python-pure-cdb Documentation

Release 2.2.0

dw, bbayles

Dec 09, 2018

Contents

The *python-pure-cdb* package (pure-cdb on PyPI) is a Python library for working with D.J. Bernstein's "constant datbases." It supports both Python 2.7 and Python 3.

In addition to being able to read and write the database files produced by other *cdb* tools, *python-pure-cdb* can produce and consume "64-bit" constant databases that don't have the usual 4 GiB restriction.

For more information on constant databases, see djb's page and Wikipedia.

The documentation for this package is available at https://python-pure-cdb.readthedocs.io.

CHAPTER 1

Contents

1.1 Getting started

1.1.1 Installation

Install the library with pip:

```
pip install pure-cdb
```

Once the library is installed, import *cdblib* to use it.

1.1.2 Reading existing cdb files

cdblib.Reader can query an existing database.

Pass it a *bytes*-like object of the file's contents to start:

```
>>> import cdblib
>>> with open('info.cdb', 'rb') as f:
...     data = f.read()
>>> reader = cdblib.Reader(data)
```

Reader instances implement a *dict*-like interface. To retrieve everything stored in the database, use the *.iteritems()* method.

```
>>> for key, value in reader.iteritems():
...     print('+{},{}:{}->{}'.format(len(key), len(value), key, value))
```

To retrieve the first value stored at a key, use the *.get()* method.

```
>>> reader.get(b'some_key')
b'some_value'
```

Note that all keys and values are *bytes* objects (*str* on Python 2). For more information, see the library documentation.

For "64-bit" database files, use *cdblib.Reader64* instead of *cdblib.Reader*.

1.1.3 Writing new cdb files

cdblib.Writer can create a new database.

Pass it a file-like object (opened in binary write mode) to start. Then write to the database with the *.put()* method.

```
>>> import cdblib
>>> with open('/tmp/new.cdb', 'wb') as f:
...    with cdblib.Writer(f) as f:
...        writer.put(b'key', b'value')
```

As with the reader class, all keys and values are *bytes* objects (*str* on Python 2).

For "64-bit" database files, use *cdblib.Writer64* instead of *cdblib.Writer*.

1.2 Library reference

1.2.1 The *Reader* classes

cdblib.Reader reads standard "32-bit" cdb files, such as those produced by the *cdbmake* CLI tool. *cdblib.Reader64* reads "64-bit" cdb files, which can be produced by this package.

The *Reader* classes take one positional argument, a *bytes*-like object with a database's content:

```
>>> import cdblib
>>> with open('info.cdb', 'rb') as f:
...     data = f.read()
>>> reader = cdblib.Reader(data)
```

An *mmap* object can be used to avoid reading an entire database into memory - see below.

Retrieving data

The *.items()* method returns a list of *(key, value)* tuples representing all of the records stored in the database (in insertion order). Note that a single key can have multiple values associated with it.

```
>>> reader.items()
[(b'k1', b'v1'), (b'k2', b'v2a'), (b'k2', b'v2b')]
```

The *.iteritems()* method is like *.items()*, but it returns an iterator over the items rather than a list.

The *.keys()* method returns a list of the keys stored in the database (in insertion order). The *.iterkeys()* method returns an iterator over the keys. Note that keys will be repeated if a single key has multiple values associated with it.

The *.values()* method returns a list of the values stored in the database (in insertion order). The *.itervalues()* method returns an iterator over the values.

Calling *len()* on a *Reader* instance returns the number of records (key-value pairs) stored in the database.

```
>>> len(reader)
3
```

The *in* operator can be used to test whether a key is present in the database.

```
>>> b'k1' in reader
True
>>> b'k3' in reader
False
```

The *.get()* method returns the first value in the database for *key*. If the key isn't in the database, *None* will be returned. To use a different default value, use the *default* keyword:

```
>>> reader.get(b'k2')
b'v2a'
>>> reader.get(b'missing')
None
>>> reader.get(b'missing', default=b'fallback')
b'fallback'
```

The *.gets()* method returns an iterator over all the values associated with *key*.

```
>>> list(reader.gets(b'k2'))
[b'v2a', b'v2b']
```

Reader instances also support dict-like retrieval of the first value associated with *key*. *KeyError* will be raised if the requested key isn't in the database.

```
>>> reader[b'k2']
b'v2a'
>>> reader[b'missing2']
KeyError: b'missing'
```

Note that the values retrieved by the *.get()* and *.gets()* methods are *bytes* objects.

If the values in the database represent integers, you can retrieve them as Python *int* objects with the *.getint()* and *.getints()* methods.

```
>>> reader.get(b'key_with_int_value')
b'1'
>>> reader.getint(b'key_with_int_value')
1
```

Similarly, the *.getstring()* and *.getstrings()* methods will retrieve the values as *str* objects.

```
>>> reader.get(b'key_with_str_value')
b'text data'
>>> reader.getstring(b'key_with_str_value')
'text data'
```

You may specify an encoding with the *encoding* keyword argument.

```
>>> reader.get(b'fancy_a_or_f')
b'\xc4'
>>> reader.getstring(b'fancy_a_or_f', encoding='cp1252')
'Ä'
>>> reader.getstring(b'fancy_a_or_f', encoding='mac-roman')
'ƒ'
```

Encoding and strict mode

Database keys are stored as *bytes* objects. By default, *Reader* instances will attempt to convert text keys (*str* on Python 3, *unicode* on Python 2) and integer keys (*int* on Python 3, *int* and *long* on Python 2) automatically.

```
>>> reader.get(b'1')  # Binary key
b'value_for_1'
>>> reader.get('1')  # Text key
b'value_for_1'
>>> reader.get(1)  # Integer key
b'value_for_1'
```

To disable this behavior, pass *strict=True* when creating the *Reader* instance. This will increase read performance, and is useful when you want to deal with *bytes* keys only.

```
>>> import cdblib
>>> with open('info.cdb', 'rb') as f:
...     data = f.read()
>>> reader = cdblib.Reader(data, strict=True)
>>> reader.get(b'1')  # Binary key
b'value_for_1'
>>> reader.get(1)
...
TypeError: key must be of type 'bytes'
```

Limiting memory usage

To avoid having to read a whole database into memory, use *cdblib.Reader* (or *cdblib.Reader64*) with *mmap.mmap*.

```
>>> from mmap import mmap, ACCESS_READ
... from cdblib import Reader
...
... with open('info.cdb', 'rb') as f:
...     with mmap(f.fileno(), 0, access=ACCESS_READ) as m:
...         reader = Reader(m)
...         reader.items()
```

See the Python docs for more information on *mmap*.

1.2.2 The *Writer* classes

cdblib.Writer produces standard “32-bit” cdb files, which should be readable by other *cdb* tools like *cdbget* and *cdbdump*. *cdblib.Writer64* produces “64-bit” cdb files, which can be read by this package.

The *Writer* classes take one positional argument, a file-like object opened in binary mode.

```
>>> import cdblib
...
... with open('info.cdb', 'wb') as f:
...     writer = cdblib.Writer(f):
...     writer.put(b'k1', b'v1a')
...     writer.finalize()
```

Writer instances don't create readable databases until their *.finalize()* method is called. You should use them as a context manager wherever possible - this ensures that *.finalize()* is called.

```
>>> with open('info.cdb', 'wb') as f:
...     with cdblib.Writer(f) as writer:
...         writer.put(b'k1', b'v1a')
```

Storing data

The *.put()* method is used to create a database record for a binary key and a binary value.

```
>>> import io
>>> import cdblib
>>> f = io.BytesIO()  # Use an in-memory database
>>> writer = cdblib.writer(f)
>>> writer.put(b'k1', b'v1a')
```

The *.puts()* method adds multiple binary values at the same key.

```
>>> writer.puts(b'k2', [b'v2a', b'v2b'])
```

To store integer values, use *.putint()* or *.putints()*.

```
>>> writer.putint(b'key_with_int_values', 1)
>>> writer.putints(b'key_with_int_values', [2, 3])
```

To store text data, use *.putstring()* or *.putstrings()*, with an optional *encoding* keyword argument. The default encoding is *'utf-8'*.

```
>>> writer.putstring(b'fancy_a', 'Ä')  # stores b'\xc3\x84'
>>> writer.putstring(b'fancy_a', 'Ä', encoding='cp1252')  # stores b'\xc4'
>>> writer.putstrings(b'boring_a', ['a', 'A'])
```

As above, don't forget to call *.finalize()* to write the database to disk if you're not using a context manager.

```
>>> writer.finalize()
```

Encoding and strict mode

Database keys are stored as *bytes* objects. As with *Reader* instances, *Writer* instances will attempt to convert text keys and integer keys automatically.

To disable this behavior, pass *strict=True* when creating the *Writer* instance. This will increase write performance, and is useful when you want to deal with *bytes* keys only.

1.2.3 Advanced usage

Alternate hash functions

By default *python-pure-cdb* will use the standard cdb hash function described on djb's page.

You can substitute in your own hash function when using a *Writer* instance, if you're so inclined. This will of course require you to use the same hash function when reading the database.

```
>>> import io
... import zlib
...
... import cdblib
...
...
... def custom_hash(x):
...     return zlib.adler32(x) & 0xffffffff
...
...
... with io.BytesIO() as f:
...     with cdblib.Writer(f, hashfn=custom_hash) as writer:
...         writer.put(b'k1', b'v1a')
...         writer.puts(b'k2', [b'v2a', b'v2b'])
...
...     reader = cdblib.Reader(f.getvalue(), hashfn=custom_hash)
...     reader.items()
[(b'k1', b'v1a'), (b'k2', b'v2a'), (b'k2', b'v2b')]
```

C extension hash function

When using CPython, you can build a C Extension that speeds up using the cdb hash function.

Set the *ENABLE_DJB_HASH_CEXT* environment variable when executing *setup.py* to enable the extension:

```
$ ENABLE_DJB_HASH_CEXT=1 python setup.py install
```

1.3 Command line tools

The *python-pure-cdb* package contains Python implementations of the cdbmake and cdbdump programs.

python-pure-cdbmake should be able to create databases that are compatible with other implementations, including the standard one. It can also create "64-bit" databases that don't have the usual 4 GiB restriction.

Similarly, *python-pure-cdbdump* should be able to read databases produced by other implementations, including the standard one. It can also read the "64-bit" databases produced by this package.

1.3.1 *python-pure-cdbmake*

This utility creates a database file from text records using the following format:

```
+klen,dlen:key->data
```

Where:

- *klen* is the length of *key* (in bytes)
- *dlen* is the length of *data* (in bytes)
- *key* can be any string of characters
- *data* can be any string of characters

Each record must end with a newline character. For example:

```
+1,2:a->bb
+2,1:aa->b
```

python-pure-cdbmake reads these records from stdin. When invoking the utility, you have to specify two file paths:

- The first (*cdb*) is the ultimate location of the database.
- The second (*cdb.tm*) is a temporary location to use when creating the database. It will be moved to the ultimate location after completion.

```
$ <records_file.txt python-pure-cdbmake ~/records_db.cdb /tmp/records_db.tmp
```

Use the *-64* switch to enable "64-bit" mode, which can write larger database files at the expense of compatibility with other *cdb* packages.

1.3.2 *python-pure-cdbdump*

This utility creates a text export of the contents of a database file.

The output format is the same as the one used by *python-pure-cdbmake* for input - see above.

python-pure-cdbdump reads the database from stdin and prints to stdout.

```
$ <~records_db.cdb python-pure-cdbdump
+1,2:a->bb
+2,1:aa->b
```

Use the *-64* switch to read databases created by this package using "64-bit" mode.

1.4 Version history

- **Version 2.2.0**
  - Added non-*strict* mode for convenience when using non-binary keys.
  - API docs are now available at ReadTheDocs.
- **Version 2.1.0**
  - Python 3 support
  - *Writer* and *Writer64* can now act as context managers.
  - A Python implementation of *cdbdump* (*python-pure-cdbdump*) is now included.
  - The Python implementation of *cdbmake* was renamed *python-pure-cdbmake* and some bugs were fixed.

1.5 Development information

Development for *python-pure-cdb* takes place on GitHub.

1.5.1 Contributing

To file a bug report or make a suggestion, please create a GitHub issue.

To contribute a patch, please create a GitHub pull request.

1.5.2 Python version support

python-pure-cdb supports the versions of Python currently being maintained by the PSF. If you find a bug when using an older version, feel free to file an issue about it, but note that it might not get fixed.

1.5.3 License

This project uses the MIT License.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search