Федеральное государственное образовательное бюджетное учреждение высшего образования

Финансовый университет при Правительстве Российской Федерации

(Финансовый университет)

Дисциплина «Программирование на языках Python и SQL»

</Pre>

5 февраля 2021 года
Семинар
ПИ19-3, ПИ19-4 - 3 подгруппа

Тема 1. SQLAlchemy и язык выражений SQL

SQL Expression Language

Введение

SQLAlchemy и язык выражений SQL

SQLAlchemy - библиотека Пайтон, которая устраняет разрыв между реляционными базами данных и традиционным программированием. Хотя SQLAlchemy позволяет «опуститься» до необработанного SQL для выполнения запросов, она поощряет мышление более высокого уровня за счет более «питонического» и дружественного подхода к запросам и обновлению базы данных. SQLAlchemy используется для взаимодействия с широким спектром баз данных. Она позволяет создавать модели данных и запросы в манере, напоминающей обычные классы и операторы Python.

Язык выражений SQL (SQL Expression Language), называемый также Кор (Core, ядро) - это инструмент SQLAlchemy для представления общих операторов и выражений SQL в стиде Пайтон. Он ориентирован на фактическую схему базы данных и стандартизирован таким образом, что обеспечивает единообразный язык для большого числа серверных баз данных.

SQLAlchemy Core имеет представление, ориентированное на схему, таблицы, ключи и индексы, как и традиционный SQL. SQLAlchemy Core эффективен в отчетах, анализе и других применениях, где полезно иметь возможность жестко контролировать запрос или работать с немоделированными данными. Надежный пул соединений с базой данных и оптимизация набора результатов идеально подходят для работы с большими объемами данных.

Кодирование

Работать с SQLAlchemy удобно в интерактивной среде "чтение-оценка-вывод" (REPL), в такой, как интерактивный ноутбук ipython http://ipython.org/ (http://ipython.org/)

Установка Юпитер Ноутбук https://jupyter.readthedocs.io/en/latest/install/notebook-classic.html (https://jupyter.readthedocs.io/en/latest/install/notebook-classic.html)

Для освоения и работы с ноутбуком iPython, рекомендуется пакет программ Анаконда (Anaconda).

https://www.anaconda.com/products/individual (https://www.anaconda.com/products/individual)

Анаконда содержит Пайтон, Юпитер ноутбук и другие часто используемые приложения для научных вычислений и обработки данных.

Порядок установки и запуска интерктивной среды Юпитер ноутбук

1. Загрузить Анаконда
2. Установить Анаконда
3. Запустить Юпитер ноутбук. Для этого использовать команду меню, либо консольную команду

   => jupyter notebook

Установка SQLAlchemy

In [ ]:
```
! pip install sqlalchemy
```

Установка драйверов баз данных

По умолчанию SQLAlchemy поддерживает SQLite3 без дополнительных драйверов. Для подключения к другим базам данных необходимы дополнительные драйвера баз данных.

- PostgreSQL. Установка драйвера Psycopg2: http://initd.org/psycopg/ (http://initd.org/psycopg/)
- MySQL (требуется версия MySQL 4.1 и выше)
- Другие

SQLAlchemy также можно использовать вместе с Drizzle, Firebird, Oracle, Syb-ase и Microsoft SQL Server. Сообщество также предоставило внешние диалекты для многих других баз данных, таких как IBM DB2, Informix, Amazon Redshift, EXASolution, SAP SQL Anywhere, Monet и многих других. Создание диалектов поддерживается SQLAlchemy.

In [ ]:
```
# PostgreSQL
! pip install psycopg2

# MySQL
! pip  install  pymysql
```

Соединение с базой данных

Чтобы подключиться к базе данных, нужно создать механизм (движок) SQLAlchemy. Механизм SQLAlchemy создает общий интерфейс с базой данных для выполнения операторов SQL.

SQLAlchemy предоставляет функцию для создания механизма с учетом *строки подключения* и, возможно, некоторых дополнительных именованных (keywords) аргументов. Строка подключения может содержать:

- Тип базы данных (Postgres, MySQL, etc.);
- Диалект, если он отличается от установленного по умолчанию для конкретного типа базы данных (Psycopg2, PyMySQL и т. д.);
- Дополнительные данные аутентификации (имя пользователя и пароль);
- Расположение базы данных (файл или имя хоста сервера базы данных);
- Дополнительный порт сервера базы данных;
- Необязательное имя базы данных.

Строка подключения позволяют нам использовать конкретный файл или место хранения. В примере 1 определяется файл базы данных SQLite с именем `listings.db</tt>:`

- `хранящийся в текущем каталоге;`
- `в памяти;`
- `с указанием полного пути к файлу (Unix и Windows).`

`В Windows строка подключения будет иметь вид engine4; \\ требуются для экранирования символа "слэш".`

`Функция create_engine</tt> возвращает экземпляр механизма SQLAlchemy.`

In [ ]:
```
# 0-1. Создание механизма SQLAlchemy со строкой подключения SQLite

import sqlalchemy
from sqlalchemy import create_engine

engine = create_engine('sqlite:///listings.db')
# engine2 = create_engine('sqlite:///:memory:')
# engine3 = create_engine('sqlite:////home/Airbnb/listings.db')
# engine4 = create_engine('sqlite:///c:\\Users\\Airbnb\\listings.db')
```

`PostgreSQL. Пример 2 демонстрирует создание механизма для локальной базы данных PostgreSQL с именем mydb`

In [ ]:
```
# 0-2. Создание механизма SQLAlchemy со строкой подключения PosrgreSQL

from sqlalchemy import create_engine
engine=create_engine('postgresql+psycopg2://username:password@localhos
t:5432/mydb')
```

MySQL. Пример 3 демонстрирует создание механизма для удаленной БД MySQL

In [ ]:
```
# 0-3. Создание механизма SQLAlchemy со строкой подключения для удаленной БД MySQL

from sqlalchemy import create_engine
engine = create_engine('mysql+pymysql://username:password''@mysql01.mikhail.internal/listings', pool_recycle=3600)
```

Теперь, когда создан экземпляр механизма соединения с базой данных, мы можем начать использовать SQLAlchemy Core чтобы связать наше приложение с сервисами базы данных.

1. Схема и типы данных

В SQLAlchemy имеется четыре категории типов данных:

- Универсальный
- Стандартный SQL
- Зависящий от поставщика
- Определяется пользователем

Универсальная категория типов данных предназначена для сопоставления типов данных в Python и SQL.

In [ ]:
```
import pandas as pd
pd.DataFrame(['BigInteger,int,BIGINT'.split(','),
              'Boolean,bool,BOOLEAN or SMALLINT'.split(','),
              'Date,datetime.date,DATE (SQLite: STRING)'.split(','),
              'DateTime,datetime.datetime,DATETIME (SQLite: STRING)'.split(','),
              'Enum,str,ENUM or VARCHAR'.split(','),
              'Float,float or Decimal,FLOAT or REAL'.split(','),
              'Integer,int,INTEGER'.split(','),
              'Interval,datetime.timedelta,INTERVAL or DATE from epoch'.split(','),
              'LargeBinary,byte,BLOB or BYTEA'.split(','),
              'Numeric,decimal.Decimal,NUMERIC or DECIMAL'.split(','),
              'Unicode,unicode,UNICODE or VARCHAR'.split(','),
              'Text,str,CLOB or TEXT'.split(','),
              'Time,datetime.time,DATETIME'.split(',')],
              columns='SQLAlchemy,Pyhon,SQL'.split(','))
```

Стандартные типы (например CHAR и NVARCHAR) используются в случаях, когда универсальные типы не отвечают требованиям из-за конкретной структуры данных.

Типы, зависящие от поставщика. Пример: поле JSON в PostgreSQL.

fromsqlalchemy.dialects.postgresqlimport JSON</tt>

Метаданные

Метаданные используются для связывания структуры базы данных. Метаданные полезно рассматривать как каталог объектов таблиц с дополнительной информацией о механизме и соединении. Метаданные необходимо импортировать и инициализировать. Инициализируем экземпляр объектов MetaData:

In [ ]:
```
# 1-1

from sqlalchemy import MetaData
metadata = MetaData()
```

Таблицы

Объекты таблиц инициализируются в SQLAlchemy Core путем вызова конструктора Table</tt> с именем таблицы и метаданными, аргументы считаются объектами столбцов. Столбцы создаются путем вызова Column с именем, типом и затем аргументами, которые представляют дополнительные конструкции и ограничения SQL. В примере 4 создадим таблицу, которая может использоваться для перечня объектов размещения гостиничного бизнеса airbnb: http://insideairbnb.com/get-the-data.html (http://insideairbnb.com/get-the-data.html)

In [ ]:
```
from sqlalchemy import (MetaData, Table, Column, Integer, Numeric, Stri
ng, DateTime, Boolean,
                        ForeignKey, create_engine)
```

In [ ]:
```
# 1-2

listing=Table('listing',metadata,
              Column('listing_id',Integer(),primary_key=True),
              Column('listing_name',String(50),index=True),
              Column('listing_url',String(255)),
              Column('host_id',Integer()),
              Column('neighbourhood_id',Integer()),
              Column('amenities',String(255)),
              Column('property_type_id',Integer()),
              Column('room_type_id',Integer()),
              Column('bedrooms',Integer()),
              Column('beds',Integer()),
              Column('normal_price',Numeric(7,2)),
              extend_existing=True
             )
```

In [ ]:
```
pd.DataFrame({'En':['listing_id','listing_name','listing_url','host_id
','neighbourhood_id',
                    'amenities','property_type_id','room_type_id','bedroo
ms','beds','price'],
              'Ru':['идентификатор объекта размещения','имя объекта разме
щения',
                    'адрес веб-страницы','идентификатор владельца',
                    'идентификатор местоположения','оборудование, удобств
а',
                    'тип собственности','тип помещения','число кроватей
','число спален','цена']})
```

Дополнительные аргументы

Рассмотрим использование дополнительных аргументов nullable, unique, onupdate</tt>

In [ ]:
```
from datetime import datetime
from sqlalchemy import DateTime
```

In [ ]:
```
# 1-3

user=Table('user',metadata,
             Column('user_id',Integer(),primary_key=True),
             Column('user_name',String(15),nullable=False,unique=True),
             Column('email_address',String(255),nullable=False),
             Column('phone',String(20),nullable=False),
             Column('password',String(25),nullable=False),
             Column('created_on',DateTime(),default=datetime.now),
             Column('updated_on',DateTime(),default=datetime.now,onupdat
e=datetime.now),
             extend_existing=True
            )
```

Ключи и ограничения

Ключи и ограничения задают с помощью объектов PrimaryKeyConstraint, UniqueConstraint, CheckConstraint</tt>

In [ ]:
```
from sqlalchemy import PrimaryKeyConstraint, UniqueConstraint, CheckCon
straint
```

Первичный ключ

В примерах 1-2 и 1-3 столбцы listing_id и user_id объявлялись первичными ключами с помощью ключевого слова primary_key. Также, можно определить составной первичный ключ, присвоив параметру primary_key значение True для нескольких столбцов. Таким образом, ключ рассматривается как кортеж, в котором столбцы, помеченные как ключ, присутствуют в порядке, в котором они были определены в таблице. Первичные ключи также могут быть определены после столбцов в конструкторе таблицы, как показано в следующем фрагменте.

In [ ]:
```
user=Table('user',metadata,
            Column('user_name',String(15),nullable=False,unique=True),
            Column('email_address',String(255),nullable=False),
            Column('phone',String(20),nullable=False),
            Column('password',String(25),nullable=False),
            Column('created_on',DateTime(),default=datetime.now),
            Column('updated_on',DateTime(),default=datetime.now,onupdat
e=datetime.now),
            PrimaryKeyConstraint('user_id', name='user_pk'),
            extend_existing=True
           )
```

Уникальность

Другое распространенное ограничение - ограничение уникальности, которое используется, чтобы гарантировать, что в данном поле значения не повторяются.

UniqueConstraint('username', name='uix_username')</tt>

Проверка значения

Этот тип ограничения используется, чтобы гарантировать, что данные, предоставленные для столбца, соответствуют набору критериев, определенных пользователем. В следующем фрагменте кода мы гарантируем, что price</tt> не может быть меньше 0,00:
CheckConstraint('price >= 0.00', name='unit_cost_positive')</tt>

Индексы

В примере 1-2 создан индекс для столбца listing_name. Когда индексы создаются, как показано в этом примере, они получают имена ix_listings_listing_name</tt>. Мы также можем определить индекс, используя явный тип конструкции. Можно обозначить несколько столбцов, разделив их запятой. Можно добавить аргумент ключевого слова unique = True</tt>, чтобы индекс был уникальным. При явном создании индексов они передаются в конструктор таблиц после столбцов. Чтобы имитировать указанный индекс явным способом, в конструктор Table</tt> требуется добавить Index('ix_listings_listing_name', 'listing_name')</tt>

In [ ]:
```
from sqlalchemy import Index
Index('ix_listings_listing_name', 'listing_name')
```

Мы также можем создавать функциональные индексы для ситуаций, когда часто требуется запрос на основе нескольких полей БД. Например, если мы хотим искать по параметрам оборудования ("удобства") и цены в качестве объединенного элемента, можно определить функциональный индекс для оптимизации поиска:

In [ ]:
```
Index('ix_test', listing.c.amenities, listing.c.price)
```

Связи и внешние ключи

Тепрь, когда имеются пользователи и объекты размещения, необходимо обеспечить связи, позволяющие пользователям бронировать те или иные объекты. Рассмотрим схему данных.

Создадим таблицы для заказов: order</tt> и line_item</tt>

In [ ]:
```
from sqlalchemy import ForeignKey
```

In [ ]:
```
# 1-4

order = Table('order', metadata,
              Column('order_id', Integer(),primary_key=True),
              Column('user_id', ForeignKey('user.user_id')),
              Column('confirmed', Boolean(),default=False),
              Column('order_price', Integer()),
              extend_existing=True
             )
```

In [ ]:
```
# 1-5

line_item = Table('line_item', metadata,
                  Column('line_item_id', Integer(), primary_key=True),
                  Column('order_id', ForeignKey('order.order_id')),
                  Column('listing_id', ForeignKey('listing.listing_id
')),
                  Column('item_start_date', DateTime()),
                  Column('item_end_date', DateTime()),
                  Column('item_price', Integer()),
                  extend_existing=True
                 )
```

В примерах 1-4 и 1-5 внешние ключи задаются с помощью строки: 'order.order_id'</tt>. Также существует явный способ задания ограничений по внешнему ключу: ForeignKeyConstraint(['order_id'], ['order.order_id'])</tt>

Сохранение таблиц

Все таблицы и определения связаны с экземпляром метаданных. Сохранение схемы в базе данных осуществляется посредством вызова метода create_all()</tt> в экземпляре метаданных с движком, в котором он должен создавать эти таблицы. По умолчанию create_all не будет пытаться воссоздать таблицы, которые уже существуют в базе данных, и его можно запускать несколько раз. Движок (механизм) SQLAlchemy определен нами ранее в примере 0-1, экземпляр метаданных создан ранее в примере 1-1. Теперь осуществим вызов метода create_all()</tt>

In [ ]:
```
metadata.create_all(engine)
```

Дополнительно: DB Browser for SQLite

https://sqlitebrowser.org/ (https://sqlitebrowser.org/)