**Модуль 9. Хранение данных вне программы  (4 ак. ч.)**

* Сохранение и восстановление данных. Библиотечный модуль pickle.
* Понятие об объектно-реляционном соответствии.
* Практикум. Применение модуля pickle для хранения и передачи данных.

## Сохранение и восстановление данных. Библиотечный модуль pickle

Модуль pickle предоставляет функции и классы для сериализации и десериализации объектов. В программировании под сериализацией понимают преобразование каких-либо данных в набор байтов, который потом обычно сохраняют в файл или передают по сети. Десериализация - это восстановление объектов из их байтовых представлений.

Часто сериализация используется для сохранения пользовательских данных между разными сессиями работы приложения, обычно игры. Более простой пример - вы работаете в интерактивном режиме и создали список или словарь, который захотите использовать в следующий раз. Для этого с помощью функции dump() модуля pickle вы сохраняете объект в файл, а с помощь load() восстанавливаете в следующий раз.

a = [1, 10, 0, -3, 9]

>>> b = {'a': 1, 'b': 2}

>>> import pickle

>>> f = open('data', 'wb')

>>> pickle.dump(a, f)

>>> pickle.dump(b, f)

>>> f.close()

>>> del a

>>> a

f = open('data', 'rb')

>>> a = pickle.load(f)

>>> a

[1, 10, 0, -3, 9]

>>> c = pickle.load(f)

>>> c

{'a': 1, 'b': 2}

>>> c == b

True

>>> c is b

False

В Python большинство объектов можно сериализовать. Однако не все. Кроме того, следует быть осторожными при сериализации объектов, созданных на основе собственных классов. Сериализуется только объект, но не класс. При десериализации класс импортируется. Поэтому не следует определять классы в том же файле, где будут создаваться и сериализоваться объекты.

Кроме методов dump() и load() есть аналогичные dumps() и loads(), которые работают без файлов. Набор байтов используется в программе.

a = [1, 10, 0, -3, 9]

>>> ab = pickle.dumps(a)

>>> ab

b'\x80\x03]q\x00(K\x01K\nK\x00J\xfd\xff\xff\xffK\te.'

>>> b = pickle.loads(ab)

>>> b

[1, 10, 0, -3, 9]

>>> type(ab)

<class 'bytes'>

Пользоваться pickle лучше только в рамках python-приложения. При обмене данными между разными языками программирования обычно используют модуль json. Также pickle никак не решает вопрос безопасности данных. Поэтому не следует десериализовать данные из неизвестных источников.

Более ранние версии Python могут не понимать протокол сериализации более поздних версий. Для совместимости в функции можно передавать номер протокола.

>>> pickle.DEFAULT\_PROTOCOL

3

>>> pickle.HIGHEST\_PROTOCOL

4

>>> a = [1, 2, 3]

>>> ab = pickle.dumps(a, 2)

>>> b = pickle.loads(ab)

>>> b

[1, 2, 3]

Обратите внимание, что модуль pickle не защищен от ошибочных или злонамеренно созданных данных в соответствии с документацией. Поэтому никогда не отбирайте данные, полученные из ненадежных или не аутентифицированных источников.

Бинарные файлы в отличие от текстовых хранят информацию в виде набора байт. Для работы с ними в Python необходим встроенный модуль pickle. Этот модуль предоставляет два метода:

dump(obj, file): записывает объект obj в бинарный файл file

load(file): считывает данные из бинарного файла в объект

При открытии бинарного файла на чтение или запись также надо учитывать, что нам нужно применять режим "b" в дополнение к режиму записи ("w") или чтения ("r"). Допустим, надо надо сохранить два объекта:

import pickle

FILENAME = "user.dat"

name = "Tom"

age = 19

with open(FILENAME, "wb") as file:

pickle.dump(name, file)

pickle.dump(age, file)

with open(FILENAME, "rb") as file:

name = pickle.load(file)

age = pickle.load(file)

print("Имя:", name, "\tВозраст:", age)

С помощью функции dump последовательно записываются два объекта. Поэтому при чтении файла также последовательно посредством функции load мы можем считать эти объекты. Консольный вывод программы:

Имя: Tom Возраст: 28

Подобным образом мы можем сохранять и извлекать из файла наборы объектов:

import pickle

FILENAME = "users.dat"

users = [

["Tom", 28, True],

["Alice", 23, False],

["Bob", 34, False]

]

with open(FILENAME, "wb") as file:

pickle.dump(users, file)

with open(FILENAME, "rb") as file:

users\_from\_file = pickle.load(file)

for user in users\_from\_file:

print("Имя:", user[0], "\tВозраст:", user[1], "\tЖенат(замужем):", user[2])

В зависимости от того, какой объект мы записывали функцией dump, тот же объект будет возвращен функцией load при считывании файла.

Консольный вывод:

Имя: Tom Возраст: 28 Женат(замужем): True

Имя: Alice Возраст: 23 Женат(замужем): False

Имя: Bob Возраст: 34 Женат(замужем): False

Несколько важных моментов о модуле pickle в python: Протокол pickle специфичен для Python – его межъязыковая совместимость не гарантируется. Это означает, что вы, скорее всего, не сможете передать информацию, чтобы сделать ее полезной на других языках программирования. Также нет гарантии совместимости между различными версиями Python, поскольку не каждая структура данных Python может быть сериализована модулем. По умолчанию используется последняя версия протокола pickle, если вы не измените ее вручную. И последнее, но не менее важное: модуль pickle не защищен от ошибочных или злонамеренно созданных данных в соответствии с документацией.

Поэтому, если у вас есть пользовательский объект, который вам может потребоваться хранить/извлекать, вы можете использовать этот формат:

import pickle

​

class MyClass():

def \_\_init\_\_(self, param):

self.param = param

​

def save\_object(obj):

try:

with open("data.pickle", "wb") as f:

pickle.dump(obj, f, protocol=pickle.HIGHEST\_PROTOCOL)

except Exception as ex:

print("Error during pickling object (Possibly unsupported):", ex)

​

obj = MyClass(10)

save\_object(obj)

​

Если вы запускаете этот скрипт, вы заметите файл под названием data.pickle , который содержит сохраненные данные

Чтобы снова загрузить один и тот же объект, мы могли бы использовать Pickle.Load () используя аналогичную логику.

import pickle

​

class MyClass():

def \_\_init\_\_(self, param):

self.param = param

​

def load\_object(filename):

try:

with open(filename, "rb") as f:

return pickle.load(f)

except Exception as ex:

print("Error during unpickling object (Possibly unsupported):", ex)

​

obj = load\_object("data.pickle")

​

print(obj.param)

print(isinstance(obj, MyClass))

​

Выход

10

True

3. Использование SQLICENCT в качестве постоянного кэша

Если вы найдете использование SQLite3 Слишком утомительно, есть гораздо лучшее решение! Вы можете использовать sqlized Для хранения постоянных данных, и это внутренне использует SQLite3 база данных для обработки хранилища.

Вы должны установить этот пакет, используя PIP:

pip install sqlitedict

​

Единственное, что вы должны иметь в виду, это то, что вам нужно использовать Ключ: значение отображения для хранения/извлечения данных, как и словарь!

ЕСТЬ ПРОТИВОПОКАЗАНИЯ. ПОСОВЕТУЙТЕСЬ С ВРАЧОМ.

Подробнее

suprastin.net

Перейти

favicon

Вот очень простой пример, используя MyClass пример.

from sqlitedict import SqliteDict

​

class MyClass():

def \_\_init\_\_(self, param):

self.param = param

​

def save(key, value, cache\_file="cache.sqlite3"):

try:

with SqliteDict(cache\_file) as mydict:

mydict[key] = value # Using dict[key] to store

mydict.commit() # Need to commit() to actually flush the data

except Exception as ex:

print("Error during storing data (Possibly unsupported):", ex)

​

def load(key, cache\_file="cache.sqlite3"):

try:

with SqliteDict(cache\_file) as mydict:

value = mydict[key] # No need to use commit(), since we are only loading data!

return value

except Exception as ex:

print("Error during loading data:", ex)

​

obj1 = MyClass(10)

save("MyClass\_key", obj1)

​

obj2 = load("MyClass\_key")

​

print(obj1.param, obj2.param)

print(isinstance(obj1, MyClass), isinstance(obj2, MyClass))

​

Выход

10 10

True True

​

Действительно, мы просто загрузили наш объект Python! Если вы заметите, sqlized создаст базу данных cache.sqlite3 автоматически, если оно не существует, а затем используйте его для хранения/загрузки данных.

В этой статье мы смотрели, как мы можем использовать Python для хранения данных по-разному.

Понятие об объектно-реляционном соответствии (ORM)

аббревиатура ORM скрывает «Object-relational mapping», что в переводе на язык Пушкина означает «Объектно-реляционное отображение» и означает «технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования»… т.е. ORM — прослойка между базой данных и кодом который пишет программист, которая позволяет созданые в программе объекты складывать/получать в/из бд.  
Все просто! Создаем объект и кладем в бд. Нужен этот же объект? Возьми из бд! Гениально! НО! Программисты забывают о первой буковке абравиатуры и пхнут в одну и ту же табличку все! Начиная от свойств объектов, что логично, и, заканчивая foreign key, что никакого отношения к объекту не имеет!

И, что самое страшное, многие тонны howto и example пропагандируют такой подход… мне кажется что первопричина кроется в постоянной балансировке между «я программист» и «я архитектор бд», а т.к. ORM плодятся и множатся — голос программиста давлеет над архитекторским. Все, дальше боли нет, только imho.

Что такое ORM? ORM дословно Object Relational Mapping - объектно-реляционное связывание - комплекс программ, позволяющих работать с базами данных, как если бы они были объектами языка программирования, в данном случае Python.

Или иначе - ORM это набор классов, добавляющих еще один уровень абстракции к таблицам, хранящимся в базе данных. Эта абстракция позволяет работать с объектами БД как с объектами используемого языка программирования, не отвлекаясь на их преобразование в язык SQL. ORM целесообразно использовать в больших проектах, использующих БД, однако для повышения скорости выполнения запросов некоторые программисты предпочитают самостоятельно формировать нужные SQL-запросы. Причина в том, что в случае сложных запросов ORM не всегда генерирует оптимальный SQL-запрос к БД, что в случае большой выборки весьма существенно, особенно если выборка идет по нескольким таблицам. Несмотря на существующие недостатки, ORM широко распространены и существуют во многих языках программирования: Java, C#, PHP, Python, C++ и др.

Использовать ORM в ваших проектах или нет - решать Вам, однако я настоятельно рекомендую ознакомиться с этой технологией, так как она широко используется во многих сферах, в том числе в Java Enterprise.

На сегодняшний момент существует несколько ORM для Python, среди них:

SQLAlchemy

Django-ORM

PonyOrm

PeeWee

SQLObject

Наиболее популярной является 1 и 2, однако 3 и 4 являются хорошей альтернативой.

Системы ORM 1 и 2 - это два больших монстра, хотя и позволяют выполнить многое, но как говорит гугл, в этом случае производительность по сравнению с обычным SQL-запросом падает до 3 раз.

Системы 3 и 4 позиционируются как легковесные, не обладают полным функционалом 1 и 2, но его достаточно для выполнения большинства сложных запросов. Стоит отметить, что их производительность по сравнению с 1 или 2 выше.

Для простоты рассмотрим особенности работы с PeeWee.

Для установки пакета с командной строке наберите pip install peewee.

Определение модели

Классы модели, поля и экземпляры сущности все отображаются на концепцию базы данных.

| **Термин** | **Соответствие** |
| --- | --- |
| Класс модели | таблица в БД |
| Экземпляр поля | столбец в таблице |
| Экземпляр сущности | строка в таблице |

При создании проекта с peewee, можно начать с определения модели.

**from** **peewee** **import** \*

# Создаем подключение к БД

db = SqliteDatabase('people.db')

**class** **Person**(Model):

name=CharField()

birthday=DateField()

is\_relative=BooleanField()

# Здесь мы указываем соединение с БД

**class** **Meta**:

database=db

В этом примере мы создаем подключение к базе данных SQLite (файл people.db) и определяем класс модели Person. В этом же классе в подклассе Meta расположена database - переменная соединения с БД.

Внимание! мы назвали нашу модель Person вместо People. Вы должны придерживаться этого правила, даже если таблица будет содержать множество людей, мы всегда назовем класс существительным в единственном числе.

В PeeWee определено много типов столбцов:

|  |  |  |  |
| --- | --- | --- | --- |
| **Field Type** | **Sqlite** | **Postgresql** | **MySQL** |
| CharField | varchar | varchar | varchar |
| FixedCharField | char | char | char |
| TextField | text | text | longtext |
| DateTimeField | datetime | timestamp | datetime |
| IntegerField | integer | integer | integer |
| BooleanField | integer | boolean | bool |
| FloatField | real | real | real |
| DoubleField | real | double precision | double precision |
| BigIntegerField | integer | bigint | bigint |
| SmallIntegerField | integer | smallint | smallint |
| DecimalField | decimal | numeric | numeric |
| PrimaryKeyField | integer | serial | integer |
| ForeignKeyField | integer | integer | integer |
| DateField | date | date | date |
| TimeField | time | time | time |
| TimestampField | integer | integer | integer |
| BlobField | blob | bytea | blob |
| UUIDField | text | uuid | varchar(40) |
| BareField | untyped | not supported | not supported |

Чтобы открыть соединение с БД нужно написать

db.connect()

и хотя в мануале пишут, что это не обязательно, тем не менее это хорошая практика, так как позволяет обнаружить ошибки при соединении.

Определение связей происходит при помощи внешнего ключа

**class** **Pet**(Model):

owner = ForeignKeyField(Person, related\_name='pets')

name = CharField()

animal\_type = CharField()

**class** **Meta**:

database = db # this model uses the "people.db" database

в этом примере создается таблица содержащая внешний ключ, который указывает на первичный ключ таблицы Person. Кроме того, в модель Person добавляется новое поле pets, которое отвечает за выборку всех объектов типа Pet, у которых внешний ключ равен первичному в таблице Person.

создание таблиц в БД происходит следующим образом:

db.create\_tables([Person, Pet])

сохранение экземпляра класса в БД происходит по команде:

save()

Пример:

grandma = Person.create(name='Grandma', birthday=date(1935, 3, 1), is\_relative=True)

grandma.save()

Подробная документация находится [здесь](http://docs.peewee-orm.com/en/latest/peewee/querying.html)

**Прежде чем бросаться и писать ORM обертку для таблиц сущностей, сначала надо спроектировать логическую структуру БД или ХД.** Установить все зависимости и способы взаимодействия между сущностями.

ORM SQLAlchemy

Работа с этой ORM начинается с импорта элементов из пакета sqlalchemy.

Для подключения к БД мы создаем объект Engine, который создает связку пула подключения и диалекта конкретной БД.

Далее нужно создать базовый класс для объявления сущностей наших данных

Потом мы описываем связываемые таблицы через наследование от класса Base.

**from** **sqlalchemy** **import** create\_engine

engine = create\_engine('sqlite:///:memory:', echo=True)

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

**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** **\_\_repr\_\_**(self):

**return** "<User(name='%s', fullname='%s', password='%s')>" % (

self.name, self.fullname, self.password)

В этом примере мы определили связку сущности User и таблицы users.

Для создания экземпляра сущности просто создайте объект класса User.

u=User(name='Isac', fullname='Newton', password='123')

Указывать параметры при вызове конструктора не обязательно, присвоить значения полям экземпляра сущности можно и после.

u=User()

u.name='Isac'

u.fullname='Newton'

u.password='123'

Для того, чтобы связаться с базой данных и начать с ней диалог, создадим класс сессия.

**from** **sqlalchemy.orm** **import** sessionmaker

Session = sessionmaker(bind=engine)

# и создадим экземпляр этого объекта

session=Session()

теперь мы можем сохранять наши экземпляры сущностей в базу.

ed\_user = User(name='ed', fullname='Ed Jones', password='edspassword')

session.add(ed\_user)

session.add(u)

session.commit()

Обратите внимание, что после commit() поле id примет автоматически инициализированного значения. (id - первичный ключ, автоинкрементный)

### **Какие плюсы использования ORM? 👍**

1. Вы в любом случае можете писать на том языке, которым уже пользуетесь. Если честно, мы, вероятно, не лучшие в написании операторов SQL. SQL - смехотворно мощный язык, но большинство из нас не часто пишут на нем. Однако мы, как правило, гораздо более бегло говорим на том или ином языке, и возможность использовать эту беглость - это потрясающе!
2. Он абстрагируется от системы базы данных, поэтому переход с MySQL на PostgreSQL или любой другой вариант, который вы предпочитаете, был легким.
3. В зависимости от ORM вы получаете множество расширенных функций из коробки, таких как поддержка транзакций, пул соединений, миграции, начальные значения, потоки и всевозможные другие полезности.
4. Многие из написанных вами запросов будут работать лучше, чем если бы вы написали их сами.

### **Какие недостатки использования ORM? 👎**

1. Если вы хорошо разбираетесь в SQL, вы, вероятно, сможете получить более эффективные запросы, написав их самостоятельно.
2. Есть накладные расходы, связанные с изучением того, как использовать любую данную ORM.
3. Первоначальная настройка ORM может быть головной болью.
4. Разработчику важно понимать, что происходит под капотом. Поскольку ORM могут служить костылем, позволяющим избежать понимания баз данных и SQL, они могут сделать вас более слабым разработчиком в этой части стека.