Урок 3

Хранение данных в БД. ORM SQLAlchemy

Python DB-API. Подключение к базе данных, объект курсора, выполнение SQL-запросов. Использование ORM для работы с базами данных. ORM SQLAlchemy. Классический и декларативный стиль работы.

Введение

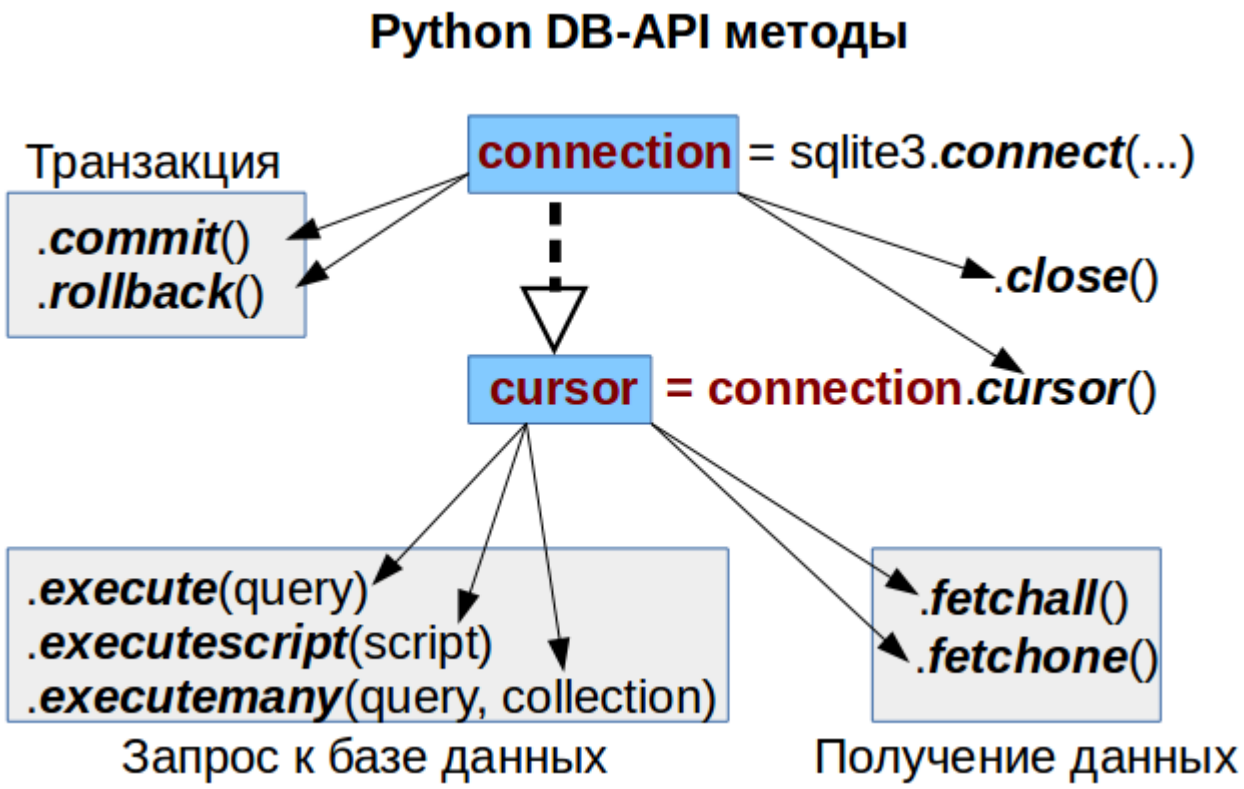
На этом уроке изучим взаимодействие Python-кода с реляционными базами данных. Рассмотрим два подхода: с использованием Python DB-API и с ORM-библиотекой SQLAlchemy.

Потребуются базовые представления о реляционных базах данных (таблицы, отношения, ключи, индексы) и языке SQL (добавление, редактирование, выборка данных).

В большинстве примеров увидим реляционную СУБД SQLite, так как она не требует установки дополнительного ПО, а модуль **sqlite3** является частью стандартной библиотеки Python. Где необходимо, приводятся примеры взаимодействия с другими СУБД.

Python DB-API

Python DB-API — это не конкретная библиотека, а набор правил, которым подчиняются отдельные модули, работающие с базами данных. Есть нюансы реализации, но общие принципы позволяют использовать один подход при работе с разными базами данных.



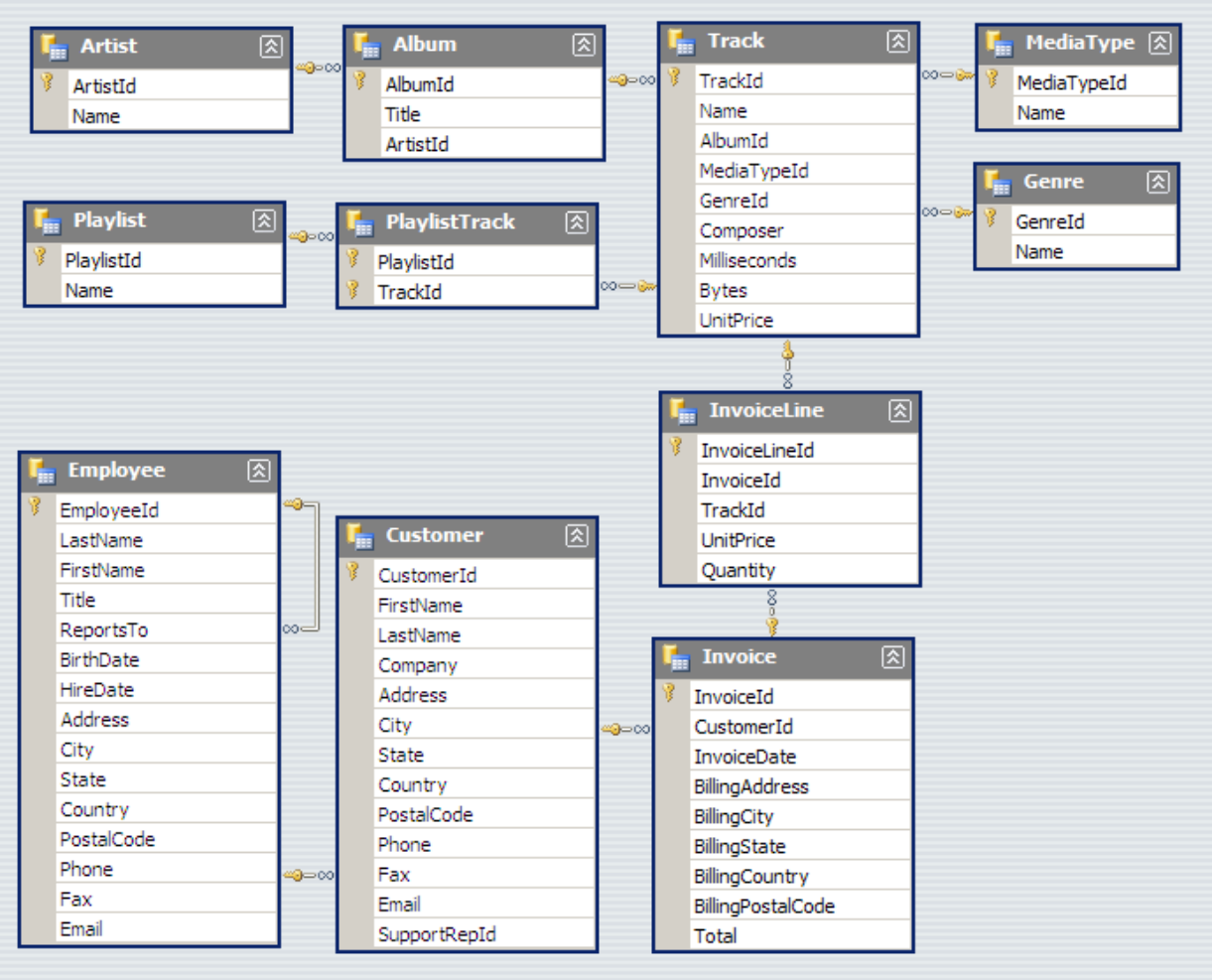
Подробное описание Python DB-API — в документе **PEP-249** (PEP 249 – Python Database API Specification v2.0).

Необходимые инструменты

- У Python есть встроенная поддержка базы данных SQLite. Дополнительно ничего устанавливать не надо, достаточно в скрипте указать импорт стандартной библиотеки:

```
import sqlite3
```

- В некоторых примерах будем использовать тестовую базу данных Chinook Database (лицензия MIT). Для примеров потребуется бинарный файл **Chinook_Sqlite.sqlite** (**examples/Chinook_Sqlite.sqlite**). Структура **Chinook Database**:



- Работать с базой (просматривать, редактировать) удобнее, если использовать программу-браузер баз данных, поддерживающую SQLite. Браузер наглядно покажет, что происходит с базой в процессе экспериментов.

Примечание: внося изменения в базу, не забудьте их применить, так как база с непримененными изменениями остается заблокированной.

Некоторые варианты браузеров БД:

- утилита для работы с базой в составе вашей IDE;
- SQLite Database Browser;
- SQLiteStudio;
- Valentina Studio.

Библиотеки для работы с БД

В стандартной библиотеке Python есть только модуль для взаимодействия с SQLite. Модули для других СУБД нужно устанавливать дополнительно. В большинстве случаев все необходимые модули устанавливаются через **pip**.

Перечислим библиотеки, обеспечивающие взаимодействие с СУБД, которые реализуют Python DB-API:

- **SQLite**: стандартный модуль sqlite3;
- **PostgreSQL**: psycopg2, PyGreSQL;
- **MySQL**: PyMySQL, mysql-connector-python;
- **MSSQL Server**: pyodbc, pymssql.

Соединение с базой, получение курсора

Рассмотрим базовый шаблон работы с DB-API, который будет использоваться во всех дальнейших примерах:

```
# Подключение библиотеки, соответствующей типу требуемой базы данных
import sqlite3

# Создание соединения с базой данных
# В данном случае это файл базы
conn = sqlite3.connect('Chinook_Sqlite.sqlite')

# Создаем курсор — это специальный объект, который делает запросы и получает их результаты
cursor = conn.cursor()

# ========== ТУТ БУДЕТ КОД РАБОТЫ С БАЗОЙ ДАННЫХ ==========
# ===== КОД ДАЛЬНЕЙШИХ ПРИМЕРОВ ВСТАВЛЯТЬ СЮДА =============

# В конце необходимо закрыть соединение с базой данных
conn.close()
```

При работе с другими базами данных можно использовать дополнительные параметры соединения — например, для PostgreSQL:

```
conn = psycopg2.connect(host=hostname, user=username, password=password, dbname=database)
```

Чтение из базы

Чтобы получить данные из БД, нужно выполнить SQL-запрос через метод курсора **execute()**, после чего получить данные через одним из fetch-методов:

```
# Выполняется SELECT-запрос к базе данных с применением обычного SQL-синтаксиса
cursor.execute("SELECT Name FROM Artist ORDER BY Name LIMIT 3")

# Получение результатов запроса
results = cursor.fetchall()
results2 =  cursor.fetchall()

print(results)
# [('A Cor Do Som',), ('Aaron Copland & London Symphony Orchestra',), ('Aaron Goldberg',)]
print(results2)
# []
```

Обратите внимание: второй раз результат из курсора без повторения самого запроса получить нельзя — вернется пустым!

Длинные запросы можно разбивать на несколько строк в произвольном порядке, если они заключены в тройные кавычки: одинарные (‘‘‘…’’’) или двойные (“””...”””):

```
cursor.execute("""
  SELECT name
  FROM Artist
  ORDER BY Name LIMIT 3
""")
```

Запись в базу

Чтобы добавить записи в БД, надо выполнить (**execute**) SQL-запрос **INSERT** и подтвердить транзакцию (**commit**):

```
# Выполняется INSERT-запрос к базе данных с обычным SQL-синтаксисом
cursor.execute("insert into Artist values (Null, 'A Aagrh!') ")

# Если выполняются изменения в базе данных, необходимо сохранить транзакцию
conn.commit()
# Проверка результатов
cursor.execute("SELECT Name FROM Artist ORDER BY Name LIMIT 3")
results = cursor.fetchall()
print(results)  # [('A Aagrh!',), ('A Cor Do Som',), ('Aaron Copland & London Symphony Orchestra',)]
```

Примечание: если к базе установлено несколько соединений и одно из них осуществляет ее модификацию, база SQLite блокируется до завершения (метод соединения **commit()**) или отмены (метод соединения **rollback()**) транзакции.

Несколько запросов за один раз

Метод курсора **execute()** позволяет делать только один запрос за раз: при попытке сделать несколько (через точку с запятой) будет ошибка:

```
cursor.execute("""
  insert into Artist values (Null, 'A Aagrh!');
  insert into Artist values (Null, 'A Aagrh-2!');
""")
# Будет ошибка
# sqlite3.Warning: You can only execute one statement at a time.
```

Решение — либо несколько раз вызвать метод курсора **execute()**:

```
cursor.execute("""insert into Artist values (Null, 'A Aagrh!');""")
cursor.execute("""insert into Artist values (Null, 'A Aagrh-2!');""")
```

...либо использовать метод курсора **executescript()**:

```
cursor.executescript("""
 insert into Artist values (Null, 'A Aagrh!');
 insert into Artist values (Null, 'A Aagrh-2!');
""")
```

Данный метод удобен, когда запрос сохранен в отдельной переменной или файле и требуется применить его к базе данных.

Подстановка значений в запрос

- **Важно!** Ни при каких условиях не используйте конкатенацию строк (+) или форматную строку для передачи переменных в SQL-запрос. Такое формирование запроса в сочетании с пользовательскими данными в нем — место для **SQL-инъекций**!

Правильный способ — использовать второй аргумент метода **execute()**.

В **SQLite** возможны два варианта:

```
# 1. С подстановкой по порядку на места знаков вопросов:
cursor.execute("SELECT Name FROM Artist ORDER BY Name LIMIT ?", ('2'))
# 2. С использованием именованных замен:
cursor.execute("SELECT Name from Artist ORDER BY Name LIMIT :limit", {"limit":
3})
```

Параметр paramstyle определяет, какой именно стиль используется для подстановки переменных в данном модуле:

```
import sqlite3
paramstyle = sqlite3.paramstyle
```

```
if paramstyle == 'qmark':
    ph = "?"
elif paramstyle == 'format':
    ph = "%s"
else:
    raise Exception("Unexpected paramstyle: %s" % paramstyle)

sql = "INSERT INTO foo VALUES (%(ph)s, %(ph)s, %(ph)s)" % { "ph" : ph }
```

Множественная подстановка значений

Для подстановки списка значений в запрос используется метод курсора **executemany()**:

```
# Обратите внимание: даже одно значение нужно передавать кортежем!
# Именно поэтому тут используется запятая в скобках
new_artists = [
    ('A Aagrh!',),
    ('A Aagrh!-2',),
    ('A Aagrh!-3',),
]
cursor.executemany("insert into Artist values (Null, ?);", new_artists)
```

Получение результатов

Для получения данных выборки (SELECT-запрос) применяются методы курсора:

- **fetchone()** — возвращает одну строку результата запроса, повторный вызов получает следующую строку и т.д. Всегда возвращает кортеж или **None**, если запрос пустой;
- **fetchmany([size=cursor.arraysize])** — возвращает набор строк результата указанного размера;
- **fetchall()** — получает все строки результата запроса.

Пример получения данных с применением метода курсора **fetchone()**:

```
cursor.execute("SELECT Name FROM Artist ORDER BY Name LIMIT 3")
print(cursor.fetchone())    # ('A Cor Do Som',)
print(cursor.fetchone())    # ('Aaron Copland & London Symphony Orchestra',)
print(cursor.fetchone())    # ('Aaron Goldberg',)
print(cursor.fetchone())    # None
```

Важно! Стандартный курсор забирает все данные с сервера сразу, вне зависимости от использования **fetchall()** или **fetchone()**.

Пример операций с БД формата SQLite через Python-библиотеку **sqlite3** представлен в **листинге 1**.

Курсор как итератор

Можно использовать объект курсора в качестве итератора:

```
# Использование курсора как итератора
```

```
for row in cursor.execute('SELECT Name from Artist ORDER BY Name LIMIT 3'):
        print(row)

# Полученный результат:
# ('A Cor Do Som',)
# ('Aaron Copland & London Symphony Orchestra',)
# ('Aaron Goldberg',)
```

Обработка ошибок

Для устойчивости программы (особенно при операциях записи) следует оборачивать инструкции обращения к БД в блоки **try-except-else** и использовать встроенный в **sqlite3** «родной» объект ошибок:

```
try:
    cursor.execute(sql_statement)
    result = cursor.fetchall()
except sqlite3.DatabaseError as err:
    print("Error: ", err)
else:
    conn.commit()
```

Использование row_factory

Атрибут **row_factory** позволяет дополнительно обрабатывать результат выборки (имеется доступ к метаданным запроса). По сути, **row_factory** — это callback-функция для обработки данных при возврате строки.

Можно обращаться к результату запроса по имени столбца. Для этого нужно воспользоваться атрибутом курсора **description**. Он возвращает сведения о столбцах для последней выборки (для каждого из них данные представлены кортежем из 7 элементов).

Пример из документации:

```
import sqlite3

def dict_factory(cursor, row):
    d = {}
    for idx, col in enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d

con = sqlite3.connect(":memory:")
con.row_factory = dict_factory
cur = con.cursor()
cur.execute("select 1 as a")
print(cur.fetchone()["a"])
```

Еще один пример возможностей **row_factory** — в **листинге 2**.

Справка по функциям Python DB-API

Модуль

- **connect()** — установка соединения с БД, создание объекта класса **Connection**;
- **threadsafety** — константа, указывающая уровень потокобезопасности модуля;
- **paramstyle** — строковая константа, задающая формат маркера подстановки данных в запрос.

Исключения (Exceptions)

- **Warning** — исключение для важных предупреждений (warnings);
- **Error** — базовый класс для исключений типа **error**;
- **InterfaceError** — ошибки, свойственные интерфейсу БД;
- **DatabaseError** — исключения, свойственные базе данных;
- **DataError** — исключения обработки данных (деление на ноль, выход за границы диапазона);
- **OperationalError** — исключения, относящиеся к операциям БД, которые не подконтрольны программисту (неожиданное отключение БД, неизвестное имя источника данных, невозможность выполнить транзакцию, ошибка выделения памяти);
- **IntegrityError** — исключение при нарушении целостности отношений (неудачная проверка внешнего ключа);
- **InternalError** — внутреннее исключение БД (некорректный курсор, ошибка синхронизации транзакции и прочее);
- **ProgrammingError** — программные ошибки (таблица не найдена или уже присутствует, ошибка SQL-синтаксиса, неверное количество параметров);
- **NotSupportedError** — исключение создается при использовании метода или API, которые не поддерживаются базой данных.

Соединение (Connection)

- **con.close()** — закрывает соединение с сервером базы данных;
- **con.commit()** — подтверждает все незавершенные транзакции;
- **con.rollback()** — откатывает все изменение в базе данных до момента, когда были запущены незавершенные транзакции.
- **con.cursor()** — создает новый курсор (экземпляр класса **Cursor**).

Курсор (Cursor)

- **cur.description** — последовательность кортежей с информацией о каждом столбце в текущем наборе данных. Кортеж имеет вид (**name, type_code, display_size, internal_size, precision, scale, null_ok**);

- **cur.rowcount** — число строк, на которые повлиял последний запрос;
- **c.arraysize** — целое число, которое используется методом **cur.fetchmany** как значение по умолчанию;
- **c.close()** — закрывает курсор, предотвращая выполнение запросов с его помощью;
- **c.callproc(procname [, param])** — вызывает хранимую процедуру;
- **c.execute(query [, param])** — выполняет запрос к базе данных (**query**);
- **c.executemany(query [, paramsequence])** — многократное выполнение запросов к базе данных (**query**);
- **c.fetchone()** — возвращает следующую запись из набора данных, полученного вызовом **c.execute*()**;
- **c.fetchmany([size])** — возвращает последовательность записей из набора данных;
- **c.fetchall()** — возвращает последовательность всех записей, оставшихся в полученном наборе данных;
- **c.nextset()** — пропускает все оставшиеся записи в текущем наборе данных и переходит к следующему;
- **c.setinputsize(sizes)** — сообщает курсору о параметрах, которые будут переданы в последующих вызовах методов **cur.execute*()**;
- **c.setoutputsize(sizes [, column])** — устанавливает размер буфера для определенного столбца в возвращаемом наборе данных.

Типы данных и их конструкторы

- **Date(year, month, day)** — формирует объект, содержащий дату;
- **Time(hour, minute, second)** — формирует объект, содержащий время;
- **Timestamp(year, month, day, hour, minute, second)** — формирует объект, содержащий временную метку;
- **DateFromTicks(ticks)** — формирует объект-дату из количества секунд;
- **TimeFromTicks(ticks)** — формирует объект-время из количества секунд;
- **TimestampFromTicks(ticks)** — формирует объект «дата–время» из количества секунд;
- **Binary(string)** — формирует объект с бинарными данными;
- **STRING** — тип для представления строковых столбцов таблицы (CHAR);
- **BINARY** — тип для представления бинарных столбцов таблицы (LONG, RAW, BLOB);
- **NUMBER** — тип для представления числовых столбцов таблицы;
- **DATETIME** — тип для представления столбцов «дата–время»;
- **ROWID** — тип для представления **Row ID** столбцов;

- **NULL-значения** представляются Python-объектом **None** как при вводе, так и при выводе.

SQLAlchemy

SQLAlchemy — это программная библиотека на Python для работы с реляционными СУБД с применением технологии ORM. **Object-Relational Mapping**, или объектно-реляционное отображение, связывает базы данных с концепциями объектно-ориентированных языков, создавая «виртуальную объектную БД». Служит для синхронизации объектов Python и записей реляционной базы данных. **SQLAlchemy** позволяет описывать структуры баз данных и способы взаимодействия с ними на Python, не используя SQL. Библиотека была выпущена в феврале 2006 под лицензией открытого ПО MIT.

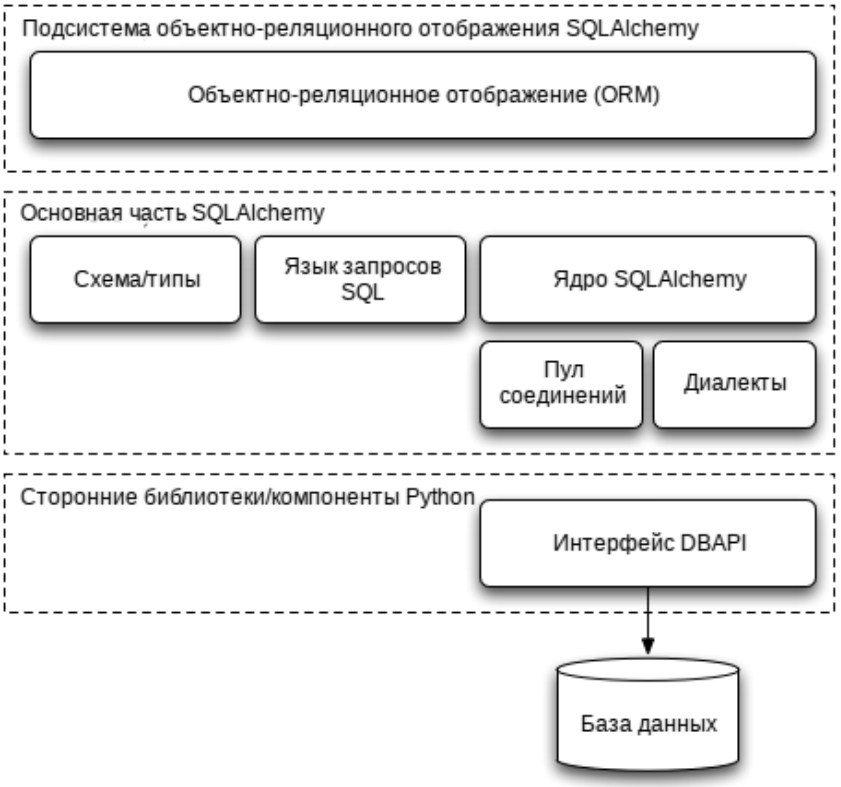
Работает **back end** для баз данных: **MySQL**, **PostgreSQL**, **SQLite**, **Oracle** и других, между которыми можно переключаться изменением конфигурации.

Преимущества

Использование SQLAlchemy для автоматической генерации SQL-кода имеет несколько преимуществ по сравнению с ручным написанием SQL-запросов:

- **Безопасность**. Параметры запросов экранируются, что делает атаки типа внедрения SQL-кода маловероятными;
- **Производительность**. Повышается вероятность повторного использования запроса к серверу базы данных, что может позволить ему в некоторых случаях применить повторно план выполнения запроса;
- **Переносимость**. **SQLAlchemy** при должном подходе позволяет писать на Python код, совместимый с несколькими back end СУБД. Несмотря на стандартизацию языка SQL, базы данных реализуют его по-разному. Абстрагироваться от этих нюансов помогает **SQLAlchemy**.

Архитектура SQLAlchemy



SQLAlchemy предоставляет богатый API для каждого уровня взаимодействия с БД, разбивая общую задачу взаимодействия на две категории: **ядро (Core)** и **объектно-реляционное представление (ORM)**. Ядро включает взаимодействие с **Python DB-API**, обработку текстовых SQL-запросов и управление схемой БД (все эти части предоставляют API). ORM-часть — это библиотека, построенная поверх ядра **SQLAlchemy** (разработчик может создать собственную).

Разделение на ядро и ORM всегда было особенностью **SQLAlchemy**. В этом есть как плюсы, так и минусы. Ядро **SQLAlchemy** позволяет ORM-слою:

- связывать в структуру **Table** атрибуты Python-класса, а не имена полей из БД;
- для формирования SELECT-запроса использовать структуру **select**, а не формировать строку запроса из разных частей;
- получать результат запроса через «фасад» (шаблон программирования) **ResultProxy**, который отображает select-структуру на каждую строку результата, а не передавать данные из курсора БД в пользовательские объекты.

Элементы ядра могут быть не видны в самом простом ORM-приложении. Ядро аккуратно встроено в ORM для плавного перехода между их конструкциями. Поэтому более сложное ORM-приложение может пропустить один или два уровня абстракции — чтобы взаимодействовать с БД, используя специфические и улучшенные настройки (если требуется).

Обратной стороной подхода «ORM–ядро» является то, что инструкции должны пройти много этапов. Стандартная реализация CPython имеет особенность вызова Python-функций, которая снижает быстродействие. Чтобы это обойти, можно сокращать цепочки вызовов функций и переносить критичные к быстродействию участки на язык C. Разработчики **SQLAlchemy** используют оба подхода для улучшения производительности (интерпретатор **PyPy** позволяет обходиться без трюков с улучшением быстродействия).

Установка SQLAlchemy

Установка **SQLAlchemy** стандартная:

```
pip install SQLAlchemy
```

Также можно скачать с официального сайта архив с SQLAlchemy и выполнить установочный скрипт **setup.py**:

```
python setup.py install
```

Чтобы удостовериться в правильности установки, следует проверить версию библиотеки:

```
import sqlalchemy
print("Версия SQLAlchemy:", sqlalchemy.__version__) # посмотреть версию
SQLALchemy
```

Объектно-реляционная модель SQLAlchemy

Соединение с базой данных

Будем использовать БД SQLite, хранящуюся в памяти, чтобы проще демонстрировать работу с **SQLAlchemy**.

Для соединения с СУБД используется функция **create_engine()** (см. **листинг 3**):

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///:memory:', echo=True)
```

Флаг **echo** включает ведение лога через стандартный модуль **logging** языка. Когда он включен, будут отображаться все создаваемые SQL-запросы.

По умолчанию соединение с БД через 8 часов простоя обрывается. Чтобы этого не случилось, нужно добавить опцию:

```
pool_recycle = 7200
```

Так каждые два часа соединение будет переустанавливаться.

Примеры создания подключений к базам данных PostgreSQL и MySQL:

```
# Создание подключения к локальной базе данных PostgreSQL
from sqlalchemy import create_engine
engine =
create_engine('postgresql+psycopg2://username:password@localhost:5432/mydb')

# Создание подключения к удаленной базе данных MySQL
from sqlalchemy import create_engine
engine =
create_engine('mysql+pymysql://cookiemonster:chocolatechip@mysql01.monster.intern
al'/
              '/cookies', pool_recycle=3600)
```

В рамках **SQLAlchemy** реализован фасадный класс для классического взаимодействия с **DB-API**. Точкой входа этого фасадного класса является вызов **create_engine**, с помощью которого устанавливается соединение и собирается конфигурационная информация. В качестве результата выполнения вызова возвращается экземпляр класса **Engine**. Этот объект представляет только способ осуществления запроса через **DB-API**, причем последний никогда непосредственно не раскрывается.

Для простого выполнения объект **Engine** предоставляет интерфейс явного исполнения запросов (**implicit execution interface**). Соединение с базой данных и курсором посредством DB-API создается и закрывается незаметно для разработчика:

```
engine = create_engine("postgresql://user:pw@host/dbname")
result = engine.execute("select * from table")
print(result.fetchall())
```

Создание таблиц

Далее необходимо «рассказать» **SQLAlchemy** о таблицах в базе данных.

Рассмотрим пример одиночной таблицы **users**, в которой хранятся записи о конечных пользователях, которые посещают сайт **N**. Необходимо определить таблицу внутри каталога **MetaData**, используя конструктор **Table()**, который похож на SQL-запрос **CREATE TABLE** (файл **листинг 3**):

```
from sqlalchemy import Table, Column, Integer, String, MetaData, ForeignKey

metadata = MetaData()
users_table = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('fullname', String),
    Column('password', String)
)
```

Далее необходимо выполнить запрос **CREATE TABLE**, параметры которого будут взяты из метаданных таблицы. Для этого вызывается метод **create_all()** с параметром **engine**, который указывает на базу. При выполнении метода автоматически проверяется присутствие такой таблицы перед ее созданием, так что можно выполнять этот метод много раз:

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

- **Обратите внимание**: колонки **VARCHAR** создаются без указания длины — для SQLite это вполне допустимый тип данных, но во многих других СУБД так делать нельзя. Чтобы выполнить этот урок в PostgreSQL или MySQL, длина для строк должна быть определена:

```
Column('name', String(50))
```

Поле «длина» в строках **String**, как и простая разрядность/точность в **Integer**, **Numeric**, используются только при создании таблиц.

Определение класса Python для отображения в таблицу

Класс **Table** хранит информацию о БД, но ничего не говорит о логике объектов, которые используются приложением. **SQLAlchemy** считает это отдельной задачей. Для соответствия таблице **users** создадим элементарный класс **User** (совершенно новый) (см. **листинг 3**):

```
class User:
    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)
```

Методы **__init__** и **__repr__** (вызывается в функции **print**) определены здесь для удобства. Они не обязательны и могут иметь любую форму. **SQLAlchemy** не вызывает **__init__** напрямую.

Настройка отображения

Теперь необходимо связать таблицы **users** и класс **User**. Эту задачу решает пакет **SQLAlchemy ORM**.

Чтобы создать отображение между таблицей и классом, необходимо использовать функцию **mapper**:

```
from sqlalchemy.orm import mapper
print(mapper(User, users_table))      #  <Mapper at 0x...; User>
```

Функция **mapper()** создаст новый Mapper-объект и сохранит его для дальнейшего применения, ассоциирующегося с нашим классом. Теперь создадим и проверим объект класса **User**:

```
from sqlalchemy.orm import mapper             # Mapper находится в пакете с ОРМ
mapper(User, users_table)                     # Создание отображения
user = User("Вася", "Василий", "qweasdzxc")
print(user)                                   # <User('Вася', 'Василий',
'qweasdzxc'>
print(user.id)                                # None
```

Атрибут **id**, который не определен в **__init__**, все равно существует из-за того, что колонка **id** существует в объекте таблицы **users_table**.

Стандартно **mapper()** создает атрибуты класса для всех колонок, что есть в **Table**. Это объекты-дескрипторы, которые определяют функциональность класса. Она может быть богатой, с возможностью отслеживать изменения и автоматически подгружать данные в базу.

Поскольку **SQLAlchemy** не получила задание сохранить «Василия» в базу, его id имеет значение **None**. Когда позже будет выполнено сохранение, в этом атрибуте будет автоматически сформированное значение.

Декларативное создание таблицы, класса и отображения

Мы приблизились к конфигурированию: рассмотрели таблицу **Table**, пользовательский класс и вызов **mapper()** — проиллюстрировали классический пример использования **SQLAlchemy**. В ней очень ценится разделение задач. Но множество приложений не требуют его, и для них **SQLAlchemy** предоставляет альтернативный, более лаконичный стиль — **декларативный.**

```
Base = declarative_base()
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)
```

Функция **declarative_base()** определяет новый класс (**Base**), от которого будут унаследованы все необходимые ORM-классы.

- **Обратите внимание**: объекты **Column** определены без указания строки имени — она будет получена из имени своего атрибута.

Объект **Table** доступен через атрибут **__table__**:

```
users_table = User.__table__
```

Имеющиеся метаданные **MetaData** также доступны:

```
metadata = Base.metadata
```

Итоги

С использованием **SQLAlchemy** работать с базой данных становится так же удобно, как со структурами языка программирования. Но **SQLAlchemy** подразумевает, что **разработчик понимает, как работает SQL**. В современных фреймворках для создания сайтов так или иначе используется ORM. В Django — своя реализация **django-orm**, а во **flask**, **pyramids** и других основном используется **SQLAlchemy**.

Можно также обратить внимание на другие ORM: **PeeWee**, **Pony ORM**.

Практическое задание

1. Начать реализацию класса «**Хранилище**» для серверной стороны. Хранение необходимо осуществлять в базе данных. В качестве СУБД использовать **sqlite**. Для взаимодействия с БД можно применять ORM.

   Опорная схема базы данных:

   - На стороне сервера БД содержит следующие таблицы:
     - клиент:
       - логин;
       - информация.
     - история_клиента:
       - время входа;
       - ip-адрес.
     - список_контактов (составляется на основании выборки всех записей с id_владельца):
       - id_владельца;
       - id_клиента.

Дополнительные материалы

1. Slideshare. Michael Bayer. Introduction to SQLAlchemy.
2. Slideshare. Relational Database Access with Python.
3. Slideshare. Introduction to SQLAlchemy by Jorge A. Medina.
4. ORM. Использование SQLAlchemy.
5. Вводная по сложным запросам в SQLAlchemy.
6. Python. Работа с базой данных, часть 1/2. Используем DB-API.
7. PEP 249 – Python Database API Specification v2.0.
8. The Novice's Guide to the Python 3 DB-API.

9. SQLAlchemy — как втянуться.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Myers Jason, Copeland Rick. Essential SQLAlchemy: Mapping Python to Databases (каталог «Дополнительные материалы»).
2. Michael Driscoll. Python 101 (Chapter 34. SQLAlchemy) (каталог «Дополнительные материалы»).
3. Лутц Марк. Программирование на Python, том II, 4-е издание (каталог «Дополнительные материалы»).
4. Бизли Дэвид. Python. Подробный справочник (каталог «Дополнительные материалы»).
5. Wiki Портала Python-программистов. SQLAlchemy.
6. SQLAlchemy Architecture. Michael Bayer.
7. Глава 20 из книги «Архитектура приложений с открытым исходным кодом», том 2.

Приложение

Листинг 1

```
# ------------------------------ Базы данных -----------------------------

# Импортируем библиотеку, соответствующую типу нашей базы данных
import sqlite3

# Создаем соединение с базой данных
# В примере это файл базы
with sqlite3.connect('company.db3') as conn:

    # Создаем курсор - это специальный объект, который делает запросы и
получает их результаты
    cursor = conn.cursor()

    # ТУТ БУДЕТ НАШ КОД РАБОТЫ С БАЗОЙ ДАННЫХ
    # КОД ДАЛЬНЕЙШИХ ПРИМЕРОВ ВСТАВЛЯТЬ В ЭТО МЕСТО

    # cursor.execute("""
    #                 create table if not exists Terminal (
    #                     id  INTEGER primary key,
    #                     title TEXT,
    #                     configuration TEXT
    #                 );
    #     """)

    # cursor.execute("""
    #                 insert into Terminal (id, title, configuration)
```

```
    #                 VALUES (?, ?, ?);""",
    #       (13, 'Terminal Bingo', '{"simle":"nothing"}'))

    cursor.execute('SELECT * FROM Terminal where id = ?', '13')

    z = cursor.fetchone()
    while z:
        print(z)
        z = cursor.fetchone()

# Не забываем закрыть соединение с базой данных
# conn.close()
```

Листинг 2

```
# ------------------------------ Базы данных -----------------------------
#                          Использование row_factory

import sqlite3


# Создадим обработчик row_factory, возвращающий список данных, а не кортежей
# при выборе только одного поля из таблицы
def my_row_factory(cursor, row):
    d = {}
    print(cursor.description)
    print(row[0])
    if len(cursor.description) == 1:
        return row[0]
    else:
        return row

con = sqlite3.connect(":memory:")

con.row_factory = my_row_factory
cur = con.cursor()

cur.execute('CREATE table USER(id integer primary key, name text)')

cur.execute('INSERT INTO USER(name) values (?)', ('petrushka',))
cur.execute('INSERT INTO USER(name) values (?)', ('barmalei',))
cur.execute('INSERT INTO USER(name) values (?)', ('alien',))
cur.execute('INSERT INTO USER(name) values (?)', ('scally',))
cur.execute('INSERT INTO USER(name) values (?)', ('molder',))
con.commit()


cur.execute("select * from USER as a")
print(cur.fetchall())
```

Листинг 3

```
# ------------------------------ Базы данных ------------------------------

# SQLAlchemy. Часть 1

import sys

# Проверим версию SQLAlchemy
try:
    import sqlalchemy
    print(sqlalchemy.__version__)
except ImportError:
    print('Библиотека SQLAlchemy не найдена')
    sys.exit(13)

# -------------------------------------------------------------------------

print(' ------ Классическое создание таблицы, класса и отображения ------')

# Работа с классическим отображением (Classical Mapping) #
########################################################

from sqlalchemy import create_engine

# Создадим БД в памяти или в файле
# Флаг `echo` включает ведение лога через стандартный модуль `logging` Питона.
engine = create_engine('sqlite:///:memory:', echo=True)

# Импортируем необходимые классы (типы данных, таблицы, метаданные, ключи)
from sqlalchemy import Table, Column, Integer, String, MetaData

# Подготовим "запрос" на создание таблицы users внутри каталога MetaData
metadata = MetaData()
users_table = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('fullname', String),
    Column('password', String)
)

# Выполним запрос CREATE TABLE
metadata.create_all(engine)


# Создадим класс для отображения таблицы БД
class User:
    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)


# Выполним связывание таблицы и класса-отображения
from sqlalchemy.orm import mapper
m = mapper(User, users_table)
print('Classic Mapping. Mapper: ', m)

# Создадим объект-пользователя
classic_user = User("Вася", "Василий", "qweasdzxc")
print('Classic Mapping. User: ', classic_user)
print('Classic Mapping. User ID: ', classic_user.id)

# --------------------------------------------------------------------------------

print(' ------ Декларативное создание таблицы и класса ------')

# Декларативное создание таблицы, класса и отображения #
#######################################################

# Для использования декларативного стиля необходима функция declarative_base
from sqlalchemy.ext.declarative import declarative_base

# Функция declarative_base создаёт базовый класс для декларативной работы
Base = declarative_base()

# На основании базового класса можно создавать необходимые классы
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)

# Таблица доступна через атрибут класса
users_table = User.__table__
print('Declarative. Table:', users_table)

# Метаданные доступны через класс Base
metadata = Base.metadata
print('Declarative. Metadata:', metadata)
```

```
print(' ----------------- Работа с сессией ---------------------------------')

#                     Создание сессии                          #
################################################################

from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)


# Класс Session будет создавать Session-объекты, которые привязаны к базе
данных
session = Session()
print('Session:', session)

#                     Добавление новых объектов                    #
###################################################################

# Для сохранения объекта User нужно добавить его к имеющейся сессии
admin_user = User("vasia", "Vasiliy Pypkin", "vasia2000")
session.add(admin_user)

# Объект, созданный через классическое отображение,
# также сохраняется в БД через сессию
session.add(classic_user)

# Простой запрос
q_user = session.query(User).filter_by(name="vasia").first()
print('Simple query:', q_user)

# Добавить сразу несколько записей
session.add_all([User("kolia", "Cool Kolian[S.A.]","kolia$$$"),
                 User("zina", "Zina Korzina", "zk18")])

# Сессия "знает" об изменениях пользователя
admin_user.password = "-=VP2001=-"
print('Session. Changed objects:', session.dirty)

# Атрибут `new` хранит объекты, ожидающие сохранения в базу данных
print('Session. New objects:', session.new)

# Метод commit() фиксирует транзакцию, сохраняя оставшиеся изменения в базу
session.commit()

print('User ID after commit:', admin_user.id)
```