Объектно-ориентированное программирование на Python

Осипов Никита Алексеевич

ЛЕКЦИЯ. СОЗДАНИЕ ПРИЛОЖЕНИЙ PYTHON ДЛЯ ВЗАИМОДЕЙСТВИЯ С БАЗОЙ ДАННЫХ

Учебные вопросы:

1. Применение плоских файлов для хранения данных.
2. Создание приложений с базой данных SQLite.
3. ORM.

Справочная информация

- SQLite Python Tutorial
- www.sqlitetutorial.net

Применение плоских файлов для хранения данных

- Для хранения данных можно использовать «плоские» (flat) файлы в любом из форматов: CSV, JSON, XML.
- Под *плоской базой* данных (*плоской таблицей*) понимается файл, содержащий данные без внутренней иерархии и ссылок на внешние файлы
  - Например, csv-файлы (comma separated values) представляют собой строки простого текста, в которых элементы данных разделены запятыми:
    - Каждая строка текста представляет собой строку данных, а каждое значение в строке, отделенное от остальных запятой, соответствует одному из полей таблицы.
  - Python предоставляет возможности для работы с таким форматом данных – имеет встроенный модуль csv и стороннюю библиотеку pandas

База данных SQLite

- SQLite – это библиотека, реализующая легковесную дисковую базу данных, не требующую отдельного серверного процесса и позволяющую получить доступ к БД с использованием языка запросов SQL

- Приложения могут использовать SQLite для внутреннего хранения данных
- Можно создать прототип приложения с использованием SQLite, а затем перенести код в более многофункциональную БД, такую как PostgreSQL или Oracle
- Модуль sqlite3 реализует интерфейс SQL, соответствующий спецификации DB-API 2.0

Работа с базой данных SQLite

- Python имеет встроенную поддержку базы данных SQLite и достаточно в скрипте указать импорт стандартной библиотеки:

```
import sqlite3
```

Основные операции в приложении при работе с базой данных:

1. Загрузка библиотеки
2. Создание и соединение с базой данных
3. Создание таблиц базы данных
4. Добавление данных
5. Запросы на получение данных
6. Обновление данных
7. Удаление данных

Создание базы данных SQLite

sqlite_01.py

```
con = sqlite3.connect('mydb.db')
# или
con = sqlite3.connect(':memory:')


cursorObj = con.cursor()



cur.execute("SQL-ЗАПРОС;")
```

Объект подключения создается с помощью метода **connect**()

Создание БД в памяти – отличный вариант для тестирования

Для выполнения операторов SQL, нужен объект курсора, создаваемый методом объекта соединения **cursor**()

Теперь можно использовать объект курсора для вызова метода **execute**() для выполнения любых запросов SQL

Создание базы данных SQLite

sqlite_03.py

```
import sqlite3
from sqlite3 import Error


try:
    con = sqlite3.connect(':memory:')
    print("Connection is established: Database - in memory")
except Error:
    print(Error)
finally:
    con.close()
```

Создается БД с блоками try, except и finally для обработки возможных исключений

Выполнение запросов

sqllite_02.py

- 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
""")
```

- Метод курсора **executescript**() позволяет выполнить несколько запросов за раз – указать запросы через точку с запятой:

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

```
conn.executescript(schema)
```

Данный метод также применяется, когда запросы сохранены в отдельной переменной или в файле и их надо применить к базе

Подстановка значений в запрос

sqllite_01,03,04.py

- Не рекомендуется использовать конкатенацию строк (+) или интерполяцию параметра в строке (%) для передачи переменных в SQL запрос
- Рекомендуется – использование второго аргумента метода execute():

Должен передаваться кортеж

```
# С подстановкой по порядку на места знаков вопросов:
age = 22,
curdat = cur.execute("SELECT * FROM user WHERE age <= ?", age)

# Или с использованием именованных переменных:
cursor.execute("SELECT Name from Artist ORDER BY Name LIMIT
:limit", {"limit": 3})
```

Вставка строк

sqllite_01.py

- Чтобы вставить данные в таблицу воспользуемся оператором INSERT INTO

```
cursorObj.execute("INSERT INTO employees VALUES(1, 'John', 700, 'HR',
'Manager', '2020-01-04')")
```

- Можно использовать знак вопроса (?) в качестве заполнителя для каждого значения

```
cursorObj.execute('''INSERT INTO employees(id, name, salary, department,
position, hireDate) VALUES(?, ?, ?, ?, ?, ?)''', entities)
```

- Для вставки нескольких строк одновременно следует использовать метод **executemany**()

```
data = [(1, "Ridesharing"), (2, "Water Purifying"), (3, "Forensics"), (4,
"Botany")]

cursorObj.executemany("INSERT INTO projects VALUES(?, ?)", data)
```

Выборка данных

sqllite_01, 03.py

- Оператор SELECT используется для выборки данных и выполняется в методе **execute** объекта курсора

```
cursorObj.execute('SELECT * FROM employees ')
```

- Для получения одиночного результата – метод курсора **fetchone**()

```
one_result = cursorObj.fetchone()
```

- Для получения нескольких данных используется метод **fetchmany**()

```
three_results = cursorObj.fetchmany(3)
```

- Для получения всех результатов запроса используется метод **fetchall**()

```
rows = cursorObj.fetchall()
```

Курсор забирает все данные с сервера согласно запроса

Обновление и удаление данных

sqllite_01, 03.py

- Оператор UPDATE используется для обновления данных

```
cursorObj = con.cursor()

cursorObj.execute('UPDATE employees SET name = "Rogers" where id = 2')
con.commit()
```

- Оператор DELETE используется для удаления данных

```
 cursorObj = con.cursor()

cursorObj.execute("DELETE FROM users WHERE lname='Parker';")
con.commit()
```

Введение в Object Relational Mapping

- ORM (Object Relational Mapping) — метод программирования для преобразования данных между несовместимыми системами типов в объектно-ориентированных языках программирования

- Основной целью API Object Relational Mapper является облегчение связывания пользовательских классов Python с таблицами базы данных, а объектов этих классов — со строками в соответствующих таблицах.
- Изменения в состояниях объектов и строк синхронно сопоставляются друг с другом. Можно выражать запросы к базе данных в терминах пользовательских классов и их определенных отношений.
- ORM построен поверх языка выражений SQL. Это высокий уровень и абстрактная схема использования (ORM — это прикладное использование языка выражений).

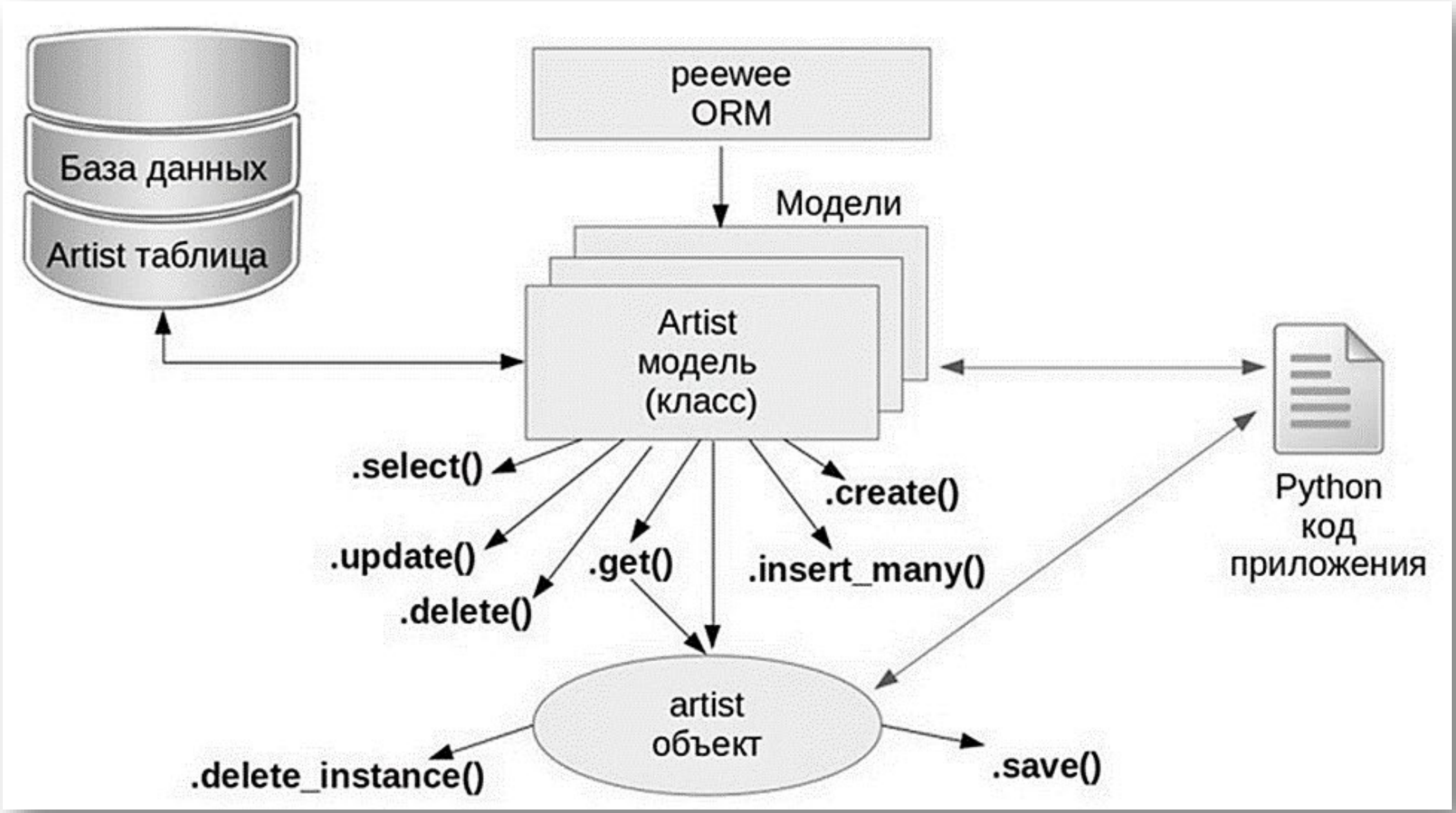
Объектно-реляционное отображение (ORM)

- Объектно-реляционное отображение (ORM) – классы сопоставляются с базой данных – что позволяет четко разделить объектную модель и схему базы данных
  - каждый класс отображается на таблицу в базовой базе данных
- Основные подходы к реализации ORM:
  - **Active Record** – отображение объекта данных на строку базы данных
  - **Data Mapper** – полностью разделяет представление данных в программе от его представления в базе данных

Возможные решения

- peewee
  - лёгкая, быстрая, гибкая ORM на Python, которая поддерживает SQLite, MySQL и PostgreSQL – применяет подход Active Record
- SQLAlchemy
  - поддерживает SQLite, MySQL, PostgreSQL, Oracle
  - применяет шаблон отображения данных Data Mapper

Обзор peewee



peewee – настройка

- Загрузить и установить последнюю версию **peewee**

```
pip install peewee
```

- **peewee** предназначен для работы с реализацией DBAPI, созданной для конкретной базы данных

```
from peewee import *

# Создаем соединение с нашей базой данных
conn = SqliteDatabase('BD_Sqlite.sqlite')

# КОД МОДЕЛЕЙ
# курсор - специальный объект для запросов и получения данных с базы
cursor = conn.cursor()

# КОД РАБОТЫ С БАЗОЙ ДАННЫХ
# закрыть соединение с базой данных
conn.close()
```

Описание моделей и их связь с базой данных

ORM концепция	Концепция базы данных
Класс модели	Таблица базы данных
Поле экземпляра (атрибут объекта)	Колонка в таблице базы данных

```
# Определяем базовую модель от которой будут наследоваться остальные
class BaseModel(Model):
    class Meta:
        database = conn  # соединение с базой

# Определяем модель исполнителя
class Artist(BaseModel):
    artist_id = AutoField(column_name='ArtistId')
    name = TextField(column_name='Name', null=True)

    class Meta:
        table_name = 'Artist'
```

Таблица связи между типом поля в модели и в базе данных

SQLAlchemy – настройка

- Загрузить и установить последнюю версию SQLAlchemy

```
pip install sqlalchemy
```

- SQLAlchemy предназначен для работы с реализацией DBAPI, созданной для конкретной базы данных
  - Использует систему диалектов для связи с различными типами реализаций DBAPI и баз данных
  - Все диалекты требуют, чтобы был установлен соответствующий драйвер DBAPI
  - Некоторые диалекты включены по умолчанию: FireBird, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, SQLite, Sybase

См. SQLAlchemy в викиучебнике

Подключение к базе данных и создание таблицы

- Объект класса **Engine** создается с помощью функции create_engine()

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///myDBase.db')
```

- Определения таблиц и связанных объектов (индекс, представление, триггеры) содержатся в метаданных
  - объект класса MetaData из метаданных SQLAlchemy представляет собой коллекцию объектов Table и связанных с ними конструкций схемы, содержит также привязку к Engine или Connection

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

- Объект класса **Table** представляет соответствующую таблицу

```
students = Table('students', meta,
    Column('id', Integer, primary_key = True),
    Column('name', String),
    Column('lastname', String),
)
```

- Создание таблицы и сохранение информации

```
meta.create_all(engine)
```

Примеры запросов SQLite и запросов с использованием SQLAlchemy

where

```
SQL :
      SELECT * FROM census
      WHERE level = F
SQLAlchemy :
      db.select([census]).where(census.columns.level == 'F')
```

in

```
SQL :
      SELECT state, level
      FROM census
      WHERE state IN (Texas, New York)
SQLAlchemy :
      db.select([census.columns.state,census.columns.level]).where(census.
columns.state.in_(['Texas', 'New York']))
```

Примеры запросов SQLite и запросов с использованием SQLAlchemy

order by

SQL :

```
SELECT * FROM census
ORDER BY State DESC, pop2000
```

SQLAlchemy :

```
db.select([census]).order_by(db.desc(census.columns.state),
census.columns.pop2000)
```

functions

SQL :

```
SELECT SUM(pop2008)
FROM census
```

SQLAlchemy :

```
db.select([db.func.sum(census.columns.pop2008)])
```

Примеры запросов SQLite и запросов с использованием SQLAlchemy

- Если есть две таблицы, у которых уже есть установленная связь, можно автоматически использовать эту связь, просто добавив нужные столбцы из каждой таблицы в оператор выбора

```
select([census.columns.pop2008, state_fact.columns.abbreviation])
```

Обновление данных в базе данных

```
db.update(table_name).values(attribute = new_value).where(condition)
```

Удаление таблицы

```
db.delete(table_name).where(condition)
```

sqlAlchemy03.py

sqlAlchemy04.py

SQLAlchemy – сопоставление

- Процесс конфигурации ORM включает описание таблиц базы данных и определение классов, которые будут сопоставлены с этими таблицами
  - В SQLAlchemy эти две задачи выполняются вместе с помощью *декларативной системы*: созданные классы включают директивы для описания фактической таблицы базы данных, в которую они отображаются.
- Базовый класс хранит каталог классов и сопоставленных таблиц в декларативной системе – декларативный базовый класс
  - Для создания базового класса используется функция declarative_base() (определена в модуле sqlalchemy.ext.declarative)

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

```
class Customers(Base):
    __tablename__ = 'customers'
    id = Column(Integer, primary_key = True)
    name = Column(String)
    address = Column(String)
    email = Column(String)
```

Класс содержит таблицу для сопоставления, а также имена и типы данных столбцов

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

sqlAlchemy03.py

sqlAlchemy04.py

SQLAlchemy – создание сеанса

- Для взаимодействия с базой данных требуется получить ее дескриптор.
  - Объект сеанса является дескриптором базы данных.
  - Класс сеанса определяется с помощью **sessionmaker**() — настраиваемого метода фабрики сеансов, который привязан к объекту.

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

sqlAlchemy03.py

sqlAlchemy04.py

SQLAlchemy – добавление объектов

- Требуется объявить объект класса и добавить его в таблицу методом **add**() объекта сеанса

```
c1 = Sales(
      name = 'Ravi Kumar',
      address = 'Station Road Nanded',
      email = 'ravi@gmail.com')
session.add(c1)
```

- Транзакция ожидает, пока она не будет сброшена с использованием метода **commit**()

```
session.commit()
```

SQLAlchemy – получение данных

sqlAlchemy04b.py

- Объект **Query** обеспечивает создание операторов SELECT
  - Объект запроса имеет метод all(), который возвращает набор результатов в виде списка объектов

```
result = session.query(Customers).all())
```

- Это утверждение фактически эквивалентно выражению SQL:

```
SELECT customers.id
   AS customers_id, customers.name
   AS customers_name, customers.address
   AS customers_address, customers.email
   AS customers_email
FROM customers
```

SQLAlchemy – обновление данных

sqlAlchemy04b.py

- Чтобы изменить данные определенного атрибута любого объекта нужно
  - присвоить ему новое значение
  - зафиксировать изменения
- Для извлечения объект из таблицы, чей идентификатор первичного ключа ID = 2 можно использовать метод **get()** :

```
x = session.query(Customers).get(2)
```

- Обновим поле Address, назначив новое значение:

```
x.address = 'Banjara Hills Secunderabad'
session.commit()
```

- Для массовых обновлений – метод **update()**:

```
session.query(Customers).filter(Customers.id!=2).update({Customers.name:
"Mr."+Customers.name},synchronize_session = False)
```

SQLAlchemy – фильтрация данных

sqlAlchemy04b.py

- Данные, представленные объектом Query, могут быть отобраны по определенному критерию с помощью метода **filter**()

```
session.query(class).filter(criteria)
```

- Набор результатов, полученный с помощью запроса SELECT в таблице «Клиенты», фильтруется по условию (ID> 2)

```
result = session.query(Customers).filter(Customers.id>2)
```

- Метод **like**() создает критерии LIKE для предложения WHERE в выражении SELECT

```
result = session.query(Customers).filter(Customers.name.like('%Ra%'))
```

- Соединение нескольких критериев – разделение запятыми или использования метода **and_** ()

```
result = session.query(Customers).filter(and_(Customers.id>2,
Customers.name.like('%Ra%')))
```

Практическое задание – БД

- Реализовать хранение данных в базе данных.
- Выбор типа базы данных и технологии разработки на усмотрение разработчика – обосновать принятые решения.