pysophia Documentation

Release 0.1

Michaël Meyer

January 25, 2016

Contents

This package provides Python bindings for sophia, a ligthweight DBM-like database. Available operations on a database are inserting a key-value pair, deleting it, or retrieving a value given its key. It is also possible to traverse the records of a database, in ascending or descending order.

CHAPTER 1

Installation

First install libsophia using this script (to be run from the source package directory if you have the source distribution, or from */tmp* or similar). Then download the bindings (preferably from Github), and install them with:

```
python setup.py install
```

If you want to check how the library performs, look at the benchmarks on the website of the author, and run the script `bench.py` located in the `tests` directory of this package, which will give you an idea of what performance you can expect from the module on your specific hardware.

CHAPTER 2

Contents

2.1 Tutorial

All examples below assume you're using Python 2. To run them under Python 3, you should replace all the `"strings"` expressions by `b"byte strings"`.

2.1.1 Basics

A database consists in a directory containing a set of files, which is represented in Python by the *sophia.Database* class. Databases are opened and closed respectively with the `Database.open()` and `Database.close()` methods:

```
import sophia

# create the object
db = sophia.Database()

# open the database, creating it if it does'nt exist yet.
db.open("mydb")

# close it
db.close()
```

Database objects are implicitly closed when they go out of scope, but you should not rely on this behaviour, and always close them explicitly when you're done with them.

Database-related errors all raise a `sophia.Error` exception. You can catch it with typical `try... except... finally` statements:

```
db = sophia.Database()
try:
    db.open("mydb")
    # do something with `db`
except sophia.Error as e:
    print("Error: %s" % e)
finally:
    db.close() # this has no effect if the db is not opened
```

The *sophia.Database* object only deals with bytes (named `str` under Python 2, `bytes` in Python 3). Transparent data serialization is done by the `sophia.ObjectDatabase` class, for which see below.

2.1.2 Storing and deleting records

Records are added with the `Database.set()` method, and deleted with `Database.delete()`:

```
# create a database
db = sophia.Database()
db.open("actresses")

# add some records
db.set("Audrey Hepburn", "Breakfast at Tiffany's")
db.set("Grace Kelly", "To Catch a Thief")

# update a record
db.set("Audrey Hepburn", "War and Peace")

# delete a record
db.delete("Audrey Hepburn")
```

When using these functions, database modifications are performed atomically, which adds some overhead. If you want to store several records at once, you can use explicit transactions. The added items will then be kept into memory until the transaction is committed or aborted, at which time they will be saved or left away, respectively. If a transaction is not committed and the underlying database object is closed, all modifications will be lost.

Transactions are really easy to perform:

```
# start a transaction
db.begin()

# add some items, remove some others
db.set("Scarlett Johansson", "The Black Dahlia")
db.set("Uma Thurman", "Pulp Fiction")
db.delete("Grace Kelly")

# save the changes
db.commit()

# make another transaction
db.begin()
db.set("Nicole Kidman", "Shakespeare in Love")
db.set("Gwyneth Paltrow", "Dogville")

# oops, interverted the films names, so abort the transaction
db.rollback()
```

2.1.3 Retrieving records

Records can be retrieved by using the `Database.get()` method, and checked for existence with the `Database.contains()` method.

```
>>> db.get("Scarlett Johansson")
"The Black Dahlia"
>>> db.contains("Scarlett Johansson")
True
>>> db.contains("Nicole Kidman") # we just aborted the transaction up there
False
```

If a second argument is given to `Database.get()`, it will be returned as value if the key is not in the database. The default is to return *None* when a key is missing.

```
>>> print(db.get("Gwyneth Paltrow"))
None
>>> db.get("Gwyneth Paltrow", "A perfect number")
"A perfect number"
```

2.1.4 Traversing records

Records can be traversed in order with the `Database.iterkeys()`, `Database.itervalues()`, or `Database.iteritems()` methods, which yield respectively the keys, the values, or the pairs of (key, value) in the database. These methods take two optional arguments: the key at which to start iterating (which need not necessarily exist in the database, in which case the next one, if any, is chosen instead), and the order in which the records should be traversed. Possible values for *order* are:

- `sophia.SPGT` - increasing order (skipping the key, if it is equal)
- `sophia.SPGTE` - increasing order (with key)
- `sophia.SPLT` - decreasing order (skipping the key, if it is equal)
- `sophia.SPLTE` - decreasing order (with key)

By default, iteration is done in lexicographical order, and starts at the very first key in the database, including it.

Here is, for example, how you would iterate over all the keys in a database starting with a given prefix, skipping the prefix itself (if it exists), and in lexicographical order:

```
import sophia, itertools

def iter_prefixes(db, prefix):
    cursor = db.iterkeys(prefix, sophia.SPGT)
    return itertools.takewhile(lambda key: key.startswith(prefix), cursor)

# create a database with some records to check this works
db = sophia.Database()
db.open("prefix_db")
db.set("think", "")
db.set("thought", "")
db.set("thinking", "")
db.set("thinker", "")
```

At the prompt:

```
>>> list(iter_prefixes(db, "think"))
['thinker', 'thinking']
```

2.1.5 Storing rich objects

It is possible to store any kind of Python object in a database, as long as this object is serialisable. The class `sophia.ObjectDatabase` defines an interface for marshalling/unmarshalling data transparently. By default, it serialises objects (both keys and values) with the `pickle` module. If the shape of your data permits it, you may prefer to use the `struct` module. It is faster than `pickle`, and is language-independent (which means you can open the same database from C, Python, Lua, or what not, without pain), but on the other hand can only handle fixed-type data.

Here is, for example, how you would write an interface for a database intended to be used for storing mappings of unicode keys to unsigned integers. Here we choose to encode the keys in UTF-8, and to represent the integers as C `unsigned long`, packed in network order (so that the database is portable across architectures):

```
import sophia, struct

# our custom structure for packing integers
value_struct = struct.Struct("!L")

# serialization functions
pack_key     = lambda k: k.encode("utf-8")
unpack_key   = lambda k: k.decode("utf-8")
pack_value   = value_struct.pack
unpack_value = lambda v: value_struct.unpack(v)[0]

# anonymous function for instantiating the `ObjectDatabase` class
# with our custom marshalling functions

MyDB = lambda: sophia.ObjectDatabase(pack_key, unpack_key,
    pack_value, unpack_value)
```

You can now create a database and access it as expected:

```
>>> db = MyDB()
>>> db.open("my_db")
>>> db.set(u"Penny", 22)
>>> db.set(u"Bruce", 45)
>>> db.get(u"Penny")
22
>>> list(db.iteritems())
[(u'Bruce', 45), (u'Penny', 22)]
```

2.1.6 Tuning

All the tuning options available in the C API are accessible from Python, at the exception of `SPALLOC` and `SPDIR`. Options are set on the `Database` object itself with the method `Database.setopt()`, which takes as argument the constant identifying the option (`SPCMP`, `SPPAGE`, etc.), and one or two arguments (depending on the option) indicating the value(s) to be set. The relevant constants are exported into the python module, so you can access them as `sophia.SPCMP`, `sophia.SPPAGE`, etc.

The more useful option is perhaps `SPCMP`, which can be used to define a custom function for ordering the keys while traversing the database. This function will be passed as argument the first key, its length, the second one, and the corresponding length, in that order, and should return -1, 0, or 1, respectively, if the first key is lower, equal, or higher than the second one. Here is how you would define one for comparing keys on their length, and attach it to your database instance:

```
def compare_on_length(key1, len1, key2, len2):
    return -1 if len1 < len2 else int(len1 > len2)

db = sophia.Database()
db.setopt(sophia.SPCMP, compare_on_length)

# add some records to check this works
db.open("cmp_db")
db.set("long key", "")
db.set("key", "")
db.set("very long key", "")
```

At the prompt:

```
>>> list(db.iterkeys())
['key', 'long key', 'very long key']
```

Options persist into a `Database` object until it is destroyed, and can't be changed while the database is opened.

2.1.7 On threading

Two things should be kept in mind if you intend to use *sophia* in a threaded environment:

- It is not possible to open more than one connection to the same database at the same time. On the other hand, it is ok to share the same database object between threads.
- It is not possible to perform a transaction or to set/delete a record while a `sophia.Cursor` object (as returned by the group of methods `Database.iterkeys()`, etc.) is alive. It is, however, possible to create a cursor object while a transaction is active.

A class `sophia.ThreadedDatabase` handles the second case by protecting the necessary functions with a lock. It should not be used, however, when it isn't necessary, as it imposes a significant overhead on writing operations. Here is a summary of what classes you should use depending on what you intend to do with them:

- If you don't work in a threaded environment, use the *sophia.Database* and `sophia.ObjectDatabase` classes.
- If you work in a threaded environment BUT don't need to iterate over the database, do the same as above, and make sure you create and open the database object in the main thread, before passing it around to the other threads, so that the connection itself is safe.
- If you work in a threaded environment AND need to iterate over the database, use the `sophia.ThreadedDatabase` class and its sibling `sophia.ThreadedObjectDatabase`.

2.1.8 Cursors pitfall

A special behaviour has to be kept in mind when dealing with cursors: it is not possible to close or reopen a database while a cursor is in use. The return value of `Database.close()` and `Database.open()` (in addition with `Database.is_closed()`), will tell you whether the database has been effectively closed or re-opened, respectively, when you call them. If `Database.open()` and `Database.close()` return *False*, you should understand that there is at least one cursor lying out there that needs to be deallocated. The database will effectively be closed as soon as the last remaining opened cursor is closed. A cursor is closed either when it has been exhausted through iteration, or when it goes out of scope:

```
>>> # open a database and create a cursor
>>> db.open("pitfall_db")
True
>>> cursor = db.iterkeys()
>>> # try to close the database while a cursor is active; this doesn't work
>>> db.close()
False
>>> # delete the cursor to make it work; the database will be closed immediately after
>>> del cursor
>>> db.is_closed()
True
```

2.1.9 Final note

You may want to check the Reference for a summary of the above, as well as the sophia documentation if you need more details.

2.2 Reference

2.2.1 Main objects

exception `sophia.`**`Error`**
Exception raised for all database-related errors.

class `sophia.`**`Database`**
Main database class.

Keys or values passed as argument to the methods which accept them are expected to be byte strings. Returned keys or values are always byte strings.

`setopt`(*constant*, *value1*[, *value2*])
Configure this database.

Calling this method is only valid when the database is closed. See the sophia documentation for a summary of the available options. `SPDIR` and `SPALLOC` are not supported.

`open`(*path*)
Open the database, creating it if doesn't exist yet.

If a connection is already active, try to close it and open a new one; in this case, *False* can be returned, which means that the previous connection has not been successfully closed because a *sophia.Cursor* object is hanging around somewhere. Otherwise, *True* is returned.

`close`()
Close the current connection, if any.

As above, a return value *False* indicates that it is not possible to close the database for the time being.

`is_closed`()
Is this database closed? *True* if so, *False* otherwise.

`set`(*key*, *value*)
Add a record, or replace an existent one.

`get`(*key*[, *default*])
Retrieve a record given its key. If it doesn't exist, return *default* if given, *None* otherwise.

`delete`(*key*)
Delete a record.

`contains`(*key*)
Is this key in the database? *True* if so, *False* otherwise.

`begin`()
Start a transaction.

`commit`()
Commit the current transaction.

`rollback`()
Abort the current transaction.

`len`()
How many records are there in this database?

`iterkeys`(*start_key=None*, *order=sophia.SPGTE*)
Iterate over all the keys in this database, starting at *start_key*, and in *order*.

Possible values for *order* are:

- `sophia.SPGT` - increasing order (skipping the key, if it is equal)
- `sophia.SPGTE` - increasing order (with key)
- `sophia.SPLT` - decreasing order (skipping the key, if it is equal)
- `sophia.SPLTE` - decreasing order

itervalues(*start_key=None*, *order=sophia.SPGTE*)
Same as *Database.iterkeys()*, but for values.

iteritems(*start_key=None*, *order=sophia.SPGTE*)
Same as *Database.iterkeys()*, but for pairs of (key, value).

2.2.2 Database models

class sophia.**ObjectDatabase**(*pack_key=pickle.dumps*, *unpack_key=pickle.loads*, *pack_value=pickle.dumps*, *unpack_value=pickle.loads*)
Database model for storing arbitrary kinds of objects.

pack_key, *unpack_key*, *pack_value*, and *unpack_value*, should be callables that, when passed an object as parameter, return a byte representation of it, suitable for storage. By default, all these functions use the `pickle` module.

class sophia.**ThreadedDatabase**
Thread-safe database model.

It should only be used if you want to use a database in a threaded environment AND need to iterate over it. Otherwise, the vanilla *Database* class is suitable (and more efficient).

class sophia.**ThreadedObjectDatabase**(*pack_key=pickle.dumps*, *unpack_key=pickle.loads*, *pack_value=pickle.dumps*, *unpack_value=pickle.loads*)
Mixing of a `ThreadedDatabase` and an `ObjectDatabase`.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

S

B

C

D

G

I

L

O

R

S