eBook Gratuit

APPRENEZ sqlalchemy

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#sqlalchem

y

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sqlalchemy

It is an unofficial and free sqlalchemy ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official sqlalchemy.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec sqlalchemy

Remarques

LA PHILOSOPHIE DE SQLALCHEMY

À partir du site Web SQLAlchemy :

> Les bases de données SQL se comportent moins comme des collections d'objets, plus la taille et les performances commencent à être importantes. les collections d'objets se comportent moins comme des tableaux et des lignes, plus l'abstraction commence à avoir d'importance. SQLAlchemy vise à intégrer ces deux principes.
>
> SQLAlchemy considère que la base de données est un moteur d'algèbre relationnel, et pas seulement une collection de tables. Les lignes peuvent être sélectionnées non seulement dans les tables, mais également dans les jointures et autres instructions sélectionnées. n'importe laquelle de ces unités peut être composée dans une plus grande structure. Le langage d'expression de SQLAlchemy est basé sur ce concept.
>
> SQLAlchemy est le plus connu pour son objet ORM (Object-Relational Mapper), un composant facultatif qui fournit le modèle de mappeur de données, où les classes peuvent être mappées à la base de données de plusieurs manières. façon proprement découplée depuis le début.
>
> L'approche globale de SQLAlchemy face à ces problèmes est totalement différente de celle de la plupart des autres outils SQL / ORM, enracinée dans une approche dite de complémentarité; Au lieu de cacher les détails relationnels SQL et objet derrière un mur d'automatisation, tous les processus sont entièrement exposés dans une série d'outils composables et transparents. La bibliothèque prend en charge l'automatisation des tâches redondantes, tandis que le développeur garde le contrôle de l'organisation de la base de données et de la manière dont SQL est construit.
>
> L'objectif principal de SQLAlchemy est de changer votre façon de penser sur les bases de données et SQL!

Versions

Version	Statut de la version	Changer le journal	Date de sortie
1.1	Bêta	1.1	2016-07-26
1.0	Version actuelle	1.0	2015-04-16
0,9	Entretien	0,9	2013-12-30
0,8	Sécurité	0,8	2013-03-09

Examples

Installation ou configuration

```
pip install sqlalchemy
```

Pour la plupart des applications courantes, en particulier les applications Web, il est généralement recommandé aux débutants d'utiliser une bibliothèque supplémentaire, telle que `flask-sqlalchemy` .

```
pip install flask-sqlalchemy
```

Bonjour le monde! (SQLAlchemy Core)

Cet exemple montre comment créer une table, insérer des données et sélectionner dans la base de données à l'aide de SQLAlchemy Core. Pour plus d'informations sur l' ORM SQLAlchemy, voir ici .

Tout d'abord, nous devons nous connecter à notre base de données.

```
from sqlalchemy import create_engine

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

> Le moteur est le point de départ de toute application SQLAlchemy. C'est une base de données pour la base de données réelle et son interface DBAPI, fournie à une application SQLAlchemy via un pool de connexions et un dialecte, qui décrit comment communiquer avec un type spécifique de combinaison base de données / DBAPI. Le moteur référence à la fois un dialecte et un pool de connexions, qui interprètent ensemble les fonctions du module DBAPI ainsi que le comportement de la base de données.

Après avoir créé notre moteur, nous devons définir et créer nos tables .

```
from sqlalchemy import Column, Integer, Text, MetaData, Table

metadata = MetaData()
messages = Table(
    'messages', metadata,
    Column('id', Integer, primary_key=True),
    Column('message', Text),
)

messages.create(bind=engine)
```

Pour expliquer davantage l'objet MetaData, voir ci-dessous les documents suivants:

> Une collection d'objets Table et leurs objets enfants associés sont appelés métadonnées de base de données.

> Nous définissons nos tables dans un catalogue appelé MetaData, en utilisant la construction Table, qui ressemble aux instructions SQL CREATE TABLE standard.

Maintenant que nos tables sont définies et créées, nous pouvons commencer à insérer des données! L'insertion implique deux étapes. Composer la construction d' insertion et exécuter la requête finale.

```
insert_message = messages.insert().values(message='Hello, World!')
engine.execute(insert_message)
```

Maintenant que nous avons des données, nous pouvons utiliser la fonction select pour interroger nos données. Les objets de colonne sont disponibles en tant qu'attributs nommés de l'attribut c sur l'objet Table, ce qui facilite la sélection directe des colonnes. L'exécution de cette instruction select renvoie un objet `ResultProxy` qui a accès à quelques méthodes, fetchone () , fetchall () et fetchmany () , qui renvoient toutes un nombre de lignes de base de données interrogées dans notre instruction select.

```
from sqlalchemy import select
stmt = select([messages.c.message])
message, = engine.execute(stmt).fetchone()
print(message)
```

```
Hello, World!
```

Et c'est tout! Voir le didacticiel SQLAlchemy SQL Expressions pour plus d'exemples et d'informations.

Bonjour le monde! (SQLAlchemy ORM)

Cet exemple montre comment créer une table, insérer des données et sélectionner dans la base de données à l'aide de l' **ORM SQLAlchemy** . Pour plus d'informations sur SQLAlchemy Core, voir ici .

Tout d'abord, nous devons nous connecter à notre base de données, qui est identique à celle que nous utiliserions avec SQLAlchemy Core (Core).

```
from sqlalchemy import create_engine

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

Après avoir connecté et créé notre moteur, nous devons définir et créer nos tables. C'est là que le langage SQLAlchemy ORM commence à différer grandement de Core. Dans ORM, le processus de création et de définition de la table commence par définir les tables et les classes que nous utiliserons pour les mapper à ces tables. Ce processus se fait en une seule étape dans ORM, que SQLAlchemy appelle le système déclaratif .

```
from sqlalchemy.ext.declarative import declarative_base
```

```
Base = declarative_base()
```

Maintenant que notre mappeur de base est déclaré, nous pouvons en sous-classer pour construire nos mappages ou `models` déclaratifs .

```
from sqlalchemy import Column, Integer, String

class Message(Base):
    __tablename__ = 'messages'

    id = Column(Integer, primary_key=True)
    message = Column(String)
```

En utilisant la classe de base déclarative, nous finissons par créer un objet `Table` et `Mapper` . De la documentation:

> L'objet Table est membre d'une plus grande collection appelée MetaData. Lorsque vous utilisez Declarative, cet objet est disponible en utilisant l'attribut .metadata de notre classe de base déclarative.

Dans cette optique, pour créer toutes les tables qui n'existent pas encore, nous pouvons appeler la commande ci-dessous, qui utilise le registre MetaData de SQLAlchemy Core.

```
Base.metadata.create_all(engine)
```

Maintenant que nos tables sont mappées et créées, nous pouvons insérer des données! L'insertion se fait par la création d'instances de mappage .

```
message = Message(message="Hello World!")
message.message # 'Hello World!
```

À ce stade, tout ce que nous avons est une instance de message au niveau du niveau d'abstraction ORM, mais rien n'a encore été enregistré dans la base de données. Pour ce faire, nous devons d'abord créer une session .

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()
```

Cet objet de session est notre gestionnaire de base de données. Selon les documents SQLAlchemy:

> Il récupère une connexion à partir d'un pool de connexions gérées par le moteur et le conserve jusqu'à ce que nous validions toutes les modifications et / ou fermions l'objet de session.

Maintenant que nous avons notre session, nous pouvons ajouter notre nouveau message à la session et valider nos modifications dans la base de données.

```
session.add(message)
session.commit()
```

Maintenant que nous avons des données, nous pouvons tirer parti du langage de requête ORM pour extraire nos données.

```
query = session.query(Message)
instance = query.first()
print (instance.message) # Hello World!
```

Mais c'est juste le début! Il y a beaucoup plus de fonctionnalités qui peuvent être utilisées pour composer des requêtes, comme `filter` , `order_by` et bien plus encore. Voir le tutoriel SQLAlchemy ORM pour plus d'exemples et d'informations.

Lire Démarrer avec sqlalchemy en ligne: https://riptutorial.com/fr/sqlalchemy/topic/1697/demarrer-avec-sqlalchemy

Chapitre 2: De liaison

Examples

Moteur

Le moteur est utilisé pour se connecter à différentes bases de données en utilisant une URL de connexion:

```
from sqlalchemy import create_engine

engine = create_engine('postgresql://user:pass@localhost/test')
```

Notez, cependant, que le moteur n'établit pas réellement de connexion tant qu'il n'est pas utilisé pour la première fois.

Le moteur crée automatiquement un pool de connexions, mais ouvre de nouvelles connexions paresseusement (c.-à-d. Que SQLAlchemy n'ouvrira pas 5 connexions si vous n'en demandez qu'une).

Utiliser une connexion

Vous pouvez ouvrir une connexion (c'est-à-dire en demander une depuis le pool) en utilisant un gestionnaire de contexte:

```
with engine.connect() as conn:
    result = conn.execute('SELECT price FROM products')
    for row in result:
        print('Price:', row['price'])
```

Ou sans, mais il faut le fermer manuellement:

```
conn = engine.connect()
result = conn.execute('SELECT price FROM products')
for row in result:
    print('Price:', row['price'])
conn.close()
```

Exécution implicite

Si vous souhaitez uniquement exécuter une seule instruction, vous pouvez utiliser le moteur directement et cela ouvrira et fermera la connexion pour vous:

```
result = engine.execute('SELECT price FROM products')
for row in result:
    print('Price:', row['price'])
```

Transactions

Vous pouvez utiliser `engine.begin` pour ouvrir une connexion et lancer une transaction qui sera `engine.begin` si une exception est `engine.begin` ou `engine.begin` autrement. C'est une manière implicite d'utiliser une transaction, car vous n'avez pas la possibilité de revenir manuellement.

```
with engine.begin() as conn:
    conn.execute(products.insert(), price=15)
```

Plus explicitement, vous pouvez commencer une transaction en utilisant une connexion:

```
with conn.begin() as trans:
    conn.execute(products.insert(), price=15)
```

Notez que nous appelons toujours `execute` sur la connexion. Comme précédemment, cette transaction sera `trans.rollback()` si une exception est déclenchée, mais nous avons également accès à la transaction, ce qui nous permet de `trans.rollback()` manuellement à l'aide de `trans.rollback()` .

Cela pourrait être fait plus explicitement comme ceci:

```
trans = conn.begin()
try:
    conn.execute(products.insert(), price=15)
    trans.commit()
except:
    trans.rollback()
    raise
```

Lire De liaison en ligne: https://riptutorial.com/fr/sqlalchemy/topic/2025/de-liaison

Chapitre 3: Flask-SQLAlchemy

Remarques

Flask-SQLAlchemy ajoute des fonctionnalités supplémentaires telles que la destruction automatique de la session en supposant que certaines choses ne sont pas ce dont vous avez besoin.

Examples

Une application minimale

Pour le cas courant d'une application Flask, il suffit de créer votre application Flask, de charger la configuration de votre choix, puis de créer l'objet SQLAlchemy en lui transmettant l'application.

Une fois créé, cet objet contient alors toutes les fonctions et aides de sqlalchemy et sqlalchemy.orm. De plus, il fournit une classe appelée Model qui est une base déclarative pouvant être utilisée pour déclarer des modèles:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:////tmp/test.db'
db = SQLAlchemy(app)


class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)

    def __init__(self, username, email):
        self.username = username
        self.email = email

    def __repr__(self):
        return '<User %r>' % self.username
```

Lire Flask-SQLAlchemy en ligne: https://riptutorial.com/fr/sqlalchemy/topic/4601/flask-sqlalchemy

Chapitre 4: L'ORM

Remarques

Le SQLAlchemy ORM est basé sur SQLAlchemy Core . Par exemple, bien que les classes de modèle utilisent des objets `Column` , elles font partie du noyau et une documentation plus pertinente y sera trouvée.

Les parties principales de l'ORM sont les classes de session , de requête et mappées (en utilisant généralement l'extension déclarative dans SQLAlchemy moderne).

Examples

Conversion d'un résultat de requête en dict

Tout d'abord la configuration de l'exemple:

```
import datetime as dt
from sqlalchemy import Column, Date, Integer, Text, create_engine, inspect
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
Session = sessionmaker()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)
    birthday = Column(Date)

engine = create_engine('sqlite://')
Base.metadata.create_all(bind=engine)
Session.configure(bind=engine)

session = Session()
session.add(User(name='Alice', birthday=dt.date(1990, 1, 1)))
session.commit()
```

Si vous interrogez des colonnes individuellement, la ligne est un `KeyedTuple` qui a une méthode `_asdict` . Le nom de la méthode commence par un seul trait de soulignement, correspondant à l'API `namedtuple` (ce n'est pas privé!).

```
query = session.query(User.name, User.birthday)
for row in query:
    print(row._asdict())
```

Lorsque vous utilisez l'ORM pour récupérer des objets, cela n'est pas disponible par défaut. Le système d'inspection SQLAlchemy doit être utilisé.

```
def object_as_dict(obj):
    return {c.key: getattr(obj, c.key)
            for c in inspect(obj).mapper.column_attrs}

query = session.query(User)
for user in query:
    print(object_as_dict(user))
```

Ici, nous avons créé une fonction pour effectuer la conversion, mais une option serait d'ajouter une méthode à la classe de base.

Au lieu d'utiliser `declarative_base` comme ci-dessus, vous pouvez le créer à partir de votre propre classe:

```
from sqlalchemy.ext.declarative import as_declarative

@as_declarative()
class Base:
    def _asdict(self):
        return {c.key: getattr(self, c.key)
                for c in inspect(self).mapper.column_attrs}
```

Filtration

Vu le modèle suivant

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

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)
    birthday = Column(Date)
```

Vous pouvez filtrer les colonnes dans la requête:

```
import datetime as dt
session.query(User).filter(User.name == 'Bob')
session.query(User).filter(User.birthday < dt.date(2000, 1, 1))
```

Pour le premier cas, il existe un raccourci:

```
session.query(User).filter_by(name='Bob')
```

Les filtres peuvent être composés en utilisant une relation AND en enchaînant la méthode de `filter` :

```
(session.query(User).filter(User.name.like('B%'))
                    .filter(User.birthday < dt.date(2000, 1, 1)))
```

Ou de manière plus flexible, en utilisant les opérateurs binaires surchargés `&` et `|` :

```
session.query(User).filter((User.name == 'Bob') | (User.name == 'George'))
```

N'oubliez pas les parenthèses internes pour gérer la priorité des opérateurs.

Commandé par

Étant donné un modèle de base:

```
class SpreadsheetCells(Base):
    __tablename__ = 'spreadsheet_cells'

    id = Column(Integer, primary_key=True)
    y_index = Column(Integer)
    x_index = Column(Integer)
```

Vous pouvez récupérer une liste ordonnée en chaînant la méthode `order_by` .

```
query = session.query(SpreadsheetCells).order_by(SpreadsheetCells.y_index)
```

Cela pourrait être enchaîné après un `filter` ,

```
query = session.query(...).filter(...).order_by(...)
```

ou pour composer davantage une requête existante.

```
query = session.query(...).filter(...)
ordered_query = query.order_by(...)
```

Vous pouvez également déterminer le sens du tri de deux manières:

1. Accéder aux propriétés du champ `asc` et `dsc` :

```
query.order_by(SpreadsheetCells.y_index.desc()) # desc
query.order_by(SpreadsheetCells.y_index.asc()) # asc
```

2. En utilisant les fonctions de module asc et desc:

```
from sqlalchemy import asc, desc

query.order_by(desc(SpreadsheetCells.y_index)) # desc
query.order_by(asc(SpreadsheetCells.y_index)) # asc
```

Accéder aux résultats de la requête

Une fois que vous avez une requête, vous pouvez en faire plus que d'itérer les résultats dans une boucle for.

Installer:

```
from datetime import date

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(Text, nullable=False)
    birthday = Column(Date)

# Find users who are older than a cutoff.
query = session.query(User).filter(User.birthday < date(1995, 3, 3))
```

Pour retourner les résultats sous forme de liste, utilisez `all()` :

```
reslist = query.all() # all results loaded in memory
nrows = len(reslist)
```

Vous pouvez obtenir un compte avec `count()` :

```
nrows = query.count()
```

Pour obtenir uniquement le premier résultat, utilisez `first()` . Ceci est très utile en combinaison avec `order_by()` .

```
oldest_user = query.order_by(User.birthday).first()
```

Pour les requêtes qui ne doivent renvoyer qu'une seule ligne, utilisez-en `one()` :

```
bob = session.query(User).filter(User.name == 'Bob').one()
```

Cela soulève une exception si la requête renvoie plusieurs lignes ou si elle n'en renvoie aucune. Si la ligne n'existe pas encore, utilisez `one_or_none()` :

```
bob = session.query(User).filter(User.name == 'Bob').one_or_none()
if bob is None:
    create_bob()
```

Cela soulèvera toujours une exception si plusieurs lignes portent le nom 'Bob'.

Lire L'ORM en ligne: https://riptutorial.com/fr/sqlalchemy/topic/2020/l-orm

Chapitre 5: La session

Remarques

Une session garde une trace des objets ORM et de leurs modifications, gère les transactions et permet d'effectuer des requêtes .

Examples

Créer une session

Une session est généralement obtenue à l'aide de `sessionmaker` , qui crée une classe de `Session` propre à votre application. Le plus souvent, la classe `Session` est liée à un moteur, ce qui permet aux instances d'utiliser le moteur implicitement.

```
from sqlalchemy.orm import sessionmaker

# Initial configuration arguments
Session = sessionmaker(bind=engine)
```

Le `engine` et la `Session` ne doivent être créés qu'une seule fois.

Une session est une instance de la classe que nous avons créée:

```
# This session is bound to provided engine
session = Session()
```

`Session.configure()` peut être utilisé pour configurer la classe ultérieurement, par exemple le démarrage de l'application plutôt que le temps d'importation.

```
Session = sessionmaker()

# later
Session.configure(bind=engine)
```

Les arguments transmis à `Session` remplacent directement les arguments transmis à `sessionmaker` .

```
session_bound_to_engine2 = Session(bind=engine2)
```

Ajout d'instances

Les objets nouveaux ou détachés peuvent être ajoutés à la session en utilisant `add()` :

```
session.add(obj)
```

Une séquence d'objets peut être ajoutée en utilisant `add_all()` :

```
session.add_all([obj1, obj2, obj3])
```

Un INSERT sera émis dans la base de données lors du prochain vidage, ce qui se produit automatiquement. Les modifications sont conservées lors de la validation de la session.

Suppression d'instances

Pour supprimer des objets persistants, utilisez `delete()` :

```
session.delete(obj)
```

La suppression effective de la base de données se produira lors du prochain vidage .

Lire La session en ligne: https://riptutorial.com/fr/sqlalchemy/topic/2258/la-session

Chapitre 6: SQLAlchemy Core

Examples

Conversion du résultat en dict

Dans SQLAlchemy core, le résultat est `RowProxy` . Dans les cas où vous voulez un dictionnaire explicite, vous pouvez appeler `dict(row)` .

Tout d'abord la configuration de l'exemple:

```
import datetime as dt
from sqlalchemy import (
    Column, Date, Integer, MetaData, Table, Text, create_engine, select)

metadata = MetaData()
users = Table(
    'users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', Text, nullable=False),
    Column('birthday', Date),
)

engine = create_engine('sqlite://')
metadata.create_all(bind=engine)

engine.execute(users.insert(), name='Alice', birthday=dt.date(1990, 1, 1))
```

Ensuite, pour créer un dictionnaire à partir d'une ligne de résultat:

```
with engine.connect() as conn:
    result = conn.execute(users.select())
    for row in result:
        print(dict(row))

    result = conn.execute(select([users.c.name, users.c.birthday]))
    for row in result:
        print(dict(row))
```

Lire SQLAlchemy Core en ligne: https://riptutorial.com/fr/sqlalchemy/topic/2022/sqlalchemy-core

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec sqlalchemy	adarsh, Community, Mattew Whitt, RazerM, Stephen Fuhry
2	De liaison	RazerM
3	Flask-SQLAlchemy	Ilya Rusin
4	L'ORM	Ilja Everilä, Mattew Whitt, RazerM, Tom Hunt
5	La session	Ilja Everilä, RazerM
6	SQLAlchemy Core	RazerM