Лекция №12

Работа с базами данных

- Базы данных
- Модели данных
- SQL
- Базы данных в Python
- Python DB-API
- SQLite
- ORM
- SQLAlchemy
- MongoDB
- Redis
- Практика

Базы данных

База данных – это совокупность записей, систематизированных таким образом, чтобы эти записи могли быть найдены и обработаны с помощью компьютера.

Правильно читать, создавать и редактировать эти записи умеет система управления базами данных (СУБД) или сервер баз данных. СУБД выполняет манипуляции с данными в соответствии с командами, которые она получает. Сервер баз данных отвечает за целостность и сохранность данных, а также обеспечивает операции ввода-вывода при доступе клиента к информации.

Основное свойство базы данных – целостность означает, что в базе данных содержится полная, непротиворечивая и адекватно отражающая предметную область информация. В связи с этим свойством к проектируемой базе данных предъявляется ряд требований, таких как неизбыточность данных, безопасность и контроль доступа к данным различных групп пользователей.

Модели данных

Хранимые в базе данные имеют определенную логическую структуру – иными словами, описываются некоторой моделью представления данных, поддерживаемой СУБД. Существуют различные модели данных:

- Иерархическая модель. В этой модели связи между данными можно описать с помощью упорядоченного графа (дерева).
- Сетевая модель позволяет отображать разнообразные взаимосвязи элементов данных в виде произвольного графа, обобщая тем самым иерархическую модель данных.
- Многомерная модель представляется набором гиперкубов, применяется, как правило, в узкоспециализированных областях.
- Объектно-ориентированная модель рассматривает отдельные записи базы как свойства объектов, а связанные по смыслу совокупности таких записей – как сами объекты.

Модели данных

- Реляционная (постреляционная) модель - совокупность отношений, содержащих информацию о предметной области. Упрощенным представлением реляционной модели данных является набор таблиц. В настоящее время реляционная модель пересмотрена в виде постреляционной модели, в которой снято ограничение неделимости данных, хранящихся в записях таблиц.

Наиболее распространенной является реляционная модель. Таблицы являются основой реляционной базы данных. Порядок строк в таблице произволен. Строки можно однозначно идентифицировать по первичный ключам. Столбы таблиц именуются и нумеруются. Таблицы могут быть связаны друг с другом через внешний ключ.

SQL

Сервер реляционной базы данных, как правило, понимает команды на языке SQL, который де-факто стал стандартом управления данными в реляционных базах данных. SQL (Structured Query Language — язык структурированных запросов) позволяет осуществлять следующие операции:

- создание баз данных и отдельных таблиц с полным описанием их структуры;
- выполнение основных операций манипулирования данными (такие как вставка, модификация и удаление данных из таблиц);
- выполнение простых и сложных запросов.

В соответствии с этим SQL условно подразделяется на DDL (Data Definition – определение данных), DML (Data Manipulation – манипулирование данными), DCL (Data Control – управление доступом к данным), TCL (Transaction Control – управление транзакциями).

Базы данных и Python

Существуют различные СУБД, использующие реляционную модель данных: SQLite, PostgreSQL, Oracle и т.д. Для работы с этими СУБД в Python существуют соответствующие библиотеки (sqlite3, psycopg2, mysql-python, cx_Oracle и т.д.).

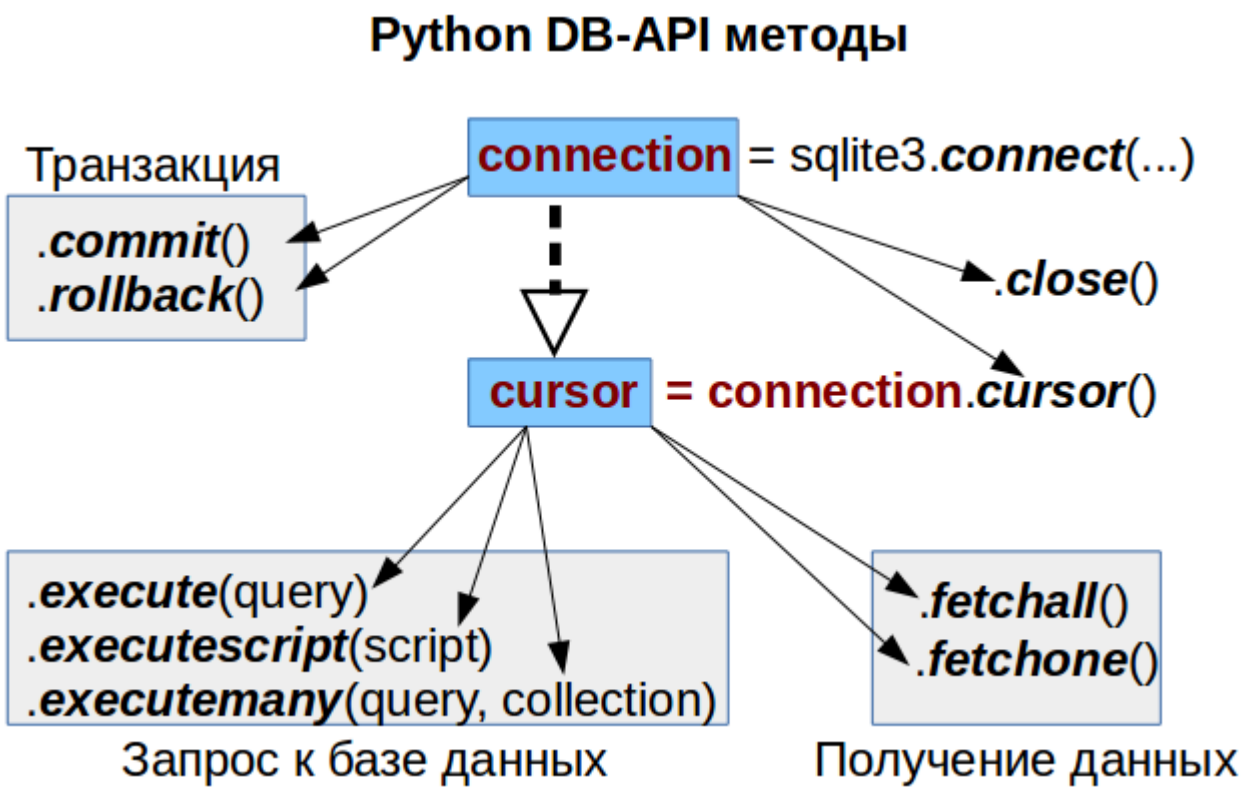
PEP 249 документирует интерфейс для работы с базами данных, и почти все библиотеки Python для работы с базами данных соответствуют этому стандарту.

Также в Python можно применять ORM (object relation mapping), работая с таблицами как с объектами, с помощью библиотеки sqlalchemy.

Есть в Python и библиотеки для работы с нереляционными базами данных, такими как mongodb.

Python DB-API

PEP 249 определяет DP-API - набор методов и интерфейсов для работы с базами данных



SQLite

SQLite это библиотека, написанная на языке C, предоставляющая легковесную файловую базу данных, не требующую отдельного серверного процесса и понимающую обращения на одном из диалектов SQL. Приложения могут использовать SQLite для хранения внутренних данных. Также SQLite позволяет эффективно и быстро создавать прототипы приложений, использующих базы данных, а затем портировать код для работы с более сложными базами данных, такими как PostgreSQL или Oracle.

Для работы с sqlite через Python достаточно импортировать библиотеку sqlite3. Для работы напрямую (без Python), можно скачать консоль или браузер.

SQLite

Для установки sqlite3 консоли в Windows надо использовать инсталлятор, который можно скачать здесь: https://www.sqlite.org/download.html
В Linux (Ubuntu), если консоль не предустановлена, надо воспользоваться apt-get, выполнив команду:

```
$ sudo apt-get install sqlite3
```

Для установки sqlite3 браузера в Windows надо использовать инсталлятор, который можно скачать здесь: https://sqlitebrowser.org/
В Linux (Ubuntu) надо выполнить следующие команды:

```
$ sudo add-apt-repository -y ppa:linuxgndu/sqlitebrowser
$ sudo apt-get update
$ sudo apt-get install sqlitebrowser
```

SQLite: консоль

Прежде, чем начать работать непосредственно с данными, нам нужно сконфигурировать структуру их хранения (таблицы).
Создадим хотя бы одну таблицу (в консоли sqlite3, .exit – для выхода):

```
sqlite> CREATE TABLE films
...> (id INTEGER PRIMARY KEY NOT NULL, name CHAR(128) NOT NULL, desc TEXT);
```

Удалить таблицу (при необходимости) так же просто:

```
sqlite> DROP TABLE films;
```

После создания необходимых таблиц, мы можем вносить в них данные. Для обозначения основных действий с данными существует специальная аббревиатура — CRUD (create, read, update, delete — «создать, прочесть, обновить, удалить») — акроним, обозначающий четыре базовые функции, используемые при работе с персистентными хранилищами данных.

SQLite: операторы SQL

В соответствии с CRUD в SQL имеются следующие операторы:

- INSERT - оператор языка SQL, который позволяет добавить строку со значениями в таблицу

```
sqlite> INSERT INTO films (name, desc) VALUES ('Cool Film', 'SHORT LONG STORY');
```

- SELECT - оператор запроса в языке SQL, возвращающий набор данных (выборку) из базы данных. Имеет множество опций.

```
sqlite> SELECT * FROM films;
# или:
sqlite> SELECT id, name FROM films WHERE id > 3 ORDER BY id DESC LIMIT 5;
```

- DELETE — в языках, подобных SQL, операция удаления записей из таблицы. Критерий отбора записей для удаления определяется выражением where. В случае, если критерий отбора не определён, выполняется удаление всех записей:

```
sqlite> DELETE FROM films WHERE id = 6;
# или:
sqlite> DELETE FROM films;
```

- UPDATE — оператор языка SQL, позволяющий обновить значения в заданных столбцах таблицы.

```
sqlite> UPDATE films SET name = 'New Film Name' WHERE id = 6;
```

sqlite3

Для работы с SQLite в Python используется библиотека sqlite3. Рассмотрим приемы работы с ней.

```
# Импортируем библиотеку, соответствующую типу нашей базы данных
import sqlite3

# Создаем соединение с нашей базой данных
# В нашем примере у нас это просто какой-то файл базы (без файла – ':memory:')
conn = sqlite3.connect('site.db')

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

# РАБОТАЕМ С БАЗОЙ

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

sqlite3

```
# Делаем SELECT запрос к базе данных, используя обычный SQL-синтаксис
cursor.execute("SELECT Name FROM Artist ORDER BY Name LIMIT 3")

# Получаем результат сделанного запроса
results = cursor.fetchall()
results2 = cursor.fetchall()
print(results)
# [('Leonardo Da Vinchi',), ('Rafael Santi',)]
print(results2)
# []

# Делаем INSERT запрос к базе данных, используя обычный SQL-синтаксис
cursor.execute("INSERT into Artist values (Null, 'Unknown artist') ")

# Если мы не просто читаем, но и вносим изменения в базу данных
# - необходимо сохранить транзакцию
conn.commit()

# Или метод rollback, если хотим отменить изменения
conn.rollback()

# Проверяем результат
cursor.execute("SELECT Name FROM Artist ORDER BY Name LIMIT 3")
results = cursor.fetchall()
print(results)
# [('Leonardo Da Vinchi',), ('Rafael Santi',), ('Unknown artist',)]
```

sqlite3: пример

```
import sqlite3

# Подключаемся к базе данных
# Файл базы данных (если он еще не создан будет создан автоматически
conn = sqlite3.connect('sqlite.db')
print('База данных открыта')

# Создаем таблицу
conn.execute('CREATE TABLE COMPANY'
             ' (ID INT PRIMARY KEY NOT NULL,'
             ' NAME TEXT NOT NULL,'
             ' AGE INT NOT NULL,'
             ' ADDRESS CHAR(50),'
             ' SALARY REAL);')
print('Таблица создана')

# Добавляем данные
conn.execute("INSERT INTO COMPANY (ID, NAME, AGE, ADDRESS, SALARY)"
             "VALUES (1, 'Paul', 32, 'California', 20000.00)")
conn.execute("INSERT INTO COMPANY (ID, NAME, AGE, ADDRESS, SALARY)"
             "VALUES (2, 'Allen', 25, 'Texas', 15000.00)")
conn.execute("INSERT INTO COMPANY (ID, NAME, AGE, ADDRESS, SALARY)"
             "VALUES (3, 'Teddy', 23, 'Norway', 20000.00)")
conn.execute("INSERT INTO COMPANY (ID, NAME, AGE, ADDRESS, SALARY)"
             "VALUES (4, 'Mark', 25, 'Richmond', 65000.00)")
conn.commit()
print('Данные добавлены')

# Читаем данные
cursor = conn.execute('SELECT id, name, address, salary from '
                      'COMPANY WHERE id = 1')
for row in cursor:
    print(row)
```

sqlite3: пример

```
# Изменяем данные
conn.execute('UPDATE COMPANY set salary = 25000.00 WHERE id = 1')
conn.commit()
print('Данные изменены')

# Проверяем, что получилось
cursor = conn.execute('SELECT id, name, address, salary from '
                      'COMPANY WHERE id = 1')
print(cursor.fetchone())

# Удаляем данные
conn.execute('DELETE from COMPANY where id = 2')
conn.commit()
print('Данные удалены')

# Проверяем, что получилось
cursor = conn.execute('SELECT id, name, address, salary from COMPANY')
for row in cursor:
    print(row)

# Изменяем данные
conn.execute('UPDATE COMPANY set salary = 25000.00 WHERE id = 1')
conn.commit()
print('Данные снова изменены')

# Проверяем, что получилось
cursor = conn.execute('SELECT id, name, address, salary from '
                      'COMPANY WHERE id = 1')
print(cursor.fetchone())
```

sqlite3: пример

```
# При необходимости подставить в запрос значения, полученные в коде Python
# НЕ следует использовать строковые операции (например, конкатенацию)
# с текстом запроса!
# Для подстановки параметров в запрос необходимо указать специальный символ
# '?' на местах подстановки в тексте запроса, а параметры передать в кортеже
# в соответствующем порядке в тот же метод execute().
conn.execute("INSERT INTO COMPANY (ID, NAME, AGE, ADDRESS, SALARY)"
             "VALUES (?, ?, ?, ?, ?)", (5, 'Allen', 32, 'Texas', 15000.00))
conn.commit()
print('Новые данные добавлены')

# Проверяем, что получилось
cursor = conn.execute('SELECT id, name, address, salary from COMPANY')
for row in cursor:
    print(row)
conn.execute('DROP TABLE COMPANY')
print('Таблица удалена')

conn.close()
```

sqlite3: пример

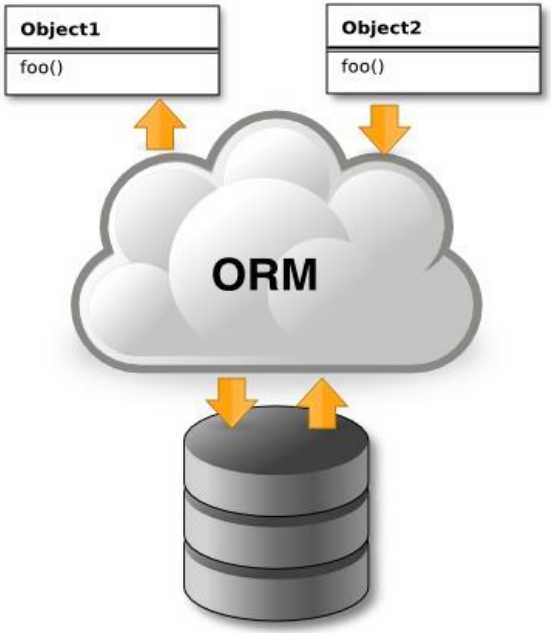
Если запустить получившийся скрипт, на экране можно увидеть следующий результат:

```
База данных открыта
Таблица создана
Данные добавлены
(1, 'Paul', 'California', 20000.0)
Данные изменены
(1, 'Paul', 'California', 25000.0)
Данные удалены
(1, 'Paul', 'California', 25000.0)
(3, 'Teddy', 'Norway', 20000.0)
(4, 'Mark', 'Richmond', 65000.0)
Данные снова изменены
(1, 'Paul', 'California', 25000.0)
Новые данные добавлены
(1, 'Paul', 'California', 25000.0)
(3, 'Teddy', 'Norway', 20000.0)
(4, 'Mark', 'Richmond', 65000.0)
(5, 'Allen', 'Texas', 15000.0)
Таблица удалена
```

ORM

ORM (Object-relational mapping – объектно-реляционное преобразование) - технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

Известные ORM в Python: SQLAlchemy, Django ORM, peewee, PonyORM, SQLObject, Storm, quick_orm и другие.



SQLAlchemy

SQLAlchemy – распространенный инструмент для работы с базами данных. В отличие от многих аналогичных библиотек SQLAlchemy предоставляет не только ORM уровень, но и обобщенный API для написания кода независимого от базы данных без использования SQL.
SQLAlchemy ORM предоставляет метод ассоциирования пользовательских классов Python с таблицами баз данных, и объектов этих классов со строками в соответствующих таблицах. Она включает в себя систему, прозрачно синхронизирующую все изменения в состояниях между объектами и соответствующими строками, равно как и систему для выполнения запросов к базе данных в терминах пользовательских классов и с учетом взаимосвязей этих классов.

sqlalchemy: пример

При использовании ORM конфигурирование начинается с описания таблиц базы данных и определения пользовательских классов, которые будут отображениями этих таблиц.

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String
from sqlalchemy.schema import CreateTable
from sqlalchemy.orm import sessionmaker

# Создаем базу данных
engine = create_engine('sqlite://')  # Либо без файла (в оперативной памяти)
# Либо с файлом, указывая относительный путь к нему: engine = create_engine('sqlite:///mybase.db')
# Либо абсолютный путь: engine = create_engine(r'sqlite:///C:\Users\User\Desktop\mybase.db')

print('База данных создана')

Base = declarative_base()

# Создаем класс, отображающий таблицу
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)
    def __repr__(self):
        return "<User(name='{}', fullname='{}', password='{}')>".format(
                self.name, self.fullname, self.password)
```

sqlalchemy: пример

```
# Создаем схему
Base.metadata.create_all(engine)
print(CreateTable(User.__table__).compile(engine))
print('Таблица создана')

# Создаем объект отображаемого класса
ed_user = User(name='ed', fullname='Ed jones', password='edpassword')

# Открываем сессию для взаимодействия с базой данных
Session = sessionmaker(bind=engine)
session = Session()
print('Сессия открыта')

# Добавляем новый объект в базу
session.add(ed_user)
session.commit()
print('Объект добавлен')

# Проверяем, что получилось
print(session.query(User).filter_by(name='ed').first())

# Добавляем несколько объектов в базу
session.add_all([
    User(name='wendy', fullname='Wendy Williams', password='pwd'),
    User(name='mary', fullname='Mary Contrary', password='qwerty'),
    User(name='fred', fullname='Fred Flinstone', password='123')])
session.commit()
print('Объекты добавлены')

# Проверяем, что получилось
print('Объектов User в базе: {}'.format(session.query(User).count()))
print(session.query(User).all())
```

sqlalchemy: пример

```
# Изменяем запись
ed = session.query(User).filter_by(name='ed').first()
ed.password = 'edsnewpassword'
session.add(ed)
session.commit()

# Проверяем, что получилось
print('Объектов User в базе: {}'.format(session.query(User).count()))
print(session.query(User).all())

# Удаляем запись
session.delete(ed)
session.commit()

# Проверяем, что получилось
print('Объектов User в базе: {}'.format(session.query(User).count()))
print(session.query(User).all())
```

SQLAlchemy содержит множество объектов и функций для покрытия всех опций SQL, в т.ч. ForeignKey, _add, _or (для формирования запросов) и многие другие.

sqlalchemy: пример

Если запустить получившийся скрипт, на экране можно увидеть следующий результат:

```
База данных создана

CREATE TABLE users (
            id INTEGER NOT NULL,
            name VARCHAR,
            fullname VARCHAR,
            password VARCHAR,
            PRIMARY KEY (id)
)


Таблица создана
Сессия открыта
Объект добавлен
<User(name='ed', fullname='Ed jones', password='edpassword')>
Объекты добавлены
Объектов User в базе: 4
[<User(name='ed', fullname='Ed jones', password='edpassword')>, <User(name='wendy', fullname='Wendy
Williams', password='pwd')>, <User(name='mary', fullname='Mary Contrary', password='qwerty')>,
<User(name='fred', fullname='Fred Flinstone', password='123')>]
Объектов User в базе: 4
[<User(name='ed', fullname='Ed jones', password='edsnewpassword')>, <User(name='wendy', fullname='Wendy
Williams', password='pwd')>, <User(name='mary', fullname='Mary Contrary', password='qwerty')>,
<User(name='fred', fullname='Fred Flinstone', password='123')>]
Объектов User в базе: 3
[<User(name='wendy', fullname='Wendy Williams', password='pwd')>, <User(name='mary', fullname='Mary
Contrary', password='qwerty')>, <User(name='fred', fullname='Fred Flinstone', password='123')>]
```

MongoDB

MongoDB – кроссплатформенная документоориентированная база данных. Классифицирована как NoSQL.

MongoDB не требует описания схемы таблиц, мы можем добавлять и удалять поля по мере необходимости. С одной стороны это упрощает разработку, когда мы имеем дело с часто меняющейся структурой данных, но с другой - добавление схемы документов позволяет контролировать ошибки (такие как некорректные типы данных или пропущенные поля), а также определять методы для работы с документами по аналогии с тем, как это делается в ORM технологии.

Основные сущности MongoDB:

Document – запись в коллекции MongoDB и основная единица данных.

Collection – группа документов в MongoDB.

Для низкоуровневой работы с MongoDB можно использовать библиотеку pymongo.

Для работы с MongoDB через объекты Python (по аналогии с ORM) применяется библиотека mongoengine (работает поверх pymongo).

MongoDB

Для установки MongoDB в Windows надо использовать инсталлятор, который можно скачать здесь: https://www.mongodb.com/download-center/community?jmp=docs. Можно установить программу как службу, либо как отдельное приложение.

Для установки MongoDB в Linux (Ubuntu) можно воспользоваться инструкцией (https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/), либо выполнить следующие команды:

```
$ sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
9DA31620334BD75D9DCB49F368818C72E52529D4
$ echo "deb [ arch=amd64 ] https://repo.mongodb.org/apt/ubuntu
trusty/mongodb-org/4.0 multiverse" | sudo tee
/etc/apt/sources.list.d/mongodb-org-4.0.list
$ sudo apt-get update
$ sudo apt-get install -y mongodb-org
```

MongoDB

Для запуска сервера MongoDB в Windows (если программа была установлена как отдельное приложение) надо создать структуру папок C:\data\db и запустить исполняемый файл сервера: C:\Program Files\MongoDB\Server\4.0\bin\mongod.exe

MongoDB

Для запуска сервера MongoDB в Linux (Ubuntu) надо подготовить конфигурационный файл:

```
$ sudo nano /etc/systemd/system/mongodb.service
```

Записать в него настройки:

```
[Unit]
Description=High-performance, schema-free document-oriented database
After=network.target

[Service]
User=mongodb
ExecStart=/usr/bin/mongod --quiet --config /etc/mongod.conf

[Install]
WantedBy=multi-user.target
```

И запустить сервер:

```
$ sudo service mongodb start
```

mongoengine: пример

```
import mongoengine as me

# Подключаемся к базе MongoDB на локальной машине
conn = me.connect('test')
print(conn)

# Объявляем коллекцию
class User(me.Document):
    email = me.StringField(required=True)
    first_name = me.StringField(max_length=50)
    last_name = me.StringField(max_length=50)

    def __repr__(self):
        return ("<User(first_name='{}'), "
                "last_name='{}', "
                "email='{}')>".format(self.first_name,
                                      self.last_name,
                                      self.email))

# Создаем документ
ross = User(email='ross@example.com', first_name='Ross', last_name='Lawley')
ross.save()

# Проверяем, что получилось
print('Документов в базе: {}'.format(User.objects.count()))
```

mongoengine: пример

```
# Делаем запрос
for i in range(5):
    User(email='test@example.com',
         first_name='User'+str(i),
         last_name='Test').save()

# Проверяем, что получилось
print(User.objects.filter(first_name='User3'))
# Двойное нижнее подчеркивание используется для
# задания регулярного выражения
print(User.objects.filter(first_name__startswith='User'))

# Удаляем запись в базе данных
User.objects(first_name__startswith='User').delete()

# Проверяем, что получилось
print('Документов в базе: {}'.format(User.objects.count()))

# Удаляем все записи в базе данных
User.objects.delete()
```

mongoengine: пример

```
MongoClient('localhost', 27017)
Документов в базе: 1
<User(first_name='User3', last_name='Test', email='test@example.com')>
[<User(first_name='User0', last_name='Test', email='test@example.com')>, <User(first_name='User1',
last_name='Test', email='test@example.com')>, <User(first_name='User2', last_name='Test',
email='test@example.com')>, <User(first_name='User3', last_name='Test', email='test@example.com')>,
<User(first_name='User4', last_name='Test', email='test@example.com')>]
Документов в базе: 1
```

Redis

Redis (REmote DIctionary Server) – хранилище элементов типа ключ-значение с поддержкой хэширования и кэширования. Часто определяется как сервер структур данных, т.к. по ключам могут содержаться различные типы данных. Redis обычно держит весь набор данных в оперативной памяти. Когда не требуется долговременное хранение данных, такая способность Redis обеспечивает ему очень высокое быстродействие в сравнении с СУБД, требующими подтверждения (коммита) транзакций.
Для установки Redis в Linux (Ubuntu) можно использовать apt-get:

```
sudo apt-get update
sudo apt-get install redis-server
```

После установки надо отредактировать файл **/etc/redis/redis.conf**, поменяв параметр **supervised no** на **supervised system** и перезапустив Redis.

```
sudo systemctl restart redis.service
```

Для работы с Redis в Python используется библиотека redis.

Redis

Для использования Redis в Windows можно скачать архив с исполняемым файлом серверного приложения отсюда:
https://github.com/dmajkic/redis/downloads
Архив необходимо распаковать, после чего запустить файл redis-server.exe

redis: пример

```
import time
import redis

# Создаем подключение
r = redis.Redis(host='localhost', port=6379, db=0)

# Добавляем элемент - пару ключ-значение
r.set('name', 'John')
# Получаем значение по ключу
print("r.get('name'): ", r.get('name'))
# Определяем тип значения по ключу
print("r.type('name'): ", r.type('name'))
# Удаляем элемент по ключу
r.delete('name')
print("r.get('name'): ", r.get('name'))

# set всегда добавляет значение строкового типа
r.set('my_int', 2)
print("r.get('my_int'): ", r.get('my_int'))
print("r.type('my_int'): ", r.type('my_int'))
# Инкрементируем значение по ключу, которое для данной
# операции интерпретируется как 64-битное знаковое целое
print("r.incr('my_int'): ", r.incr('my_int'))
# Проверяем существование значения по ключу
print("r.exists('my_int'): ", r.exists('my_int'))
```

redis: пример

```
r.set('temp_value', 'value')
# Задаем время жизни ключа (в секундах), по истечении
# которого он будет автоматически удален с сервера
r.expire('temp_value', 5)
# Узнаем оставшееся время жизни ключа
print("r.ttl('temp_value'): ", r.ttl('temp_value'))
print("r.get('temp_value'): ", r.get('temp_value'))

time.sleep(5)
print("After 5 seconds...")

print("r.ttl('temp_value'): ", r.ttl('temp_value'))
print("r.get('temp_value'): ", r.get('temp_value'))
print("r.exists('temp_value'): ", r.exists('temp_value'))

# hashset - хэшсет
# Добавляем пару ключ-значение в хэшсет user:1
r.hset('user:1', 'name', 'John')
# Добавляем еще одну пару ключ-значение в хэшсет user:1
r.hset('user:1', 'email', 'john@gmail.com')
print("r.hget('user:1', 'name'): ", r.hget('user:1', 'name'))
print("r.hkeys('user:1'): ", r.hkeys('user:1'))
print("r.hgetall('user:1'): ", r.hgetall('user:1'))
```

redis: пример

```
# list - список # Добавляем элементы в список
r.rpush('my_list', 'elem1')
r.rpush('my_list', 'elem2')
r.rpush('my_list', 'elem3')
r.rpush('my_list', 'elem4')
print("r.llen('my_list'): ", r.llen('my_list'))
print("r.lindex('my_list', 0): ", r.lindex('my_list', 0))
print("r.lrange('my_list', 1, 3): ", r.lrange('my_list', 1, 3))

# Реализация паттерна издатель-подписчик
p = r.pubsub(ignore_subscribe_messages=True)
p.subscribe('my-chat')
print("p.get_message(): ", p.get_message())
r.publish('my-chat', 'user: Hello!')
while True:
    msg = p.get_message()
    if msg:
        print("p.get_message(): ", msg)
        break
```

redis: вывод на экран

```
r.get('name'):  b'John'
r.type('name'):  b'string'
r.get('name'):  None
r.get('my_int'):  b'2'
r.type('my_int'):  b'string'
r.incr('my_int'):  3
r.exists('my_int'):  1
r.ttl('temp_value'):  5
r.get('temp_value'):  b'value'
After 5 seconds...
r.ttl('temp_value'):  -2
r.get('temp_value'):  None
r.exists('temp_value'):  0
r.hget('user:1', 'name'):  b'John'
r.hkeys('user:1'):  [b'name', b'email']
r.hgetall('user:1'):  {b'name': b'John', b'email': b'john@gmail.com'}
r.llen('my_list'):  0
r.lindex('my_list', 0):  b'elem1'
r.lrange('my_list', 1, 3):  [b'elem2', b'elem3', b'elem4']
p.get_message():  None
p.get_message():  {'type': 'message', 'pattern': None, 'channel': b'my-chat', 'data':
b'user: Hello!'}
```

Практика

1. Написать класс-обертку над SQLite (с возможностями менеджера контекста), которая может на вход принимать строки SQL запросов и возвращать данные в формате json. Класс должен иметь, как минимум, методы select и execute.
2. * Написать скрипт, работающий под SQLite/MySQL/PostgreSQL, который создает 3 сущности: производители, покупатели, товары. Необходимо добавить демо-данные и выполнить следующие выборки:
   - найти всех производителей у которых более 2 товаров по цене 10 долларов и ниже;
   - найти всех покупателей, которые делали заказы, и сгруппировать их по компаниям производителей чьи товары покупались;
   - найти самые популярные товары у каждого производителя и указать сколько таких товаров было куплено;
   - найти всех производителей товаров, которые продавались, с указанием их выручек по каждому виду товара и те, которые еще не продавались