Индексно-последовательные файлы

сериализация

Пример

```
>>> import dbm
>>> db=dbm.open('storage','c')
>>> db['1']='qq'
>>> db['2']='ww'
>>> db.keys()
[b'2', b'1’]

$ ls -l storage.db
-rw-r--r--  1 fat  _lpoperator  16384 28 мар 22:23 storage.db
$ file storage.db
storage.db: Berkeley DB 1.85 (Hash, version 2, native byte-order)
```

Что это такое?

- Хранимая таблица формата ключ-значение
- Классическая библиотека dbm – К. Томпсон, AT&T 1979
- Более поздние варианты:
  - Ndbm: In 1986 Berkeley produced ndbm (standing for New Database Manager). This added support for having multiple databases open concurrently.
  - Sdbm: Some versions of Unix excluded ndbm due to licensing issues, so in 1987 Ozan Yigit produced this public-domain clone.
  - BDB: 1991 successor to ndbm by Sleepycat Software (now Oracle) created to get around the AT&T Unix copyright on BSD.
  - GDBM (GNU dbm): A Free/Libre version written by Philip A. Nelson for the GNU project. It added support for arbitrary-length data in the database: previously all data had a fixed maximum length. The latest version was released on 11 March 2017
  - tdb (trivial database library): developed and used internally within the Samba suite, implements an API inspired by GDBM but also supports multiple writers, released under the LGPL license.
  - tdbm: a version of ndbm with atomic transactions, in-memory databases, and other extensions, released under a BSD-style open source license.
  - MDBM: Ndbm work-alike hashed database library based on sdbm which is based on Per-Aake Larson's Dynamic Hashing algorithms.
  - Да тыщи их

Модуль dbm в Python 3

- Часть стандартной библиотеки (работает без pip)
- Поддерживает форматы ndbm и gdbm
- Открытая база похожа на стандартный словарь Python
  - В качестве ключей и значений только байтовые строки
- Неплохая альтернатива РДБМС (особенно sqlite) для простых приложений

Только байтовые значения

>>> db['1']="В чащахъ юга жилъ бы цитрусъ, да но фальшивый экземпляръ"

- Не очень хорошая идея (результат зависит от настроек Python)

Только байтовые значения

>>> db['1']="В чащахъ юга жилъ бы цитрусъ, да но фальшивый экземпляръ"

- Не очень хорошая идея (результат зависит от настроек Python)

>>> db['1']="В чащахъ юга жилъ бы цитрусъ, да но фальшивый экземпляръ".encode("utf-8")

>>> db['1']

b'\xd0\x92 \xd1\x87\xd0\xb0\xd1\x89\xd0\xb0\xd1\x85\xd1\x8a
\xd1\x8e\xd0\xb3\xd0\xb0 \xd0\xb6\xd0\xb8\xd0\xbb\xd1\x8a \xd0\xb1\xd1\x8b
\xd1\x86\xd0\xb8\xd1\x82\xd1\x80\xd1\x83\xd1\x81\xd1\x8a, \xd0\xb4\xd0\xb0
\xd0\xbd\xd0\xbe
\xd1\x84\xd0\xb0\xd0\xbb\xd1\x8c\xd1\x88\xd0\xb8\xd0\xb2\xd1\x8b\xd0\xb9
\xd1\x8d\xd0\xba\xd0\xb7\xd0\xb5\xd0\xbc\xd0\xbf\xd0\xbb\xd1\x8f\xd1\x80\xd1\x8a'

>>> db['1'].decode("utf-8")

'В чащахъ юга жилъ бы цитрусъ, да но фальшивый экземпляръ'

Как хранить структурированные данные

- Использовать сериализацию
- Стандартный модуль pickle (бинарные данные, Python-only)
  - The pickle module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.
- JSON (текст, human-readable, поддерживается многими языками)
  - Сам по себе умеет сериализовать скаляры, списки, туплы и словари
  - Предоставляет обработчики для поддержки других типов
  - Но обработчики надо писать вам самим
- JSON – строка (не байты),
  - *ensure_ascii=True*

Как же сериализовать и десериализовать объект в JSON?

- Сериализовать легко

```
>>> json.dumps(q.__dict__)
{"_d": "qq"}
```

- А как обратно?

```
>>> class Payload(object):
... def __init__(self, j):
...         self.__dict__ = json.loads(j)
```

- Не очень хорошая идея

Обработчик десериализации

```
class Payload(object):
        def __init__(self, action, method, data):
                self.action = action
                self.method = method
                self.data = data
import json
def as_payload(dct):
        return Payload(dct['action'], dct['method'], dct['data’])
payload = json.loads(message, object_hook = as_payload)
```

namedtuple

```
>>> Point = namedtuple('Point', ['x', 'y’])
>>> p = Point(11, y=22) # instantiate with positional or keyword arguments
>>> p[0] + p[1] # indexable like the plain tuple (11, 22)
33
>>> x, y = p # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y # fields also accessible by name
33
```

namedtuple

- Альтернатива data object
- Похожи на объекты JavaScript (ЕВПОЧЯ)
- Проще описывать, чем класс

>>> Point = namedtuple('Point', ['x', 'y’])

- Немутабельные
- Есть ограничения по именам атрибутов (зарезервированные слова, подчерки)
- От них можно наследовать обычные классы

Namedtuple и JSON

```
data = '{"name": "John Smith", "hometown": "New York"}’
x = json.loads(data, object_hook=lambda d: namedtuple('X’,
        d.keys())(*d.values()))
print x.name, x.hometown
```

- Воспроизведет структуру JSON, какова бы она ни была

Namedtuple и JSON по-культурному

```
data = '{"name": "John Smith", "hometown": "New York"}’
Dobject=namedtuple(‘Dobject’, [‘name’, ‘hometown’])
x=Dobject(**json.loads(data))
```

JSONPICKLE

```
>>> import jsonpickle
>>> class Q:
...         def __init__(_self, d):
...             _self._d=d
>>> q=Q('qq')
>>> jsonpickle.encode(q)
{"py/object": "__main__.Q", "_d": "qq"}
>>> jsonpickle.encode(q, unpicklable=False)
{"_d": "qq"}
```

Еще про jsonpickle

- Не сохраняет код объектов, только данные
- Требует, чтобы класс с таким именем был определен в контексте, где вы зовете decode (бинарный pickle ведет себя аналогично)
- Класс должен быть определен на уровне модуля
- Разумеется, методы объекта возьмутся из класса, а не из JSON
- Конструктор при декодировании не исполняется
- Есть методы для управления процедурой упаковки, аналогичные pickle (читайте документацию)