Introdução a SQLAlchemy

Conceitos ORM e utilização da biblioteca SQLAlchemy

O que é um banco de dados?

- Nós conseguimos inserir e obter dados.
- Dados são gravados em tabelas.
- Estas tabelas são formadas por linhas e colunas.
- As colunas são chamadas de campos; as linhas de registros.
- Expressões nos permitem obter registros no banco de dados.

O que é um banco de dados relacional?

- A unidade (objeto) mais importante é a coluna.
- Colunas formam registros e tabelas.
- Linhas podem ser obtidos através de duas ou mais tabelas usando “joins”.
- Linhas podem ser formadas em “tabelas derivadas” usando subqueries (subconsultas, views, tabelas de índice, etc).
- Funções de agregação, agrupamento, funções, triggers (gatilhos), stored procedures, etc.
- O objeto mais importante é a tabela.
- Transações.

Como estabelecer comunicação com banco de dados?

- APIs de banco de dados.
- Layers de abstração (criados para facilitar a comunicação entre banco de dados e aplicação).
- Mapeadores de objetos relacionais (ORM).

O que é um ORM?

- Automatiza a persistência de modelos de domínio nos esquemas relacionais.
- Suporta “queries” e geração de SQL (structured query language).
- Intermédio entre orientação a objetos e objetos relacionais (relacionamentos entre classes e objetos do banco de dados).

Quanta abstração um ORM deveria providenciar?

- Demonstrar como os dados são guardados e acessados.
- Demonstrar que a base de dados é relacional, possuindo suporte ao relacionamento entre campos e tabelas, tais como registros (joins).
- Abstrair não significa esconder. Assim como qualquer outra API, um dos objetivos do SQLAlchemy é oferecer um melhor suporte de comunicação entre conceitos e possibilidades de um software específico.

Então ORM significa esconder a sujeira do SQL embaixo do tapete?

- Significa um intermédio entre SQL e o banco de dados relacional.
- Além de SQL, oferece suporte direto a API do banco de dados em uso.
- O que é SQL? SQL é uma linguagem relacional partindo do ponto que ela possui suporte para lidar com objetos tais como tabelas, views, stored procedures, functions, types, etc.
- SQLAlchemy oferece um meio de controlar o esquema e o modelo de objeto, interagindo com a base de dados.

O que é SQLAlchemy?

- SQLAlchemy é uma biblioteca de acesso a dados.
- Suporta comunicação entre diversos bancos de dados, tais como PostgreSQL, MySQL, Oracle.
- Fornece suporte a funções específicas de cada um dos bancos de dados aos quais oferece suporte.

Como instalar o SQLAlchemy?

- Executando o script setup.py que pode ser encontrado dentro da versão compactada e baixada do SQLAlchemy.
- Usando setuptools (easy_install suportado por python 2.x).
- Usando distribute (easy_install suportado por python 3.x).
- Usando pip.
- easy_install SQLAlchemy ou pip install SQLAlchemy.

ORM Tutorial

- Os exemplos desse tutorial podem ser encontrados no diretório “exemplos” no mesmo nível que a apresentação está localizada.

Conferindo a instalação

- Verificar se o import da biblioteca não retorna a exceção ImportError.
- Para verificar a versão instalada, executar a seguinte linha no console:

```
>>> import sqlalchemy

>>> sqlalchemy.__version__

'0.8.0'
```

Conectando

- Este tutorial visa o aprendizado. Para tal usaremos SQLite.
- Não serão criados arquivoas, mas sim, utilizaremos o SQLite em memória (sqlite:///:memory:)
  >>> from sqlalchemy import create_engine
  >>> engine = create_engine('**sqlite:///:memory:**', echo=True)
- A função create_engine cria um objeto da classe Engine que cria uma interface de comunicação entre a DBAPI (Database Application Programming Interface) e o banco de dados que está sendo usado.

Declarando um mapeamento

- Para de fato ter um ORM estabelecido, é necessário representar as tabelas que serão usadas.
- Cada classe declarada no código mapeará a tabela alvo no banco de dados.
- O mapeamento se utilizada do módulo Declarative que engloba as classes e tabelas relativas a base.

```
>>> from sqlalchemy.ext.declarative import
declarative_base
>>> Base = declarative_base()
```

Mapeando as tabelas

- Agora que temos a base definida e a conexão, podemos mapear as tabelas que guardarão os dados da nossa aplicação.
- Para isso começaremos com a criação da tabela Users:

```
>>> from sqlalchemy import Column, Integer, String
>>> class User(Base):
...     __tablename__ = 'users'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     fullname = Column(String)
...     password = Column(String)
...     def __init__(self, name, fullname, password):
...             self.name = name
...             self.fullname = fullname
...             self.password = password
...     def __repr__(self):
...             return "<User('%s', '%s', '%s')>" % (self.name, self.fullname, self.password)
```

Criando as tabelas no banco de dados

- Por enquanto, as tabelas apenas estão mapeadas. Elas ainda precisam ser criadas.
- A classe Base possui um objeto chamado metadata, cujo qual possui um método para criar as tabelas chamado create_all.
- Para este método deve ser passar nossa Engine como fonte de conexão.
  >>> Base.metadata.create_all(engine)
- Após a chamada deste método o **DDL** das tabelas será gerado na tela com a ajuda do módulo logging, mas apenas se o parâmetro echo da chamada do método create_engine estiver setado para True.

Algumas diferenças entre bancos

- Alguns bancos de dados necessitam que o tamanho de um campo string seja especificado, ex: Column(String(100)). Precisão de campos numéricos também não são atribuídas pela biblioteca.
- Bancos como Oracle não possuem atributos como campos autoincrementais, para isso seria necessário atribuir uma sequence ao campo:
  >>> id = Column(Integer, Sequence('seq_user'), primary_key=True)
- SQLAlchemy não possui mecanismos para identificar se o banco de dados em uso utiliza campos autoincrementais ou não, assim como não possuem mecanismos para fazer decisões pelo programador, o que significa que é necessário codificar estas parametrizações.

Inserindo dados através de mapeamentos

- Precisamos definir os valores dos campos a serem inseridos.
- Para isso, criamos uma instância da classe User chamada fulano.
  >>> fulano = User('fulano', 'Fulano da Silva', 'fulano123').
- Lembra do método (descriptor) __repr__ da classe User?
  Veja ele em ação:
  >>> fulano
  <User('fulano', 'Fulano da Silva', 'fulano123')>
- Nenhum dado ainda foi inserido. Os valores atribuídos aos atributos da classe User ainda não estão persistentes no banco de dados.

Sessão

- Para que possamos inserir o objeto fulano na base de dados SQLite que criamos, precisamos iniciar uma sessão.
  >>> from sqlalchemy.orm import sessionmaker
  >>> Session = sessionmaker(bind=engine)
  >>> Session
  sessionmaker(class_='Session'autoflush=True,
  bind=Engine(**sqlite:///:memory:**), autocommit=False, expire_on_commit=True)
- Sessões estabelecem todas as comunicações entre os objetos locais e a base de dados, representando uma área de trabalho que guarda todos estes objetos.
- Para adicionar um objeto na nossa sessão, precisamos instanciar um objeto para a classe Session, a mesma que passamos nossa engine como parâmetro.
  >>> session = Session()

Adicionando objetos na Sessão

- O método add está disponível para adicionarmos objetos na sessão.
  >>> session.add(fulano)
- Caso você já esteja familiarizado com o método __dict__, é possível ver que o objeto fulano já se encontra na sessão (alguns dados do retorno do método foram omitidos para melhor visualização):
  >>> session.__dict__
  <User('fulano', 'Fulano da Silva', 'fulano123')>}, 'bind': Engine(**sqlite:///:memory:**), '_deleted': {}, '_flushing': False, 'identity_map': {}, 'dispatch': <sqlalchemy.event.SessionEventsDispatch object at 0x3b25dd0>, '_enable_transaction_accounting': True
- Agora o objeto fulano está inserido no banco de dados. Podemos selecionar este registro na base de dados efetuando uma query sobre esta sessão.

Utilizando objetos intermediados pela sessão

- O objeto session tem um método chamado query. Ele é responsável por carregar instâncias de objetos do banco de dados.
  >>> usuario = session.query(User).filter_by(name='fulano').first()
- Para verificar como o objeto fulano é o mesmo que usuario:
  >>> fulano is usuario
  True
- Podemos modificar o nome do usuário fulano.
  >>> fulano.name = 'fulaninho'
  Após modificar o nome, o registro ainda não foi atualizado.
  >>> session.dirty
  IdentitySet([<User('fulaninho', 'Fulano da Silva', 'fulano123')>])
  session.dirty retorna uma representação dos objetos que estão modificados mas não foram, de fato, atualizados na base de dados.

Utilizando objetos intermediados pela sessão

- Podemos adicionar mais usuarios na tabela Users
  >>> session.add_all([User('ciclano', 'Ciclano da Silva', 'ciclano123'), User('beltrano', 'Beltrano da Silva', 'beltrano123')])
- Podemos verificar a existência destes registros como novos
  >>> session.new
  IdentitySet([<User('beltrano', 'Beltrano da Silva', 'beltrano123')>, <User('ciclano', 'Ciclano da Silva', 'ciclano123')>])
  Para inserir estes registros nós usamos o método commit.
  >>> session.commit()
- O método **commit** é a certeza de que os dados serão salvos na base de dados. Após o método commit, session.dirty e session.new não apresentam mais objetos de anteriormente, pois eles são deletados da área de trabalho da sessão.

Desfazendo transações

- Se você errou um update, delete ou insert, você pode desfazer a transação utilizando o método rollback.
  >>> session.query(User).filter_by(name='ciclano').delete()
  Esta operação deleta o usuário apenas na sessão atual em que o interpretador do Python está reconhecendo.
- Para desfazer este delete, o método **rollback** pode ser usado.
  >>> session.rollback()
- Ao consultar o usuário de atributo name com o valor 'ciclano', ele voltará a ser representado na seguinte consulta dos usuários:
  >>> session.query(User).all()

Query - obtendo resultados

- O objeto Query aceita critérios de seleção. Dentre os critérios de seleção podemos citar filtros como equal, not equals, in, not in, like, is null, is not null e os famosos and e or.
- Para selecionar o usuário de nome beltrano na tabela Users:
  >>> session.query(User).filter(User.name=='beltrano')
  O resultado é um iterável (__iter__), mas também pode se obter o retorno em tipo de lista agregando ao final dos critérios os métodos first(), one() and all().
  >>> usuarios = session.query(User).all()
  >>> usuarios = session.query(User).first()
  >>> usuarios = session.query(User).one()
- Caso você queria contar o total de registros de uma query, utilize **count**.
  >>> session.query(User).filter(User.name != None).count()

“Hardcoded” SQL Strings

- Para aqueles que gostam de ver a query sendo gerada ou apenas querem testar uma query, o objeto Query também aceita literais:
  ```
  >>> for user in session.query(User).filter("name='ciclano'").all():
      print(user)
  ```
- Porém, o modo acima não é o mais seguro pois como pode ser visto, o log da query assegura que nenhum parâmetro passou pelo método bind() que garante que SQL Injection não seja usado no seu programa.
  ```
  >>> session.query(User).filter('name=:name').params(name='ciclano').one()
  ```
  Este método funciona como uma troca dos valores filtrados :bind (todos os valores com “dois pontos”) pelo valor passado no método params().

Relacionamentos

- Por enquanto temos apenas uma tabela: Users (usuários). E se quisermos guardar os e-mails dos nossos usuários? Precisamos criar uma tabela chamada Addresses (Endereços).
- Para que o mapeamento de relações entre chaves seja possível, precisamos importar algumas funcionalidades:
  ```
  >>> from sqlalchemy import ForeignKey
  >>> from sqlalchemy.orm import relationship, backref
  ```

Relacionamentos

- Agora temos que mapear a classe Address:

```
>>> class Address(Base):
...     __tablename__ = 'addresses'
...     id = Column(Integer, primary_key=True)
...     email_address = Column(String, nullable=False)
...     user_id = Column(Integer, ForeignKey('users.id'))
...     user = relationship("User",
backref=backref('addresses', order_by=id))
...     def __init__(self, email_address):
...             self.email_address = email_address
...     def __repr__(self):
...             return "<Address('%s')>" % self.email_address
```

Relacionamentos

- ForeignKey faz a conexão do campo user_id da tabela Address com a chave primária id da tabela Users (users.id = address.user_id).
- relationship() conecta Address a classe User, determinando que user será um relacionamento muitos-para-um (muitos endereços para um usuário).
- backref() conecta a classe User ao objeto Addresses. Isso signifca que a conexão é feita de moda inversa, criando um relacionamento um-para-muitos (um usuário para muitos endereços).
- Agora precisamos criar a tabela.
  >>> Base.metadata.create_all(engine)
  A saída do comando mostra que a tabela Users não foi modificada e que a tabela Addresses foi criada.

Manipulando relacionamentos

- Criaremos um novo usuário para criar/editar relacionamentos.
  >>> chico = User('chico', 'Francisco da Silva', 'chico123')
  >>> chico.addresses
  []
- O usuário “chico” não tem nenhum endereço associado.
- Podemos fazer isso diretamente na propriedade addresses do objeto user.
  >>> chico.addresses = [Address(email_address='chico123@gmail.com'), Address(email_address='franciscochico123@gmail.com')]
  >>> chico.addresses
- [<Address('chico123@gmail.com')>, <Address('franciscochico123@gmail.com')>]

Manipulando relacionamentos

- É necessário adicionar estes dados na sessão e realizar o commit.
  >>> session.add(chico)
  >>> session.commit()
- Podemos verificar a existência do usuário “chico” na base.
  >>> chico = session.query(User).filter_by(name='chico').one()
  >>> chico
  >>> <User('chico', 'Francisco da Silva', 'chico123')>
- Podemos verificar os endereços atribuídos ao chico, também.
  >>> chico.addresses
- Este é um efeito chamado de **lazy loading** que carrega coleções de objetos referenciados por relacionamentos.

Dúvidas?

Py[Tchê]

Junte-se ao PyTchê!

Acesse http://pytche.org/.