dbm — Unix Key-Value Databases

Purpose: dbm provides a generic dictionary-like interface to DBM-style, string-keyed databases

`dbm` is a front-end for DBM-style databases that use simple string values as keys to access records containing strings. It uses `whichdb()` to identify databases, then opens them with the appropriate module. It is used as a back-end for `shelve`, which stores objects in a DBM database using `pickle`.

Database Types

Python comes with several modules for accessing DBM-style databases. The default implementation selected depends on the libraries available on the current system and the options used when Python was compiled. Separate interfaces to the specific implementations allow Python programs to exchange data with programs in other languages that do not automatically switch between available formats, or to write portable data files that will work on multiple platforms.

dbm.gnu

`dbm.gnu` is an interface to the version of the `dbm` library from the GNU project. It works the same as the other DBM implementations described here, with a few changes to the `flags` supported by `open()`.

Besides the standard `'r'`, `'w'`, `'c'`, and `'n'` flags, `dbm.gnu.open()` supports:

- `'f'` to open the database in *fast* mode. In fast mode, writes to the database are not synchronized.
- `'s'` to open the database in *synchronized* mode. Changes to the database are written to the file as they are made, rather than being delayed until the database is closed or synced explicitly.
- `'u'` to open the database unlocked.

dbm.ndbm

The `dbm.ndbm` module provides an interface to the Unix ndbm implementations of the dbm format, depending on how the module was configured during compilation. The module attribute `library` identifies the name of the library `configure` was able to find when the extension module was compiled.

dbm.dumb

The `dbm.dumb` module is a portable fallback implementation of the DBM API when no other implementations are available. No external dependencies are required to use `dbm.dumb`, but it is slower than most other implementations.

Creating a New Database

The storage format for new databases is selected by looking for usable versions of each of the sub-modules in order.

- `dbm.gnu`
- `dbm.ndbm`
- `dbm.dumb`

The `open()` function takes `flags` to control how the database file is managed. To create a new database when necessary, use `'c'`. Using `'n'` always creates a new database, overwriting an existing file.

```
# dbm_new.py

import dbm

with dbm.open('/tmp/example.db', 'n') as db:
    db['key'] = 'value'
    db['today'] = 'Sunday'
    db['author'] = 'Doug'
```

In this example, the file is always re-initialized.

```
$ python3 dbm_new.py
```

`whichdb()` reports the type of database that was created

whichdb() reports the type of database that was created.

```
# dbm_whichdb.py

import dbm

print(dbm.whichdb('/tmp/example.db'))
```

Output from the example program will vary, depending on which modules are installed on the system.

```
$ python3 dbm_whichdb.py

dbm.ndbm
```

Opening an Existing Database

To open an existing database, use `flags` of either `'r'` (for read-only) or `'w'` (for read-write). Existing databases are automatically given to `whichdb()` to identify, so it as long as a file can be identified, the appropriate module is used to open it.

```
# dbm_existing.py

import dbm

with dbm.open('/tmp/example.db', 'r') as db:
    print('keys():', db.keys())
    for k in db.keys():
        print('iterating:', k, db[k])
    print('db["author"] =', db['author'])
```

Once open, `db` is a dictionary-like object. New keys are always converted to byte strings when added to the database, and returned as byte strings.

```
$ python3 dbm_existing.py

keys(): [b'key', b'today', b'author']
iterating: b'key' b'value'
iterating: b'today' b'Sunday'
iterating: b'author' b'Doug'
db["author"] = b'Doug'
```

Error Cases

The keys of the database need to be strings.

```
# dbm_intkeys.py
```

```
import dbm

with dbm.open('/tmp/example.db', 'w') as db:
    try:
        db[1] = 'one'
    except TypeError as err:
        print(err)
```

Passing another type results in a `TypeError`.

```
$ python3 dbm_intkeys.py

dbm mappings have bytes or string keys only
```

Values must be strings or `None`.

```
# dbm_intvalue.py
```

```
import dbm
```

```
with dbm.open('/tmp/example.db', 'w') as db:
    try:
        db['one'] = 1
    except TypeError as err:
        print(err)
```

A similar TypeError is raised if a value is not a string.

```
$ python3 dbm_intvalue.py

dbm mappings have byte or string elements only
```

See also

- Standard library documentation for dbm
- Python 2 to 3 porting notes for anydbm
- Python 2 to 3 porting notes for whichdb
- shelve – Examples for the shelve module, which uses dbm to store data.

shelve — Persistent Storage of Objects

sqlite3 — Embedded Relational Database

Quick Links

Database Types
dbm.gnu
dbm.ndbm
dbm.dumb
Creating a New Database
Opening an Existing Database
Error Cases

This page was last updated 2016-12-28.

Navigation

shelve — Persistent Storage of Objects
sqlite3 — Embedded Relational Database

Other Writing

- Blog
- The Python Standard Library By Example