`!dbm` --- Interfaces to Unix "databases"

- Key and value types
- !dbm.sqlite3 --- SQLite backend for dbm
- !dbm.gnu --- GNU database manager
- !dbm.ndbm --- New Database Manager
- !dbm.dumb --- Portable DBM implementation

dbm

Source code: `Lib/dbm/__init__.py`

`!dbm` is a generic interface to variants of the DBM database:

- `dbm.sqlite3`
- `dbm.gnu`
- `dbm.ndbm`

If none of these modules are installed, the slow-but-simple implementation in module `dbm.dumb` will be used. There is a third party interface to the Oracle Berkeley DB.

> [!NOTE] None of the underlying modules will automatically shrink the disk space used by the database file. However, `dbm.sqlite3`, `dbm.gnu` and `dbm.dumb` provide a `!reorganize` method that can be used for this purpose.

error

A tuple containing the exceptions that can be raised by each of the supported modules, with a unique exception also named `dbm.error` as the first item --- the latter is used when `dbm.error` is raised.

whichdb(filename)

This function attempts to guess which of the several simple database modules available --- `dbm.sqlite3`, `dbm.gnu`, `dbm.ndbm`, or `dbm.dumb` --- should be used to open a given file.

Return one of the following values:

- `None` if the file can't be opened because it's unreadable or doesn't exist
- the empty string (`''`) if the file's format can't be guessed
- a string containing the required module name, such as `'dbm.ndbm'` or `'dbm.gnu'`

3.11 *filename* accepts a `path-like object`.

open(file, flag='r', mode=0o666)

Open a database and return the corresponding database object.

param file
The database file to open.

If the database file already exists, the `whichdb` function is used to determine its type and the appropriate module is used; if it does not exist, the first submodule listed above that can be imported is used.

type file
`path-like object`

param str flag

- `'r'` (default): Open existing database for reading only.
- `'w'`: Open existing database for reading and writing.
- `'c'`: Open database for reading and writing, creating it if it doesn't exist.
- `'n'`: Always create a new, empty database, open for reading and writing.

param int mode
The Unix file access mode of the file (default: octal `0o666`), used only when the database has to be created.

3.11 *file* accepts a `path-like object`.

The object returned by `~dbm.open` supports the basic functionality of mutable `mappings <mapping>`; keys and their corresponding values can be stored, retrieved, and deleted, and iteration, the `in` operator and methods `!keys`, `!get`, `!setdefault` and `!clear` are available. The `!keys` method returns a list instead of a view object. The `!setdefault` method requires two arguments.

Key and values are always stored as `bytes`. This means that when strings are used they are implicitly converted to the default encoding before being stored. For detailed information about

accepted types, see `dbm-key-value-types`.

These objects also support being used in a `with` statement, which will automatically close them when done.

3.2 `!get` and `!setdefault` methods are now available for all `!dbm` backends.

3.4 Added native support for the context management protocol to the objects returned by `~dbm.open`.

3.8 Deleting a key from a read-only database raises a database module specific exception instead of `KeyError`.

3.13 `!clear` methods are now available for all `!dbm` backends.

Key and value types

The accepted types for keys and values vary by backend. Keys and values are handled the same way, except where noted below:

- **Traditional backends:**
  - `dbm.gnu` and `dbm.ndbm`: Accept `str` and `bytes` objects.
  - `dbm.dumb`: Accepts `str` and `bytes` objects; `bytearray` is acceptable as a value, but not as a key.
- **SQLite backend** (`dbm.sqlite3`): Accepts `str`, `bytes`, `int`, `float`, `bool`, `bytearray`, `memoryview`, and `array.array` objects.

Storage format:

All keys and values are stored as `bytes` objects in the database. When retrieving, you'll always get bytes back, regardless of the original type stored.

Type conversion examples:

- `db['key'] = 'string'` stored as `b'string'`
- `db['key'] = bytearray(b'data')` stored as `b'data'` (`dbm.dumb` and `dbm.sqlite3` only)
- `db['key'] = 42` stored as `b'42'` (`dbm.sqlite3` only)
- `db['key'] = 3.14` stored as `b'3.14'` (`dbm.sqlite3` only)
- `db['key'] = True` stored as `b'1'` (`dbm.sqlite3` only)
- `db['key'] = False` stored as `b'0'` (`dbm.sqlite3` only)
- `db['key'] = memoryview(b'data')` stored as `b'data'` (`dbm.sqlite3` only)
- `db['key'] = array.array('i', [1, 2, 3])` stored as bytes (for example, on little-endian: `b'\x01\x00\x00\x00\x02\x00\x00\x00\x03\x00\x00\x00'`) (`dbm.sqlite3` only)

The following example records some hostnames and a corresponding title, and then prints out the contents of the database:

```
import dbm

# Open database, creating it if necessary.
with dbm.open('cache', 'c') as db:

    # Record some values
    db[b'hello'] = b'there'
    db['www.python.org'] = 'Python Website'
    db['www.cnn.com'] = 'Cable News Network'

    # Note that the keys are considered bytes now.
    assert db[b'www.python.org'] == b'Python Website'
    # Notice how the value is now in bytes.
    assert db['www.cnn.com'] == b'Cable News Network'

    # Often-used methods of the dict interface work too.
    print(db.get('python.org', b'not present'))

    # Storing a non-string key or value behavior depends on the backend:
    # - Traditional backends (ndbm, gnu, dumb) will raise a TypeError
    # - The sqlite3 backend accepts it and converts to bytes
    db['www.yahoo.com'] = 4

# db is automatically closed when leaving the with statement.
```

Module `shelve`
Persistence module which stores non-string data.

The individual submodules are described in the following sections.

`!dbm.sqlite3` --- SQLite backend for dbm

dbm.sqlite3

3.13

Source code: `Lib/dbm/sqlite3.py`

This module uses the standard library `sqlite3` module to provide an SQLite backend for the `!dbm` module. The files created by `!dbm.sqlite3` can thus be opened by `sqlite3`, or any other SQLite browser, including the SQLite CLI.

not WASI.

This module does not work or is not available on WebAssembly. See `wasm-availability` for more information.

open(filename, /, flag="r", mode=0o666)

Open an SQLite database.

param filename
The path to the database to be opened.

type filename
`path-like object`

param str flag

> - `'r'` (default): Open existing database for reading only.
> - `'w'`: Open existing database for reading and writing.
> - `'c'`: Open database for reading and writing, creating it if it doesn't exist.
> - `'n'`: Always create a new, empty database, open for reading and writing.

param mode
The Unix file access mode of the file (default: octal `0o666`), used only when the database has to be created.

The returned database object behaves similar to a mutable `mapping`, but the `!keys` method returns a list, and the `!setdefault` method requires two arguments. It also supports a "closing" context manager via the `with` keyword.

The following methods are also provided:

sqlite3.close()

Close the SQLite database.

sqlite3.reorganize()

If you have carried out a lot of deletions and would like to shrink the space used on disk, this method will reorganize the database; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

> [!NOTE] While reorganizing, as much as two times the size of the original database is required in free disk space. However, be aware that this factor changes for each `!dbm` submodule.

3.15

`!dbm.gnu` --- GNU database manager

dbm.gnu

Source code: `Lib/dbm/gnu.py`

The `!dbm.gnu` module provides an interface to the `GDBM (GNU dbm)` library, similar to the `dbm.ndbm` module, but with additional functionality like crash tolerance.

> [!NOTE] The file formats created by `!dbm.gnu` and `dbm.ndbm` are incompatible and can not be used interchangeably.

not Android, not iOS, not WASI.

This module is not supported on `mobile platforms <mobile-availability>` or `WebAssembly platforms <wasm-availability>`.

Unix.

error

Raised on `!dbm.gnu`-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

open_flags

A string of characters the *flag* parameter of `~dbm.gnu.open` supports.

GDBM_VERSION_INFO

A named tuple containing the three components of the GDBM library version that was used for building the module: *major*, *minor*, and *patch*. All values are integers. The components can also be accessed by name, so `dbm.gnu.GDBM_VERSION_INFO[0]` is equivalent to `dbm.gnu.GDBM_VERSION_INFO.major` and so on. This may be different from the GDBM library actually used at runtime, which is available as `gdbm_version_info`.

3.16

gdbm_version_info

A named tuple containing the version of the GDBM library actually loaded by the interpreter, with the same fields as `GDBM_VERSION_INFO`.

3.16

gdbm_version

The release string of the GDBM library actually loaded by the interpreter, like `'GDBM version 1.26. 30/07/2025'`.

3.16

open(filename, flag="r", mode=0o666, /)

Open a GDBM database and return a `!gdbm` object.

param filename
The database file to open.

type filename
`path-like object`

param str flag

- `'r'` (default): Open existing database for reading only.
- `'w'`: Open existing database for reading and writing.
- `'c'`: Open database for reading and writing, creating it if it doesn't exist.
- `'n'`: Always create a new, empty database, open for reading and writing.

The following additional characters may be appended to control how the database is opened:

- `'f'`: Open the database in fast mode. Writes to the database will not be synchronized.
- `'s'`: Synchronized mode. Changes to the database will be written immediately to the file.
- `'u'`: Do not lock database.

Not all flags are valid for all versions of GDBM. See the `open_flags` member for a list of supported flag characters.

param int mode
The Unix file access mode of the file (default: octal `0o666`), used only when the database has to be created.

raises error
If an invalid *flag* argument is passed.

3.11 *filename* accepts a `path-like object`.

`!gdbm` objects behave similar to mutable `mappings <mapping>`, but methods `!items`, `!values`, `!pop`, `!popitem`, and `!update` are not supported, the `!keys` method returns a list, and the `!setdefault` method requires two arguments. It also supports a "closing" context manager via the `with` keyword.

3.2 Added the `!get` and `!setdefault` methods.

3.13 Added the `!clear` method.

The following methods are also provided:

gdbm.close()

Close the GDBM database.

gdbm.firstkey()

It's possible to loop over every key in the database using this method and the `nextkey` method. The traversal is ordered by GDBM's internal hash values, and won't be sorted by the key values. This method returns the starting key.

gdbm.nextkey(key)

Returns the key that follows *key* in the traversal. The following code prints every key in the database `db`, without having to create a list in memory that contains them all:

```
k = db.firstkey()
while k is not None:
    print(k)
    k = db.nextkey(k)
```

gdbm.reorganize()

If you have carried out a lot of deletions and would like to shrink the space used by the GDBM file, this routine will reorganize the database. `!gdbm` objects will not shorten the length of a database file except by using this reorganization; otherwise, deleted file space will be kept and reused as new (key, value) pairs are added.

> [!NOTE] While reorganizing, as much as one time the size of the original database is required in free disk space. However, be aware that this factor changes for each `!dbm` submodule.

gdbm.sync()

When the database has been opened in fast mode, this method forces any unwritten data to be written to the disk.

`!dbm.ndbm` --- New Database Manager

dbm.ndbm

Source code: `Lib/dbm/ndbm.py`

The `!dbm.ndbm` module provides an interface to the `NDBM (New Database Manager)` library. This module can be used with the "classic" NDBM interface or the `GDBM (GNU dbm)` compatibility interface.

> [!NOTE] The file formats created by `dbm.gnu` and `!dbm.ndbm` are incompatible and can not be used interchangeably.

> [!WARNING] The NDBM library shipped as part of macOS has an undocumented limitation on the size of values, which can result in corrupted database files when storing values larger than this limit. Reading such corrupted files can result in a hard crash (segmentation fault).

not Android, not iOS, not WASI.

This module is not supported on `mobile platforms <mobile-availability>` or `WebAssembly platforms <wasm-availability>`.

Unix.

error

Raised on `!dbm.ndbm`-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

library

Name of the NDBM implementation library used: `'GNU gdbm'`, `'Berkeley DB'` or `'ndbm'`.

3.16 The value is `'ndbm'` for a classic NDBM library. It was `'GNU gdbm'` before.

GDBM_VERSION_INFO gdbm_version_info gdbm_version

Information about the GDBM library in use, with the same meaning as the constants of the same names in `dbm.gnu`. Only available if `library` is `'GNU gdbm'`.

3.16

BDB_VERSION

The version string of the Berkeley DB library that was used for building the module, like `'Berkeley DB 5.3.28: (September  9, 2013)'`. This may be different from the Berkeley DB library actually used at runtime, which is available as `bdb_version`. Only available if `library` is `'Berkeley DB'`.

3.16

bdb_version

The version string of the Berkeley DB library actually loaded by the interpreter. Only available if `library` is `'Berkeley DB'`.

3.16

BDB_VERSION_INFO

A named tuple containing the three components of the Berkeley DB library version that was used for building the module: *major*, *minor*, and *patch*. All values are integers. The components can also be accessed by name, so `dbm.ndbm.BDB_VERSION_INFO[0]` is equivalent to `dbm.ndbm.BDB_VERSION_INFO.major` and so on. This may be different from the Berkeley DB library actually used at runtime, which is available as `bdb_version_info`. Only available if `library` is `'Berkeley DB'`.

3.16

bdb_version_info

A named tuple containing the version of the Berkeley DB library actually loaded by the interpreter, with the same fields as `BDB_VERSION_INFO`. Only available if `library` is `'Berkeley DB'`.

3.16

open(filename, flag="r", mode=0o666, /)

Open an NDBM database and return an `!ndbm` object.

param filename
The basename of the database file (without the `.dir` or `.pag` extensions).

type filename
`path-like object`

param str flag

- `'r'` (default): Open existing database for reading only.
- `'w'`: Open existing database for reading and writing.
- `'c'`: Open database for reading and writing, creating it if it doesn't exist.
- `'n'`: Always create a new, empty database, open for reading and writing.

param int mode
The Unix file access mode of the file (default: octal `0o666`), used only when the database has to be created.

3.11 Accepts `path-like object` for filename.

`!ndbm` objects behave similar to mutable `mappings <mapping>`, but methods `!items`, `!values`, `!pop`, `!popitem`, and `!update` are not supported, the `!keys` method returns a list, and the `!setdefault` method requires two arguments. It also supports a "closing" context manager via the `with` keyword.

3.2 Added the `!get` and `!setdefault` methods.

3.13 Added the `!clear` method.

The following method is also provided:

ndbm.close()

Close the NDBM database.

`!dbm.dumb` --- Portable DBM implementation

dbm.dumb

Source code: `Lib/dbm/dumb.py`

single: databases

> [!NOTE] The `!dbm.dumb` module is intended as a last resort fallback for the `!dbm` module when a more robust module is not available. The `!dbm.dumb` module is not written for speed and is not nearly as heavily used as the other database modules.

The `!dbm.dumb` module provides a persistent `dict`-like interface which is written entirely in Python. Unlike other `!dbm` backends, such as `dbm.gnu`, no external library is required.

The `!dbm.dumb` module defines the following:

error

Raised on `!dbm.dumb`-specific errors, such as I/O errors. `KeyError` is raised for general mapping errors like specifying an incorrect key.

open(filename, flag="c", mode=0o666)

Open a `!dbm.dumb` database.

param filename
The basename of the database file (without extensions). A new database creates the following files:

- `{filename}.dat`
- `{filename}.dir`

type database
`path-like object`

param str flag

- `'r'`: Open existing database for reading only.
- `'w'`: Open existing database for reading and writing.
- `'c'` (default): Open database for reading and writing, creating it if it doesn't exist.
- `'n'`: Always create a new, empty database, open for reading and writing.

param int mode
The Unix file access mode of the file (default: octal `0o666`), used only when the database has to be created.

> [!WARNING] It is possible to crash the Python interpreter when loading a database with a sufficiently large/complex entry due to stack depth limitations in Python's AST compiler.

> [!WARNING] `!dbm.dumb` does not support concurrent read/write access. (Multiple simultaneous read accesses are safe.) When a program has the database open for writing, no other program should have it open for reading or writing.

3.5 `~dbm.dumb.open` always creates a new database when *flag* is `'n'`.

3.8 A database opened read-only if *flag* is `'r'`. A database is not created if it does not exist if *flag* is `'r'` or `'w'`.

3.11 *filename* accepts a `path-like object`.

The returned database object behaves similar to a mutable `mapping`, but the `!keys` and `!items` methods return lists, and the `!setdefault` method requires two arguments. It also supports a "closing" context manager via the `with` keyword.

The following methods are also provided:

dumbdbm.close()

Close the database.

dumbdbm.reorganize()

If you have carried out a lot of deletions and would like to shrink the space used on disk, this method will reorganize the database; otherwise, deleted file space will not be reused.

> [!NOTE] While reorganizing, no additional free disk space is required. However, be aware that this factor changes for each `!dbm` submodule.

3.15

dumbdbm.sync()

Synchronize the on-disk directory and data files. This method is called by the `shelve.Shelf.sync` method.

□□□ #2
□ □□□ □□□ 2026-10-31 17:07:48 UTC
□ □□□ □□□ 2026-10-31 17:07:51 UTC