statements; any of these units can be composed into a larger structure. SQLAlchemy's expression language builds on this concept from its core.

SQLAlchemy is most famous for its object-relational mapper (ORM), an optional component that provides the **data mapper pattern**, where classes can be mapped to the database in open ended, multiple ways - allowing the object model and database schema to develop in a cleanly decoupled way from the beginning.

SQLAlchemy's overall approach to these problems is entirely different from that of most other SQL / ORM tools, rooted in a so-called **complimentarity**- oriented approach; instead of hiding away SQL and object relational details behind a wall of automation, all processes are **fully exposed** within a series of composable, transparent tools. The library takes on the job of automating redundant tasks while the developer remains in control of how the database is organized and how SQL is constructed.

The main goal of SQLAlchemy is to change the way you think about databases and SQL!

Read some **key features of SQLAlchemy**, as well as **what people are saying** about SQLAlchemy.

CURRENT RELEASES

1.4.21 - 2021-07-14 - announce
[changes](#) | [migration notes](#) | [docs](#)

1.3.24 - 2021-03-30 - announce
[changes](#) | [migration notes](#) | [docs](#)

PagerDuty
Use tools,
not snake oil.
[Listen to Podcast](#)

Let software do the work of driving culture change.

ADS VIA CARBON

SPONSOR SQLALCHEMY!

[Donate](#) [Donate to SQLAlchemy through PayPal](#)

 Sponsor SQLAlchemy through the Tidelift Subscription

LATEST NEWS

SQLAlchemy 1.4.21 Released
Wed, 14 Jul 2021

SQLAlchemy 1.4.20 Released
Mon, 28 Jun 2021

SQLAlchemy 1.4.19 Released

Flask-SQLAlchemy — Flask-SQL 2.x

https://flask-sqlalchemy.palletsprojects.com/en/2.x/

Logo

Project Links

[Donate to Pallets](#)
[Website](#)
[PyPI releases](#)
[Source Code](#)
[Issue Tracker](#)

Contents

[Flask-SQLAlchemy](#)
▪ Requirements
▪ User Guide
▪ API Reference
▪ Additional Information

Quick search

Go

Flask SQLAlchemy

Flask SQLAlchemy is an extension for [Flask](#) that adds support for [SQLAlchemy](#) to your application. It aims to simplify using SQLAlchemy with Flask by providing useful defaults and extra helpers that make it easier to accomplish common tasks.

See the [SQLAlchemy documentation](#) to learn how to work with the ORM in depth. The following documentation is a brief overview of the most common tasks, as well as the features specific to Flask-SQLAlchemy.

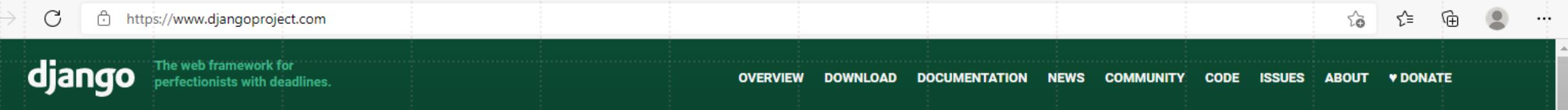
Requirements

Our Version	Python	Flask	SQLAlchemy
2.1	2.7, 3.4+	0.12+	0.8+ or 1.0.10+ w/ Python 3.7
3.0+ (in dev)	2.7, 3.5+	1.0+	1.0+

User Guide

- [Quickstart](#)
 - [A Minimal Application](#)
 - [Simple Relationships](#)
 - [Road to Enlightenment](#)
- [Introduction into Contexts](#)
- [Configuration](#)
 - [Configuration Keys](#)
 - [Connection URI Format](#)
 - [Using custom MetaData and naming conventions](#)
 - [Timeouts](#)

v: 2.x



Django makes it easier to build better Web apps more quickly and with less code.

Get started with Django

Django ORM

Meet Django

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.



Ridiculously fast.

Django was designed to help developers take applications from concept to completion

Download latest release: 3.2.5

[DJANGO DOCUMENTATION >](#)

Support Django!



McInnes Cooper donated to the Django Software Foundation to support Django development. [Donate today!](#)

peewee — peewee 3.14.4 documentation

Niezabezpieczona | docs.peewee-orm.com/en/latest/ | Edit on GitHub

peewee latest

Docs » peewee

peewee

Peewee is a simple and small ORM. It has few (but expressive) concepts, making it easy to learn and intuitive to use.

- a small, expressive ORM
- python 2.7+ and 3.4+ (developed with 3.6)
- supports sqlite, mysql, postgresql and cockroachdb
- tons of extensions

Peewee's source code hosted on [GitHub](#).

New to peewee? These may help:

Install and Testing
Quickstart
Example app
Using Peewee Interactively
Contributing
Database
Models and Fields
Querying
Query operators
Relationships and Joins
API Documentation
SQLite Extensions
Hayhouse, extensions to Peewee
Query Examples
Query Builder
Fixes
Changes in 3.0
Read the Docs

v: latest ▾



Object-Relational Mapper

Docs

Online editor

Releases

Pony is a Python ORM with beautiful query syntax

Write your database queries using Python generators & lambdas

[Try PonyORM now](#)[Support PonyORM](#)

Pony ORM

Free open-source software



Github



Twitter



Telegram



Join the newsletter

Custom software development

Query builders



Advertise without compromises Niche

targeted GDPR compliant ads for
developers [Learn more!](#)

Ad by EthicalAds

 Edit on GitHub

PyPika - Python Query Builder

[build](#) [unknown](#) [coverage](#)  [98%](#) [docs](#) [passing](#) [pypi](#) [v0.48.9](#) [license](#) [Apache-2.0](#)

Abstract

What is PyPika?

PyPika is a Python API for building SQL queries. The motivation behind PyPika is to provide a simple interface for building SQL queries without limiting the flexibility of handwritten SQL. Designed with data analysis in mind, PyPika leverages the builder design pattern to construct queries to avoid messy string formatting and concatenation. It is also easily extended to take full advantage of specific features of SQL database vendors.

What are the design goals for *PyPika*?

PyPika is a fast, expressive and flexible way to replace handwritten SQL (or even ORM for the courageous souls amongst you). Validation of SQL correctness is not an explicit goal of PyPika. With such a large number of SQL database vendors providing a robust validation of input data is difficult. Instead you are encouraged to check inputs you provide to PyPika or appropriately handle errors raised from your SQL database - just as you would have if you were writing SQL yourself.

Contents

- Installation
 - Tutorial
 - Selecting Data
 - Tables, Columns, Schemas, and Databases

SQLAlchemy 2.0

Wstęp







0010011101101010100110101010111001





001001110110101010100110101010111001

```
CREATE TABLE user
(
    user_id INT NOT NULL PRIMARY KEY,
);
```





ORACLE®

001001110110101010011010101011001



```
CREATE TABLE user
(
    user_id numeric(10) not null,
    CONSTRAINT user_pk PRIMARY KEY (user_id)
);
```



SQL
Alchemy

0010011101101010100110101010111001





SQL
Alchemy



0010011101101010100110101010111001





SQL
Alchemy



0010011101101010100110101010111001

```
Table('user', metadata,
      Column('user_id', Integer, primary_key=True)
     )
```





SQL
Alchemy



```
Table('user', metadata,
      Column('user_id', Integer, primary_key=True)
     )
```



0010011101101010100110101010111001

```
CREATE TABLE user
(
    user_id INT NOT NULL PRIMARY KEY,
);
```



SQL
Alchemy



0010011101101010100110101010111001

ORACLE®



```
Table('user', metadata,
      Column('user_id', Integer, primary_key=True)
     )
```



SQL
Alchemy



0010011101101010100110101010111001

```
Table('user', metadata,
      Column('user_id', Integer, primary_key=True)
     )
```

ORACLE®



```
CREATE TABLE user
(
    user_id numeric(10) not null,
    CONSTRAINT user_pk PRIMARY KEY (user_id)
);
```

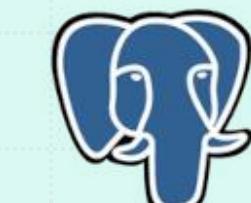


SQL
Alchemy

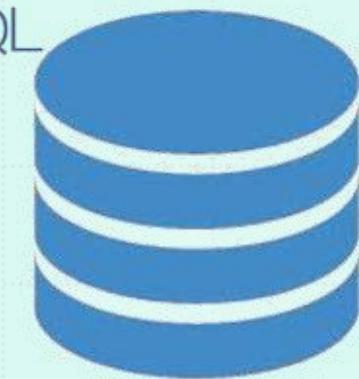


```
Table('user', metadata,
      Column('user_id', Integer, primary_key=True)
     )
```

0010011101101010100110101010111001



PostgreSQL



ORACLE®



MySQL®



Microsoft®
SQL Server



SQLAlchemy wstęp



SQLAlchemy - The Database Toolkit for Python

home features news documentation community download

The Python SQL Toolkit and Object Relational Mapper

SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

It provides a full suite of well known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple and Pythonic domain language.

Documentation

- [Current Documentation \(version 2.0\)](#) - learn SQLAlchemy here
 - Documentation Overview
 - Installation Guide
 - ORM Quickstart
 - Comprehensive Tutorial
 - Reference Guides
 - Object Relational Mapping (ORM)
 - Core (Connections, Schema Management, SQL)
 - Dialects (specific backends)
- [Documentation by Version](#)
 - [Version 2.1 \(development\)](#)
 - [Version 2.0](#)
 - [Version 1.4](#)
 - [Version 1.3](#)

CURRENT RELEASES

2.0.28 - 2024-03-04 - [announce](#)
[changes](#) | [migration notes](#) | [docs](#)

pypi v2.0.28 downloads 109M/month
python 3.7 | 3.8 | 3.9 | 3.10 | 3.11 | 3.12

1.4.52 - 2024-03-04 - [announce](#)
[changes](#) | [migration notes](#) | [docs](#)

2.1 - next major series
[What's New in 2.1?](#) | [docs](#)

EthicalAds
Reach your exact developer niche with GDPR compliant, contextually targeted ads

SQLAlchemy documentation

SQLAlchemy Documentation — +

https://docs.sqlalchemy.org/en/20/index.html

SQLAlchemy THE DATABASE TOOLKIT FOR PYTHON

home features news documentation community download

SQLAlchemy 2.0 Documentation

Release: 2.0.28 CURRENT RELEASE | Release Date: March 4, 2024

Contents | Index | Download this Documentation

Search terms: search... Search

Getting Started

New to SQLAlchemy? Start here:

- **For Python Beginners:** [Installation Guide](#) - basic guidance on installing with pip and similar
- **For Python Veterans:** [SQLAlchemy Overview](#) - brief architectural overview

Tutorials

New users of SQLAlchemy, as well as veterans of older SQLAlchemy release series, should start with the [SQLAlchemy Unified Tutorial](#), which covers everything an Alchemist needs to know when using the ORM or just Core.

- **For a quick glance:** [ORM Quick Start](#) - a glimpse at what working with the ORM looks like
- **For all users:** [SQLAlchemy Unified Tutorial](#) - In depth tutorial for Core and ORM

Migration Notes

Users coming from older versions of SQLAlchemy, especially those transitioning from the 1.x style of working, will want to review this documentation.

- [Migrating to SQLAlchemy 2.0](#) - Complete background on migrating from 1.3 or 1.4 to 2.0
- [What's New in SQLAlchemy 2.0?](#) - New 2.0 features and behaviors beyond the 1.x migration
- [Changelog catalog](#) - Detailed changelogs for all SQLAlchemy Versions

Reference and How To

SQLAlchemy ORM - Detailed guides and API reference for using the ORM

- **Mapping Classes:** [Mapping Python Classes](#) | [Relationship Configuration](#)
- **Using the ORM:** [Using the ORM Session](#) | [ORM Querying Guide](#) | [Using AsyncIO](#)
- **Configuration Extensions:** [Association Proxy](#) | [Hybrid Attributes](#) | [Mutable Scalars](#) | [Automap](#) | [All extensions](#)
- **Extending the ORM:** [ORM Events and Internals](#)
- **Other:** [ORM Configuration](#) | [ORM Session Configuration](#) | [ORM Querying Configuration](#) | [ORM Configuration Examples](#)

SQLAlchemy Core - Detailed guides and API reference for working with Core

- **Engines, Connections, Pools:** [Engine Configuration](#) | [Connections](#), [Transactions](#), [Results](#) | [AsyncIO Support](#) | [Connection Pooling](#)
- **Schema Definition:** [Overview](#) | [Introspection / Reflection](#) | [DDL](#) | [DML](#) | [DQL](#)

SQLAlchemy 2.0 documentation

SQLAlchemy 2.0.28

`pip install SQLAlchemy`

Released: Mar 4, 2024

Database Abstraction Library

Navigation

≡ Project description

Release history

 Download files

Project links

 Homepage

 Documentation

Project description

[pypi](#) v2.0.28 [python](#) 3.7 | 3.8 | 3.9 | 3.10 | 3.11 | 3.12 [downloads/month](#) 103M

The Python SQL Toolkit and Object Relational Mapper

Introduction

SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL. SQLAlchemy provides a full suite of well known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple and elegant API that is natural to work with.

Major SQLAlchemy features include:

- An induction

SQLAlchemy PyPI

sqlalchemy / sqlalchemy

Type ⌂ to search

Code Issues Pull requests Discussions Actions Projects Wiki Security Insights

 sqlalchemy Public

Sponsor Watch 95 Fork 1.3k Star 8.5k

main 19 Branches 307 Tags Go to file Add file Code

zzzeek	document caveat for #11054	10fb132 · 3 hours ago	16,855 Commits
.github	Bump pypa/cibuildwheel from 2.16.2 to 2.16.5 (#10947)	last month	
doc/build	document caveat for #11054	3 hours ago	
examples	Update black to 24.1.1	last month	
lib/sqlalchemy	use a fixed date that's not near DST switchover	4 hours ago	
test	use a fixed date that's not near DST switchover	4 hours ago	
tools	Add PEP 646 integration	2 months ago	
.git-blame-ignore-revs	update .git-blame-ignore-revs to exclude black update com...	last month	
.gitignore	Better syncronize async result docs with plain ones.	2 years ago	
.gitreview	create main branch in gerrit	3 years ago	
.pre-commit-config.yaml	Update black to 24.1.1		
AUTHORS	capitalize author name		

About

The Database Toolkit for Python

www.sqlalchemy.org

python sqlalchemy sql

Readme MIT license Code of conduct Security policy Activity Custom properties 8.5k stars 95 watching 1.3k forks Report repository

SQLAlchemy Github

TUTORIALS

The most up-to-date and complete tutorials available for getting started with SQLAlchemy are: * the [SQLAlchemy 1.4/2.0 Tutorial](#) which is a full rewrite of the classic "1.x" SQLAlchemy tutorials; users starting with the latest SQLAlchemy releases should start here. * the [Core](#) and [ORM](#) tutorials are recommended for those using "1.x style" codebases. A few other online resources include:

- [SQLAlchemy 2.0 - The One-Point-Four-Ening 2021 - Python Web Conf 2021](#)

Author: Mike Bayer



This is the newest version of the "getting started" tutorial that presents SQLAlchemy from the perspective of the new 2.0 series.

- [Video](#)
- [Student Download](#)
- [Introduction to SQLAlchemy](#) - presented at many Pycon and other conferences

SQLAlchemy tutorials

Dialects — SQLAlchemy 2.0 Documentation

https://docs.sqlalchemy.org/en/20/dialects/index.html

SQLAlchemy

THE DATABASE TOOLKIT FOR PYTHON

home features news documentation community download

SQLAlchemy 2.0 Documentation

Release: 2.0.28 CURRENT RELEASE | Release Date: March 4, 2024

Dialects

The **dialect** is the system SQLAlchemy uses to communicate with various types of DBAPI implementations and databases. The sections that follow contain reference documentation and notes specific to the usage of each backend, as well as notes for the various DBAPIs.

All dialects require that an appropriate DBAPI driver is installed.

Included Dialects

- PostgreSQL
- MySQL and MariaDB
- SQLite
- Oracle
- Microsoft SQL Server

Support Levels for Included Dialects

The following table summarizes the support level for each included dialect.

Database	Fully tested in CI	Normal support	Best effort
Microsoft SQL Server	2017	2012+	2005+
MySQL / MariaDB	5.6, 5.7, 8.0 / 10.8, 10.9	5.6+ / 10+	5.0.2+ / 5.0.2+
Oracle	18c	11+	9+
PostgreSQL	12, 13, 14, 15	9.6+	9+
SQLite	3.36.0	3.12+	3.7.16+

Dialekty

Supported database versions for included dialects

MySQL and MariaDB — SQLAlchemy

https://docs.sqlalchemy.org/en/20/dialects/mysql.html

SQLAlchemy

THE DATABASE TOOLKIT FOR PYTHON

home features news documentation community download

SQLAlchemy 2.0 Documentation

Release: 2.0.28 CURRENT RELEASE | Release Date: March 4, 2024

SQLAlchemy 2.0 Documentation

CURRENT RELEASE

Home | Download this Documentation

Search terms:

EthicalAds Reach your exact developer niche with GDPR compliant, contextually targeted ads

Ads by EthicalAds

Dialects

- PostgreSQL
- MySQL and MariaDB
 - Support for the MySQL / MariaDB database.
 - Supported Versions and Features
 - MariaDB Support
 - MariaDB-Only Mode
 - Connection Timeouts and Disconnects
 - CREATE TABLE arguments including Storage Engines
 - Case Sensitivity and Table Reflection
 - Transaction Isolation Level
 - AUTO_INCREMENT Behavior

MySQL and MariaDB

Support for the MySQL / MariaDB database.

The following table summarizes current support levels for database release versions.

Support type	Versions
Fully tested in CI	5.6, 5.7, 8.0 / 10.8, 10.9
Normal support	5.6+ / 10+
Best effort	5.0.2+ / 5.0.2+

Supported MySQL / MariaDB versions

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- mysqlclient (maintained fork of MySQL-Python)
- PyMySQL
- MariaDB Connector/Python
- MySQL Connector/Python
- asyncmy
- aiomysql
- CyMySQL
- PuNDRC

Sterowniki mysql

MySQL and MariaDB — SQLAlchemy 2.0 Documentation

CURRENT RELEASE

Home | Download this Documentation

Search terms: search...

Advertise without compromises Niche targeted GDPR compliant ads for developers Learn more!

Ads by EthicalAds

Dialects

- PostgreSQL
- MySQL and MariaDB**
 - Support for the MySQL / MariaDB database.
 - Supported Versions and Features
 - MariaDB Support
 - MariaDB-Only Mode
 - Connection Timeouts and Disconnects
 - CREATE TABLE arguments including Storage Engines
 - Case Sensitivity and Table Reflection
 - Transaction Isolation Level
 - AUTO_INCREMENT Behavior
 - Server Side Cursors
 - Unicode
 - Charset Selection
 - Dealing with Binary Data Warnings and Unicode

MySQL-Connector

Support for the MySQL / MariaDB database via the MySQL Connector/Python driver.

DBAPI

Documentation and download information (if applicable) for MySQL Connector/Python is available at: <https://pypi.org/project/mysql-connector-python/>

Connecting

Connect String:

```
mysql+mysqlconnector://<user>:<password>@<host>[:<port>]/<dbname>
```

Note

The MySQL Connector/Python DBAPI has had many issues since its release, some of which may remain unresolved, and the mysqlconnector dialect is **not tested as part of SQLAlchemy's continuous integration**. The recommended MySQL dialects are mysqlclient and PyMySQL.

asyncmy

Support for the MySQL / MariaDB database via the asyncmy driver.

DBAPI

Documentation and download information (if applicable) for asyncmy is available at: <https://github.com/long2ice/asyncmy>

Connecting

Connect String:

```
mysql+asyncmy://user:password@host:port/dbname[?key=value&key=value...]
```

Using a special asyncio mediation layer, the asyncmy dialect is usable as the backend for the SQLAlchemy `asyncio` extension package.

This dialect should normally be used only with the `create_async_engine()` engine creation function:

```
from sqlalchemy.ext.asyncio import create_async_engine
engine = create_async_engine("mysql+asyncmy://user:pass@hostname/dbname?charset=utf8mb4")
```

**Sterownik
mysql-connector-python**

Architektura



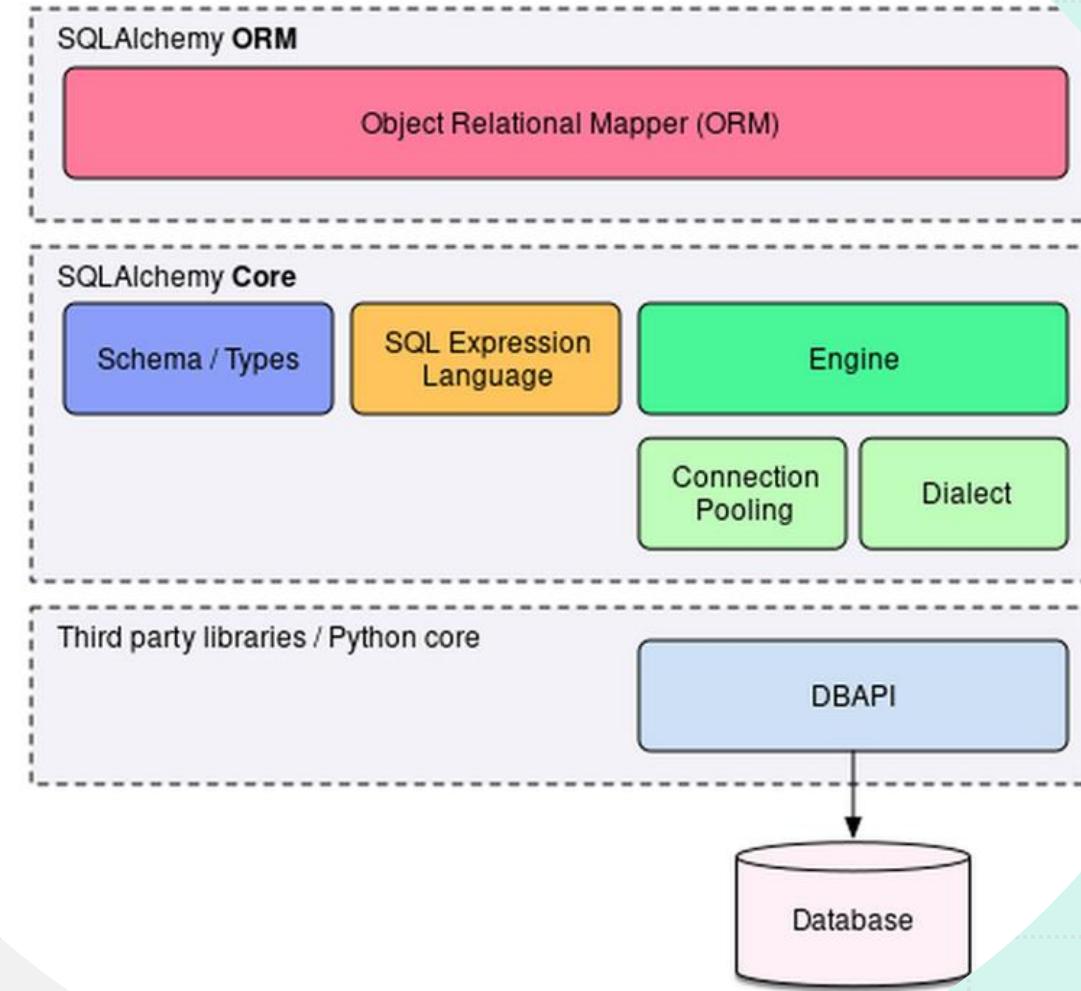
Dwa podstawowe modele użycia

SQLAlchemy składa się z dwóch API, jednym zbudowanym "na szczeblu" drugiego:

1. **Core** aka SQL Expression Language
2. **ORM** (Object Relational Mapper)



Podstawowa architektura



Dialekty (+sterowniki)



SQLAlchemy 1.3 Documentation

Release: **1.3.12 CURRENT RELEASE** | Release Date: December 16, 2019

SQLAlchemy 1.3
Documentation

CURRENT RELEASE

[Contents](#) | [Index](#)

Search terms:



Bring your team together
with Slack, the
collaboration hub for
work.

SQLite

Support for the SQLite database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- [pysqlite](#)
- [pysqlcipher](#)

Date and Time Types

SQLite does not have built-in DATE, TIME, or DATETIME types, and pysqlite does not implement them. SQLAlchemy provides a bridge between Python `datetime` objects and a SQLite-supported format. SQLAlchemy's `DateTime` and `Time` types are mapped to the SQLite `TEXT` type when SQLite is used. The implementation classes are `NaiveDateTime` and `NaiveTime`. Both also nicely support ordering. There's no relic support for the `DATE`, `TIME`, and `DATETIME` types.

SQLAlchemy 1.3 Documentation

Release: **1.3.12** **CURRENT RELEASE** | Release Date: December 16, 2019

SQLAlchemy 1.3
Documentation

CURRENT RELEASE

[Contents](#) | [Index](#)

Search terms:

fastly

Hello, edge cloud. So
long, slow delivery. Start
testing now for \$0.

ads via Carbon

PostgreSQL

Support for the PostgreSQL database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- [psycopg2](#)
- [pg8000](#)
- [psycopg2cffi](#)
- [py-postgresql](#)
- [pygresql](#)
- [zxJDBC for Jython](#)

Sequences/SERIAL/IDENTITY

SQLAlchemy 1.3 Documentation

Release: **1.3.12 CURRENT RELEASE** | Release Date: December 16, 2018

SQLAlchemy 1.3
Documentation

CURRENT RELEASE

[Contents](#) | [Index](#)

Search terms:

DATASTAX
LUNA

DataStax Luna - budget-friendly, flexible support for open source Apache Cassandra™.

MySQL

Support for the MySQL database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- mysqlclient (maintained fork of MySQL-Python)
- PyMySQL
- MySQL Connector/Python
- CyMySQL
- OurSQL
- Google Cloud SQL
- PyODBC
- zxjdb for Jython

Supported Versions

SQLAlchemy 1.3 Documentation

Release: **1.3.12** **CURRENT RELEASE** | Release Date: December 16, 2018

SQLAlchemy 1.3
Documentation

CURRENT RELEASE

[Contents](#) | [Index](#)

Search terms:



Bring your team together
with Slack, the
collaboration hub for
work.

Oracle

Support for the Oracle database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- [cx-Oracle](#)
- [zxJDBC for Jython](#)

Connect Arguments

The dialect supports several `create_engine()` arguments which affect the behavior of the dialect:

- `use_ansi` - Use ANSI JOIN constructs (see the section on [join constructs](#))
- `optimize_limits` - defaults to `True`

SQLAlchemy 1.3 Documentation

Release: **1.3.12 CURRENT RELEASE** | Release Date: December 16, 2018

SQLAlchemy 1.3
Documentation

CURRENT RELEASE

[Contents](#) | [Index](#)

Search terms:

D2
IQ

Kommander: Delivering
centralized governance

ads via Carbon

Microsoft SQL Server

Support for the Microsoft SQL Server database.

DBAPI Support

The following dialect/DBAPI options are available. Please refer to individual DBAPI sections for connect information.

- [PyODBC](#)
- [mxODBC](#)
- [pymssql](#)
- [zxJDBC for Jython](#)
- [adodbapi](#)

Auto Increment Behavior / IDENTITY Columns

SQL Server provides so-called “auto increment” columns, which are generated by the database system.

Instalacja



Instalacja

```
(.venv) C:\Users\User\PycharmProjects\database_programming\mysql_connector_python\documentation>pip install sqlalchemy
```

Weryfikacja

```
(.venv) C:\Users\User\PycharmProjects\database_programming\mysql_connector_python\documentation>python  
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022, 19:58:39) [MSC v.1934 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> import sqlalchemy  
>>> sqlalchemy.__version__  
'2.0.28'
```



Napis połączeniowy (ang. connection string)



Definicja

Napis połączeniowy (ang. connection string) to napis zawierający wszystkie niezbędne informacje do połączenia z bazą danych. Wśród informacji znajdują się:

1. dialect bazy (`sqlite`, `postgresql`, `mysql`, ...)
2. sterownik (ang. connector aka driver) - biblioteka Pythona, która pozwala na pracę z konkretnym dialektem
3. lokalizacja i nazwa bazy



Anatomia

Postać pełna

dialect[+driver]://user:password@hostname/dbname[?key=value]



Przykłady

SQLite

(using *pysqLite* driver)

```
'sqlLITE:///db_name.db'
```

PostgreSQL

(using *psycopg2* driver)

```
'postgreSQL://xavier:postgres@localhost:5432/db_name'
```

MySQL

(using *mysql-connector-python* driver)

```
'mysql+mysqlconnector://root:mysql@localhost:3306/db_name'
```



Przykład mysql

```
py
cnx_string = "mysql+mysqlconnector://root:admin@localhost"
```



Silnik (ang. Engine)



Opis

Silnik to obiekty klasy `Engine` w SQLAlchemy. Zapewnia połączenie z bazą oraz obsługuje operacje wykonywane na bazie. Jest centralnym źródłem połączeń z bazą. Stanowi fabrykę połączeń oraz zarządza pulą połączeń. Jest leniwy (leniwa inicjalizacja, ang. *lazy initialization*).

Połączenie z bazą wymaga przekazania do silnika napisu połączeniowego.

Zalecanym sposobem tworzenia obiektu klasy silnik jest funkcja `create_engine`.



Przykład inicjalizacji silnika

```
from sqlalchemy import create_engine  
  
cnx_string = "mysql+mysqlconnector://root:admin@localhost"  
engine = create_engine(cnx_string)
```



Funkcja `create_engine` tworzy obiekty klasy Engine

```
def create_engine(url: Union[str, _url.URL], **kwargs: Any) -> Engine:  
    """Create a new :class:`_engine.Engine` instance.  
  
    The standard calling form is to send the :ref:`URL <database_urls>` as the  
    first positional argument, usually a string  
    that indicates database dialect and connection arguments::  
  
        engine = create_engine("postgresql+psycopg2://scott:tiger@localhost/test")  
  
    .. note::  
  
        Please review :ref:`database_urls` for general guidelines in composing  
        URL strings. In particular, special characters, such as those often  
        part of passwords, must be URL encoded to be properly parsed.  
  
    Additional keyword arguments may then follow it which  
    establish various options on the resulting :class:`_engine.Engine`  
    and its underlying :class:`.Dialect` and :class:`_pool.Pool`  
    constructs::  
  
        engine = create_engine("mysql+mysqldb://scott:tiger@hostname/dbname",  
                               pool_recycle=3600, echo=True)
```

The string form of the URL is



Klasa **Engine** komponuje się z obiektu klasy **Pool** (odpowiedzialnej za zarządzanie połączeniem) oraz obiektu klasy **Dialect** (odpowiedzialnej za generowanie poprawnych literałów SQL)

```
class Engine(  
    ConnectionEventsTarget, log.Identified, inspection.Inspectable["Inspector"]  
):  
    ...  
  
    dialect: Dialect  
    pool: Pool  
    url: URL  
    hide_parameters: bool  
  
    def __init__(  
        self,  
        pool: Pool,  
        dialect: Dialect,  
        url: URL,  
        logging_name: Optional[str] = None,  
        echo: Optional[_EchoFlagType] = None,  
        query_cache_size: int = 500,  
        execution_options: Optional[Mapping[str, Any]] = None,  
        hide_parameters: bool = False,  
    ):  
        self.pool = pool  
        self.url = url
```



```
In [1]: from sqlalchemy import create_engine
```

```
In [2]: engine = create_engine("mysql+mysqlconnector://root:admin@localhost")
```

```
In [3]: print(dir(engine))
```

```
['__annotations__', '__class__', '__class_getitem__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__orig_bases__', '__parameters__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__subclasshook__', '__weakref__', '_compiled_cache', '_connection_cls', '_echo', '_execution_options', '_has_events', '_is_future', '_is_protocol', '_lru_size_alert', '_option_cls', '_optional_conn_ctx_manager', '_run_ddl_visitor', '_schema_translate_map', '_should_log_debug', '_should_log_info', '_sqlalchemy_namespace', 'begin', 'clear_compiled_cache', 'connect', 'dialect', 'dispatch', 'dispose', 'driver', 'echo', 'engine', 'execution_options', 'parameters', 'logger', 'logging_name', 'name', 'pool', 'raw_connection', 'update_execution_options']
```

Klasa Engine

Wśród metod klasy **Engine** znajduje się metoda **connect**.



Metoda **connect** zwraca obiekt klasy **Connection**

```
def connect(self) -> Connection:  
    """Return a new :class:`_engine.Connection` object.  
  
    The :class:`_engine.Connection` acts as a Python context manager, so  
    the typical use of this method looks like::  
  
        with engine.connect() as connection:  
            connection.execute(text("insert into table values ('foo')"))  
            connection.commit()  
  
    Where above, after the block is completed, the connection is "closed"  
    and its underlying DBAPI resources are returned to the connection pool.  
    This also has the effect of rolling back any transaction that  
    was explicitly begun or was begun via autobegin, and will  
    emit the :meth:`_events.ConnectionEvents.rollback` event if one was  
    started and is still in progress.  
  
    .. seealso::  
  
        :meth:`_engine.Engine.begin`  
  
    """  
  
    return self._connection_cls(self)
```



Klasa **Connection** reprezentuje połączenie i jest menadżerem kontekstu

```
class Connection(ConnectionEventsTarget, inspection.Inspectable["Inspector"]):
    """Provides high-level functionality for a wrapped DB-API connection.

    The :class:`_engine.Connection` object is procured by calling the
    :meth:`_engine.Engine.connect` method of the :class:`_engine.Engine`
    object, and provides services for execution of SQL statements as well
    as transaction control.

    The Connection object is **not** thread-safe. While a Connection can be
    shared among threads using properly synchronized access, it is still
    possible that the underlying DBAPI connection may not support shared
    access between threads. Check the DBAPI documentation for details.

    The Connection object represents a single DBAPI connection checked out
    from the connection pool. In this state, the connection pool has no
    affect upon the connection, including its expiration or timeout state.
    For the connection pool to properly manage connections, connections
    should be returned to the connection pool (i.e. ``connection.close()``)
    whenever the connection is not in use.

    .. index::
       single: thread safety; Connection

    """

    dispatch: dispatcher[ConnectionEventsTarget]

    class LocalSession namespaces = HasLocalScope.engine.Connection()
```



```
In [1]: from sqlalchemy import create_engine

In [2]: engine = create_engine("mysql+mysqlconnector://root:admin@localhost")

In [3]: with engine.connect() as conn:
...:     print(dir(conn))
...:

['_Connection__can_reconnect', '_Connection__in_begin', '_Connection__savepoint_seq', '__annotations__', '__class__', '__class_getitem__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__orig_bases__', '__parameters__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__subclasshook__', '__weakref__', '_allow_rollback', '_autobegin', '_begin_impl', '_begin_twophase_impl', '_commit_impl', '_commit_twophase_impl', '_cursor_execute', '_dbapi_connection', '_echo', '_exec_insertmany_context', '_exec_single_context', '_execute_clauseelement', '_execute_compiled', '_execute_context', '_execute_ddl', '_execute_default', '_execute_function', '_execution_options', '_get_required_nested_transaction', '_get_required_transaction', '_handle_dbapi_exception', '_handle_dbapi_exception_noconnection', '_has_events', '_invalid_transaction', '_invoke_before_exec_event', '_is_autocommit_isolation', '_is_disconnect', '_is_protocol', '_log_debug', '_log_info', '_message_formatter', '_nested_transaction', '_prepare_twophase_impl', '_reentrant_error', '_release_savepoint_impl', '_revalidate_connection', '_rollback_impl', '_rollback_to_savepoint_impl', '_rollback_twophase_impl', '_run_ddl_visitor', '_safe_close_cursor', '_savepoint_impl', '_schema_translate_map', '_sqla_logger_namespace', '_still_open_and_dbapi_connection_is_valid', '_trans_context_manager', '_transaction', 'begin', 'begin_nested', 'begin_twophase', 'close', 'closed', 'commit', 'commit_prepared', 'connection', 'default_isolation_level', 'exec_driver_sql', 'execute', 'execution_options', 'get_execution_options', 'get_isolation_level', 'get_nested_transaction', 'info', 'invalidate', 'recover_twophase', 'rollback', 'rollback_prep']
```

Klasa Connection

W klasie **Connection** rozpoznajemy dobrze już znane nam metody. Metoda **execute** zgodnie z DB-API jest składową kurSORA, ale tym razem kurSOR jest abstrakcją ukrytą przed użytkownikiem (SQLAlchemy nie jest sterownikiem)



Metoda execute



Metoda **execute** zwraca obiekty klasy *CursorResult*

```
def execute(
    self,
    statement: Executable,
    parameters: Optional[_CoreAnyExecuteParams] = None,
    *,
    execution_options: Optional[CoreExecuteOptionsParameter] = None,
) -> CursorResult[Any]:
    r"""Executes a SQL statement construct and returns a
    :class:`_engine.CursorResult`.

    :param statement: The statement to be executed. This is always
        an object that is in both the :class:`_expression.ClauseElement` and
        :class:`_expression.Executable` hierarchies, including:

        * :class:`_expression.Select`
        * :class:`_expression.Insert`, :class:`_expression.Update`,
        | :class:`_expression.Delete`
        * :class:`_expression.TextClause` and
        | :class:`_expression.TextualSelect`
        * :class:`_schema.DDL` and objects which inherit from
        | :class:`_schema.ExecutableDDLElement`

    :param parameters: parameters which will be bound into the statement.
```



Klasa **CursorResult** reprezentuje wynik zwracany dla danego stanu kurSORA

```
class CursorResult(Result[_T]):  
    """A Result that is representing state from a DBAPI cursor.  
  
    .. versionchanged:: 1.4 The :class:`.CursorResult` class replaces the previous :class:`.ResultProxy` interface. This classes are based on the :class:`.Result` calling API which provides an updated usage model and calling facade for SQLAlchemy Core and SQLAlchemy ORM.  
  
    Returns database rows via the :class:`.Row` class, which provides additional API features and behaviors on top of the raw data returned by the DBAPI. Through the use of filters such as the :meth:`.Result.scalars` method, other kinds of objects may also be returned.  
  
    .. seealso::  
  
        :ref:`tutorial_selecting_data` - introductory material for accessing :class:`_engine.CursorResult` and :class:`.Row` objects.  
  
    """  
  
    __slots__ = (  
        "context",  
        "dialect",  
        "cursor",
```



```
[ '__annotations__', '__class__', '__class_getitem__', '__delattr__', '__dict__', '__dir__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__',
  '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', __iter__, '__le__', '__lt__', '__module__', '__ne__', '__new__', '__next__',
  '__orig_bases__', '__parameters__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__subclasshook__', '_allrows',
  '_assert_no_memoizations', '_attributes', '_column_slices', '_echo', '_fetchall_impl', '_fetchiter_impl', '_fetchmany_impl', '_fetchone_impl', '_generate', '_generate_rows',
  '_getter', '_init_metadata', '_is_cursor', '_is_protocol', '_iter_impl', '_iterator_getter', '_manyrow_getter', '_memoized_keys', '_metadata', '_next_impl',
  '_no_result_metadata', '_onerow_getter', '_only_one_row', '_post_creational_filter', '_raw_all_rows', '_raw_row_iterator', '_real_result', '_reset_memoizations', '_rewind',
  '_row_getter', '_row_logging_fn', '_set_memoized_attribute', '_soft_close', '_soft_closed', '_source_supports_scalars', '_tuple_getter', '_unique_filter_state',
  '_unique_strategy', '_yield_per', 'all', 'close', 'closed', 'columns', 'connection', 'context', 'cursor', 'cursor_strategy', 'dialect', 'fetchall', 'fetchmany', 'fetchone',
  'first', 'freeze', 'inserted_primary_key', 'inserted_primary_key_rows', 'is_insert', 'keys', 'last_inserted_params', 'last_updated_params', 'lastrow_has_defaults',
  'lastrowid', 'mappings', 'memoized_attribute', 'memoized_instancemethod', 'merge', 'one', 'one_or_none', 'partitions', 'postfetch_cols', 'prefetch_cols',
  'returned_defaults', 'returned_defaults_rows', 'returns_rows', 'rowcount', 'scalar', 'scalar_one', 'scalar_one_or_none', 'scalars',
  'splice_vertically', 'supports_sane_multi_rowcount', 'supports_sane_rowcount', 't', 'tuple']
```

Klasa CursorResult

W klasie *CursorResult* rozpoznajemy dobrze już znane nam funkcje.



Funckja text

Wykonywanie surowych zapytań



Opis

```
from sqlalchemy import create_engine, text

cnx_string = "mysql+mysqlconnector://root:admin@localhost"
engine = create_engine(cnx_string)

stmt = text("""CREATE DATABASE IF NOT EXISTS company_db;""")

with engine.connect() as cnx:
    cnx.execute(stmt)
```



Funkcja **text** tworzy obiekty klasy *TextClause*

```
@_document_text_coercion(paramname="text", meth_rst=":func:`.text`, param_rst=":paramref:`.text.text`")  
def text(text: str) -> TextClause:  
    r"""Construct a new :class:`_expression.TextClause` clause,  
    representing  
    a textual SQL string directly.
```

E.g.:::

```
from sqlalchemy import text  
  
t = text("SELECT * FROM users")  
result = connection.execute(t)
```

The advantages `:func:`_expression.text`` provides over a plain string are backend-neutral support for bind parameters, per-statement execution options, as well as bind parameter and result-column typing behavior, allowing SQLAlchemy type constructs to play a role when executing a statement that is specified literally. The construct can also be provided with a ``.c`` collection of column elements, allowing

- 💡 it to be embedded in other SQL expression constructs as a subquery.

Bind parameters are specified by name, using the format ```:name```.



Klasa **TextClause** reprezentuje literal SQL

```
class TextClause(  
    roles.DDLConstraintColumnRole,  
    roles.DDLExpressionRole,  
    roles.StatementOptionRole,  
    roles.WhereHavingRole,  
    roles.OrderByRole,  
    roles.FromClauseRole,  
    roles.SelectStatementRole,  
    roles.InElementRole,  
    Generative,  
    Executable,  
    DQLDMLClauseElement,  
    roles.BinaryElementRole[Any],  
    inspection.Inspectable["TextClause"],  
):  
    """Represent a literal SQL text fragment.
```

E.g.::

```
from sqlalchemy import text  
  
t = text("SELECT * FROM users")  
result = connection.execute(t)
```



Emitowanie wygenerowanych zapytań sql

```
from sqlalchemy import create_engine  
  
cnx_string = "mysql+mysqlconnector://root:admin@localhost"  
engine = create_engine(cnx_string, echo=True)
```

```
C:\Users\User\PycharmProjects\database_programming\.venv\Scripts\python.exe "C:\Users\User\PycharmProjects\database_programming\main.py"  
2024-03-12 06:30:37,938 INFO sqlalchemy.engine.Engine SELECT DATABASE()  
2024-03-12 06:30:37,938 INFO sqlalchemy.engine.Engine [raw sql] {}  
2024-03-12 06:30:37,940 INFO sqlalchemy.engine.Engine SELECT @@sql_mode  
2024-03-12 06:30:37,940 INFO sqlalchemy.engine.Engine [raw sql] {}  
2024-03-12 06:30:37,941 INFO sqlalchemy.engine.Engine SELECT @@lower_case_table_names  
2024-03-12 06:30:37,941 INFO sqlalchemy.engine.Engine [raw sql] {}  
2024-03-12 06:30:37,942 INFO sqlalchemy.engine.Engine BEGIN (implicit)  
2024-03-12 06:30:37,942 INFO sqlalchemy.engine.Engine CREATE DATABASE IF NOT EXISTS company_db;  
2024-03-12 06:30:37,942 INFO sqlalchemy.engine.Engine [generated in 0.00031s] {}  
2024-03-12 06:30:37,943 INFO sqlalchemy.engine.Engine ROLLBACK  
  
Process finished with exit code 0
```



Parametry związane (ang. bounded parameters)

colon format

```
stmt = text("""
INSERT INTO employee(first_name, last_name, position, salary) VALUE
(:fn, :ln, :pos, :sal)
;""")  
employees = [
    {'fn': 'John', 'ln': 'Doe', 'pos': 'Manager', 'sal': 5000.00},
    {'fn': 'Jane', 'ln': 'Smith', 'pos': 'Developer', 'sal': 4000.00},
    {'fn': 'Alice', 'ln': 'Johnson', 'pos': 'HR', 'sal': 4500.00}
]  
  
with engine.connect() as conn:  
    conn.execute(stmt, employees)  
    conn.commit()  
    print("Done.")
```



pyformat (jeden z 5 dopuszczalnych formatów w DB-API, zależy od użytego sterownika)

```
2024-03-12 12:13:50,916 INFO sqlalchemy.engine.Engine INSERT INTO employee (first_name, last_name, position, salary) VALUES (%(first_name_m0)s, %(last_name_m0)s, %(position_m0)s, %(salary_m0)s), %(first_name_m1)s, %(last_name_m1)s, %(position_m1)s, %(salary_m1)s), %(first_name_m2)s, %(last_name_m2)s, %(position_m2)s, %(salary_m2)s)
```

```
2024-03-12 12:13:50,916 INFO sqlalchemy.engine.Engine [no key 0.00033s] {'first_name_m0': 'John', 'last_name_m0': 'Doe', 'position_m0': 'Manager', 'salary_m0': 5000.0, 'first_name_m1': 'Jane', 'last_name_m1': 'Smith', 'position_m1': 'Developer', 'salary_m1': 4000.0, 'first_name_m2': 'Alice', 'last_name_m2': 'Johnson', 'position_m2': 'HR', 'salary_m2': 4500.0}
```

```
2024-03-12 12:13:50,954 INFO sqlalchemy.engine.Engine COMMIT
```



Dwa style



No autocommit

```
stmt = text("""
    INSERT INTO employee(first_name, last_name, position, salary) VALUE
    (:fn, :ln, :pos, :sal)
""")
employees = [
    {'fn': 'John', 'ln': 'Doe', 'pos': 'Manager', 'sal': 5000.00},
    {'fn': 'Jane', 'ln': 'Smith', 'pos': 'Developer', 'sal': 4000.00},
]
employees2 = [
    {'fn': 'Alice', 'ln': 'Johnson', 'pos': 'HR', 'sal': 4500.00}
]
with engine.connect() as conn:
    conn.execute(stmt, employees)
    conn.execute(stmt, employees2)

print("Done.")
```



No autocommit

```
2024-03-12 07:19:42,104 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2024-03-12 07:19:42,104 INFO sqlalchemy.engine.Engine
INSERT INTO employee(first_name, last_name, position, salary) VALUE
(%(fn)s, %(ln)s, %(pos)s, %(sal)s)
;
2024-03-12 07:19:42,104 INFO sqlalchemy.engine.Engine [generated in 0.00034s] [{"fn": "John", "ln": "Doe", "pos": "Manager", "sal": 5000.0}, {"fn": "Alice", "ln": "Johnson", "pos": "HR", "sal": 4500.0}]
2024-03-12 07:19:42,105 INFO sqlalchemy.engine.Engine
INSERT INTO employee(first_name, last_name, position, salary) VALUE
(%(fn)s, %(ln)s, %(pos)s, %(sal)s)
;
2024-03-12 07:19:42,105 INFO sqlalchemy.engine.Engine [generated in 0.00025s] {"fn": "Alice", "ln": "Johnson", "pos": "HR", "sal": 4500.0}
2024-03-12 07:19:42,106 INFO sqlalchemy.engine.Engine ROLLBACK
Done.
```



So ?



1. Commit as you go

```
stmt = text("""
    INSERT INTO employee(first_name, last_name, position, salary) VALUE
    (:fn, :ln, :pos, :sal)
""")  
employees = [
    {'fn': 'John', 'ln': 'Doe', 'pos': 'Manager', 'sal': 5000.00},
    {'fn': 'Jane', 'ln': 'Smith', 'pos': 'Developer', 'sal': 4000.00},
]  
  
employees2 = [
    {'fn': 'Alice', 'ln': 'Johnson', 'pos': 'HR', 'sal': 4500.00}
]  
  
with engine.connect() as conn:
    conn.execute(stmt, employees)
    conn.commit()
    conn.execute(stmt, employees2)
    conn.commit()  
  
print("Done.")
```



1. Commit as you go

```
2024-03-12 07:14:25,794 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2024-03-12 07:14:25,794 INFO sqlalchemy.engine.Engine
INSERT INTO employee(first_name, last_name, position, salary) VALUE
(%(fn)s, %(ln)s, %(pos)s, %(sal)s)
;
2024-03-12 07:14:25,794 INFO sqlalchemy.engine.Engine [generated in 0.00040s] [{"fn': 'John', 'ln': 'Doe', 'pos': 'Manager', 'sal': 5000.0}, {'f
        4000.0}]
2024-03-12 07:14:25,795 INFO sqlalchemy.engine.Engine COMMIT
2024-03-12 07:14:25,798 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2024-03-12 07:14:25,798 INFO sqlalchemy.engine.Engine
INSERT INTO employee(first_name, last_name, position, salary) VALUE
(%(fn)s, %(ln)s, %(pos)s, %(sal)s)
;
2024-03-12 07:14:25,798 INFO sqlalchemy.engine.Engine [generated in 0.00036s] {"fn': 'Alice', 'ln': 'Johnson', 'pos': 'Intern', 'sal': 4500}
2024-03-12 07:14:25,798 INFO sqlalchemy.engine.Engine COMMIT
Done.
```



SQLAlchemy

Or



2. Begin once

```
stmt = text("""
    INSERT INTO employee(first_name, last_name, position, salary) VALUE
    (:fn, :ln, :pos, :sal)
""";)

employees = [
    {'fn': 'John', 'ln': 'Doe', 'pos': 'Manager', 'sal': 5000.00},
    {'fn': 'Jane', 'ln': 'Smith', 'pos': 'Developer', 'sal': 4000.00},
]

employees2 = [
    {'fn': 'Alice', 'ln': 'Johnson', 'pos': 'HR', 'sal': 4500.00}
]

with engine.begin() as conn:
    conn.execute(stmt, employees)
    conn.execute(stmt, employees2)

print("Done.")
```



2. Begin once

```
2024-03-12 07:16:12,514 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2024-03-12 07:16:12,515 INFO sqlalchemy.engine.Engine
INSERT INTO employee(first_name, last_name, position, salary) VALUE
(%(fn)s, %(ln)s, %(pos)s, %(sal)s)
;
2024-03-12 07:16:12,515 INFO sqlalchemy.engine.Engine [generated in 0.00025s] [{"fn': 'John', 'ln': 'Doe', 'pos': 'Manager', 'sal': 5000.0}]
2024-03-12 07:16:12,516 INFO sqlalchemy.engine.Engine
INSERT INTO employee(first_name, last_name, position, salary) VALUE
(%(fn)s, %(ln)s, %(pos)s, %(sal)s)
;
2024-03-12 07:16:12,516 INFO sqlalchemy.engine.Engine [generated in 0.00021s] {'fn': 'Alice', 'ln': 'Johnson', 'pos': 'HR', 'sal': 4500.0}
2024-03-12 07:16:12,516 INFO sqlalchemy.engine.Engine COMMIT
Done.
```



Klasa Row



Iterując po obiekcie klasy **CursorResult** otrzymujemy obiekty klasy **Row**.

```
class Row(BaseRow, Sequence[Any], Generic[_TP]):  
    """Represent a single result row.
```

The `:class:`.Row`` object represents a row of a database result. It is typically associated in the 1.x series of SQLAlchemy with the `:class:`_engine.CursorResult`` object, however is also used by the ORM for tuple-like results as of SQLAlchemy 1.4.

The `:class:`.Row`` object seeks to act as much like a Python named tuple as possible. For mapping (i.e. dictionary) behavior on a row, such as testing for containment of keys, refer to the `:attr:`.Row._mapping`` attribute.

`.. seealso::`

`:ref:`tutorial_selecting_data`` - includes examples of selecting rows from `SELECT` statements.

`.. versionchanged:: 1.4`

Renamed ```RowProxy``` to `:class:`.Row``. `:class:`.Row`` is no longer a `UnboundObject` in that it contains the final form of data within it.



Obiekty klasy **Row** reprezentują pojedynczy rekord result set-a i zachowują się jak krotka nazwana (ang. named tuple)

```
from sqlalchemy import create_engine, select
from tables import employee_table

engine = create_engine('mysql+mysqlconnector://root:admin@localhost/company_db')

stmt = select(employee_table)
with engine.connect() as cnx:
    result = cnx.execute(stmt)
    for id_, first_name, last_name, position, salary in result: # NamedTuple - tuple assignment
        print(f"{first_name} {last_name} ({position}) [{salary}]")
```



Obiekty klasy **Row** reprezentują pojedynczy rekord result set-a i zachowują się jak krotka nazwana (ang. named tuple)

```
from sqlalchemy import create_engine, select
from tables import employee_table

engine = create_engine('mysql+mysqlconnector://root:admin@localhost/company_db')

stmt = select(employee_table)
with engine.connect() as cnx:
    result = cnx.execute(stmt)
    for row in result:
        print(f"{row[1]} {row[2]} ({row[3]}) [{row[4]}]") # NamedTuple - Integer index
```



Obiekty klasy **Row** reprezentują pojedynczy rekord result set-a i zachowują się jak krotka nazwana (ang. named tuple)

```
from sqlalchemy import create_engine, select
from tables import employee_table

engine = create_engine('mysql+mysqlconnector://root:admin@localhost/company_db')

stmt = select(employee_table)
with engine.connect() as cnx:
    result = cnx.execute(stmt)
    for row in result:
        print(f'{row.first_name} {row.last_name} ({row.position}) [{row.salary}]') # NamedTuple - attribute name
```



Obiekty klasy **Row** reprezentują pojedynczy rekord result set-a i zachowują się jak krotka nazwana (ang. named tuple)

```
from sqlalchemy import create_engine, select
from tables import employee_table

engine = create_engine('mysql+mysqlconnector://root:admin@localhost/company_db')

stmt = select(employee_table)
with engine.connect() as cnx:
    result = cnx.execute(stmt)
    for row in result.mappings(): # NamedTuple - mapping access
        print(f'{row.get('first_name')} {row.get('last_name')} ({row.get('position')}) [{row.get('salary')}]')
```



SQLAlchemy Core



SQLAlchemy Expression Language

SQLAlchemy posiada zestaw python-owych narzędzi do komunikacji z bazą danych bez potrzeby bezpośredniego pisania zapytań SQL. **SQLAlchemy Expression Language** (język wyrażeń SQLAlchemy), bo tak jest nazywany ten zestaw, to klasy, metody oraz funkcje, których wywołania mapowane są na odpowiednie instrukcje SQL i tak wygenerowane zapytania SQL są wysyłane do bazy. W ten sposób nie pisząc nawet jednej linijki kodu SQL można wykonać na bazie większości tradycyjnych operacji.



Instrukcje DDL



Struktury bazodanowe

Struktury bazodanowe są reprezentowane w SQLAlchemy Expression Language przez klasy takie jak:

1. Metadata
 2. Table
 3. Column
 4. SQL Datatype Objects (takie jak Integer, String, ...)
- i są nazywane kolektywnie metadynamami bazy danych (ang. database metadata).*



1. Metadata

Klasa **Metadata** jest kolekcją, w której przechowujemy wszystkie informacje na temat istniejących w bazie struktur. Stanowi fasadę na python-owy słownik, którego kluczami są nazwy tabel, a wartościami obiekty reprezentujące te tabele (czyli obiekty klasy Table)



Inicjalizacja obiektu klasy **Metadata**.

```
from sqlalchemy import create_engine
from sqlalchemy import MetaData

cnx_string = "mysql+mysqlconnector://root:admin@localhost/company_db"
engine = create_engine(cnx_string)

metadata = MetaData()
```



1. Metadata

```
class MetaData(HasSchemaAttr):
    """A collection of :class:`_schema.Table` objects and their associated schema constructs.

    Holds a collection of :class:`_schema.Table` objects as well as an optional binding to an :class:`_engine.Engine` or :class:`_engine.Connection`. If bound, the :class:`_schema.Table` objects in the collection and their columns may participate in implicit SQL execution.

    The :class:`_schema.Table` objects themselves are stored in the :attr:`_schema.MetaData.tables` dictionary.

    :class:`_schema.MetaData` is a thread-safe object for read operations. Construction of new tables within a single :class:`_schema.MetaData` object, either explicitly or via reflection, may not be completely thread-safe.

.. seealso::

    :ref:`metadata_describing` - Introduction to database metadata
```



2. Table

Klasa **Table** reprezentuje bazodanową tablicę. Komponuje się z obiektów klasy *Column*.



Inicjalizacja obiektu klasy **Table**.

```
from sqlalchemy import create_engine
from sqlalchemy import MetaData, Table, Column, Integer, String, Float

cnx_string = "mysql+mysqlconnector://root:admin@localhost/company_db"
engine = create_engine(cnx_string)

metadata = MetaData()

employee_table = Table(
    name: 'employee',
    metadata,
    *args: Column(_name_pos: 'id', Integer, primary_key=True),
    Column(_name_pos: 'first_name', String(50)),
    Column(_name_pos: 'last_name', String(100)),
    Column(_name_pos: 'position', String(100)),
    Column(_name_pos: 'salary', Float)
)
```



2. Table

```
class Table(  
    DialectKwargs, HasSchemaAttr, TableClause, inspection.Inspectable["Table"]  
):  
    """Represent a table in a database.  
  
e.g.:  
    mytable = Table(  
        "mytable", metadata,  
        Column('mytable_id', Integer, primary_key=True),  
        Column('value', String(50))  
    )
```

The `:class:`_schema.Table`` object constructs a unique instance of itself based on its name and optional schema name within the given `:class:`_schema.MetaData`` object. Calling the `:class:`_schema.Table`` constructor with the same name and same `:class:`_schema.MetaData`` argument a second time will return the *same* `:class:`_schema.Table`` object - in this way the `:class:`_schema.Table`` constructor acts as a registry function.

.. seealso::

[:ref:`metadata_describing` - Introduction to database metadata](#)

....

```
__visit_name__ = "table"
```



3. Column

Klasa **Column** reprezentuje pojedynczą kolumnę w tabeli.
Komponuje się z obiektów klas reprezentujących typ danych SQL
(takich jak `Integer`, `String`...) oraz obiektów klas reprezentujących
więzy (takich jak `PrimaryKeyConstraint`, `UniqueConstraint`, ...)



Inicjalizacja obiektu klasy **Column**.

```
from sqlalchemy import create_engine
from sqlalchemy import MetaData, Table, Column, Integer, String, Float

cnx_string = "mysql+mysqlconnector://root:admin@localhost/company_db"
engine = create_engine(cnx_string)

metadata = MetaData()

employee_table = Table(
    name: 'employee',
    metadata,
    *args: Column(_name_pos: 'id', Integer, primary_key=True),
    Column(_name_pos: 'first_name', String(50)),
    Column(_name_pos: 'last_name', String(100)),
    Column(_name_pos: 'position', String(100)),
    Column(_name_pos: 'salary', Float)
)
```



3. Column

```
class Column(DialectKwargs, SchemaItem, ColumnClause[_T]):  
    """Represents a column in a database table."""  
  
    __visit_name__ = "column"  
  
    inherit_cache = True  
    key: str  
  
    server_default: Optional[FetchedValue]  
  
    def __init__(  
        self,  
        __name_pos: Optional[  
            Union[str, _TypeEngineArgument[_T], SchemaEventTarget]  
        ] = None,  
        __type_pos: Optional[  
            Union[_TypeEngineArgument[_T], SchemaEventTarget]  
        ] = None,  
        *args: SchemaEventTarget,  
        name: Optional[str] = None,  
        type_: Optional[_TypeEngineArgument[_T]] = None,  
        autoincrement: _AutoIncrementType = "auto",  
        default: Optional[Any] = None,  
        doc: Optional[str] = None,  
        key: Optional[str] = None,  
        index: Optional[bool] = None,  
        unique: Optional[bool] = None,  
        info: Optional[_InfoType] = None,  
        nullable: Optional[  
            Union[bool, Literal[SchemaConst.NULL_UNSPECIFIED]]]
```



4. SQL DataType Objects

SQLAlchemy posiada bogaty zbiór klas reprezentujących poszczególne typy danych w SQL. Wszystkie te klasy dziedziczą po wspólnej nadklasie **TypeEngine**. Hierarchia typów w SQLAlchemy wyróżnia dwie kategorie tych typów:

1. Typy danych "CamelCase"
2. Typy danych "UPPERCASE"



4a CamelCase datatypes

Jest to zbiór najbardziej powszechnych typów danych jakie występują w relacyjnych bazach danych. Obiekty tych klas są rzutowane na odpowiadający im typ w konkretnym RDBMS (np. String w MySQL zostanie zrzutowany na VARCHAR, a w Oracle na NVARCHAR). Należą do nich

BigInteger	Enum	Interval	PickleType	Text	Uuid
Boolean	Double	LargeBinary	SchemaType	Time	
Date	Float	MatchType	SmallInteger	Unicode	
DateTime	Integer	Numeric	String	UnicodeText	



4b UPPERCASE datatypes

W odróżnieniu od typów danych CamelCase jest to zbiór klas reprezentujących konkretny typ danych. Nazwa klasy jest powiązana w jednoznaczny sposób z konkretnym typem SQL. Użycie klasy jest równoznaczne z użyciem typu danych SQL o nazwie tej klasy, niezależnie od tego z jakim RDBMS pracujemy. Możliwa jest sytuacja, w której RDBMS nieobsługuje użytego przez nas typu. W takim przypadku wystąpi błąd. Należą do nich:

ARRAY	BOOLEAN	DATETIME	FLOAT	NCHAR	SMALLINT	UUID
BIGINT	CHAR	DECIMAL	INT	NVARCHAR	TEXT	VARBINARY
BINARY	CLOB	DOUBLE	JSON	NUMERIC	TIME	VARCHAR
BLOB	DATE	DOUBLE_PRECISION	INTEGER	REAL	TIMESTAMP	



```
>>> from sqlalchemy import MetaData  
>>> metadata = MetaData()  
>>> print(dir(metadata))  
['__annotations__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__slots__', '__str__', '__subclasshook__', '__visit_name__', '__weakref__', '_add_table', '_compiler_dispatch', '_fk_memos', '_generate_compiler_dispatch', '_init_items', '_original_compiler_dispatch', '_remove_table', '_schema_item_copy', '_schemas', '_sequences', '_set_parent', '_set_parent_with_dispatch', '_use_schema_map', 'clear', 'create_all', 'create_drop_stringify dialect' 'dispatch',  
'drop_all', 'info', 'naming_convention', 'reflect', 'remove', 'schemas']
```

Metody klasy Metadata

Wysyłanie instrukcji DDL:

- *create_all*
- *drop_all*

