lmdbdict

Release 0.3

May 14, 2021

Contents

CHAPTER 1

How to use lmdbdict

The purpose of this package is to make lmbd easier to use. I simply write a high-level wrapper for it, so that you can write and read it just like a python dict.

1.1 Install

```
pip install lmdbdict
```

or build from master:

```
pip install git+https://github.com/ruotianluo/lmdbdict.git
```

[1]:
```
!pip install lmdbdict
```

```
Requirement already satisfied: lmdbdict in /Users/ruotianluo/github/lmdbdict (0.2)
Requirement already satisfied: lmdb in /Users/ruotianluo/anaconda3/lib/python3.7/site-
↪packages (from lmdbdict) (0.98)
```

1.2 Basic usage

First, import it:

[2]:
```
from lmdbdict import lmdbdict
```

You can instantiate an empty lmdbdict in a "python file" way.

[3]:
```
lmdbpath = 'abc.lmdb'
d = lmdbdict(lmdbpath, mode='w')
```

Similar to files, we support two modes here, read only('r') and write('w'). Under read only, you will not able to change the contents of the lmdbdict. Under write mode, if the file doesn't exist, a new file will be created; if the file already

exists, you will be writing upon the saved content (It's more like 'a' mode instead 'w' mode in the python file context). You will also be able to read under write mode.

After this point, it will behave similar as a python dict.

[4]:
```
# Assign key value pairs
d[1] = 2
d[2] = 3
```

[5]:
```
# get keys
print(d.keys())
```

```
[1, 2]
```

[6]:
```
# get value
d[2]
```

[6]:
```
3
```

[7]:
```
# deletion
del d[1]
```

[8]:
```
# values() is not currently supported
print(d.values())
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
<ipython-input-8-f3ee7ba2c5f2> in <module>
      1 # values() is not currently supported
----> 2 print(d.values())

~/github/lmdbdict/lmdbdict/lmdbdict.py in values(self)
    193
    194     def values(self):
--> 195         raise NotImplementedError
    196
    197     def items(self):

NotImplementedError:
```

Like files, we also expose `flush` function, to let the users safely write the data to the disk. You can only `flush` under write mode.

[9]:
```
d.flush()
```

Delete the lmdbdict instance will also automatically flush if under write mode.

[10]:
```
del d
```

Open the above lmdb in read mode.

[11]:
```
d = lmdbdict(lmdbpath, mode = 'r')
```

[12]:
```
d[2]
```

[12]:
```
3
```

[13]:
```
# No write under read mode
d[2] = 4
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-13-bb30fe0eceb5> in <module>
      1 # No write under read mode
----> 2 d[2] = 4

~/github/lmdbdict/lmdbdict/lmdbdict.py in __setitem__(self, key, value)
    181
    182     def __setitem__(self, key, value):
--> 183         assert self.mode == 'w', 'can only write item in write mode'
    184         # in fact even key is __len__ it should be fine, because it's dumped␣
 ↪in pickle mode.
    185         assert key not in ['__keys__'], \

AssertionError: can only write item in write mode
```

1.3 Customize the dumps and load functions

The lmdb can only save bytes. To support key/values of any type, we use pickle loads and dumps by default to convert the objects into bytes and back.

However, in many cases, pickle may not be the optimal solution if you know what type your keys and values are. So we support customized dumps and loads. You can feed any function you like or use the options we provide.

Note that, only use this if you know what's going on. First, the dumps needs to output a bytes object. Second, the loads has to be exact invert of dumps, otherwise you can't retreive the same thing. Thirdly, you need to make sure the dumps and loads work for your keys and values.

Here we provide a simple example:

[14]:
```
loads_func = dumps_func = lambda x: x
lmdbpath = 'abcd.lmdb'
d = lmdbdict(lmdbpath, 'w',
             key_dumps=dumps_func,
             key_loads=loads_func,
             value_dumps=dumps_func,
             value_loads=loads_func)
```

The loads and dumps here are identical function, so the keys and values have to be bytes.

[15]:
```
d[b'1'] = b'2'
```

[16]:
```
d.keys()
```

[16]:
```
[b'1']
```

The dumps and loads will also be saved in the lmdb file. When you open an existing lmdb file (generated by lmdbdict), you don't need to specify the dumps and loads; they will be loaded from the lmdb file.

[17]:
```
del d
d = lmdbdict(lmdbpath, 'r')
d[b'1']
```

[17]:
```
b'2'
```

Instead of raw functions, you can also feed the method names that are already implemented by me. The current options include `identity`, `pyarrow`, `ascii`, `utf8`, etc.

For example, if you know you keys and values are str, you can then use `ascii` or `utf8`, depending on if your string includes unicode characters. (It's always safer to use `utf8`).

[18]:
```
lmdbpath = 'abcde.lmdb'
d = lmdbdict(lmdbpath, 'w',
             key_dumps='utf8',
             key_loads='utf8',
             value_dumps='utf8',
             value_loads='utf8')
```

or equivalently

[19]:
```
# d = lmdbdict(lmdbpath, 'w',
#              key_method='utf8',
#              value_method='utf8')
```

[20]:
```
d['1'] = '2'
```

CHAPTER 2

API Documentation

2.1 lmdbdict

class lmdbdict.lmdbdict.**lmdbdict**(*lmdb_path*, *mode='r'*, *key_method=None*, *value_method=None*, *key_dumps=None*, *key_loads=None*, *value_dumps=None*, *value_loads=None*, *unsafe=False*, *readahead=False*)

Bases: object

__init__(*lmdb_path*, *mode='r'*, *key_method=None*, *value_method=None*, *key_dumps=None*, *key_loads=None*, *value_dumps=None*, *value_loads=None*, *unsafe=False*, *readahead=False*)

Args: value/key_dumps/loads: can be picklable functions or str or None if None: then default pickle if 'identity' then func = lambda x: x if saved in the db, then use what's in db unsafe: if True, you can getitem by the key even the key is not in the self._keys. readahead: for lmdb reader, only make sense when mode='r'

keys()

__getstate__()

Make it pickable

values()

items()

update(*d*)

flush()

sequential_iter()

lmdbdict.lmdbdict.**LMDBDict**

alias of *lmdbdict.lmdbdict.lmdbdict*

2.2 lmdbdict.utils

lmdbdict.utils.**picklable_wrapper**(*obj*)
Wrap the object with PicklableWrapper only if it's not natively picklable Note: it's not intended to be run a lot of times

lmdbdict.utils.**loads_either**(*cloudpickle_out*, *pickle_out*)
If cloudpickle dumps is available then load with cloudpickle first, because cloudpickle is safer to use for callabels then pickle. If not availble, then resort to pickle.

class lmdbdict.utils.**PicklableWrapper**(*obj*)
Bases: object

Wrap an object to make it more picklable, note that it uses heavy weight serialization libraries that are slower than pickle. It's best to use it only on closures (which are usually not picklable). This is a simplified version of https://github.com/joblib/joblib/blob/master/joblib/externals/loky/cloudpickle_wrapper.py RT change: we save both cloudpickle and pickle dumps. This is for the case that sender and receiver may not have the same enviroment. cloudpickle is preferred because it's better for callables.

2.3 lmdbdict.methods

lmdbdict.methods.**identity**(*x*)

lmdbdict.methods.**ascii_encode**(*x*)

lmdbdict.methods.**ascii_decode**(*x*)

lmdbdict.methods.**utf8_encode**(*x*)

lmdbdict.methods.**utf8_decode**(*x*)

lmdbdict.methods.**pa_dumps**(*x*)

lmdbdict.methods.**pa_loads**(*x*)

Python Module Index

l

Index

Symbols

A

F

I

K

L

P

S

U

V