Data Storage

"Unlike mainstream component programming, scripts usually do not introduce new components but simply "wire" existing ones. Scripts can be seen as introducing behavior but no new state. /.../ Of course, there is nothing to stop a "scripting" language from introducing persistent state — it then simply turns into a normal programming language"

Clemens Szyperski, in Component Software

Overview

Python comes with drivers for a number of very similar database managers, all modeled after Unix's **dbm** library. These databases behaves like ordinary dictionaries, with the exception that you can only use strings for keys and values (the **shelve** module can handle any kind of value).

The anydbm module

This module provides a unified interface to the simple database drivers supported by Python.

The first time it is imported, the **anydbm** module looks for a suitable database driver, testing for **dbhash**, **gdbm**, **dbm**, or **dumbdbm**, in that order. If no such module is found, it raises an **ImportError** exception.

The **open** function is used to open or create a database, using the chosen database handler.

Example: Using the anydbm module

```
# File:anydbm-example-1.py

import anydbm

db = anydbm.open("database", "c")
db["1"] = "one"
db["2"] = "two"
db["3"] = "three"
db.close()

db = anydbm.open("database", "r")
for key in db.keys():
    print repr(key), repr(db[key])
```

```
'2' 'two'
'3' 'three'
'1' 'one'
```

The whichdb module

This module can be used to figure out which database handler that was used for a given database file.

Example: Using the whichdb module

```
# File:whichdb-example-1.py

import whichdb

filename = "database"

result = whichdb.whichdb(filename)

if result:
    print "file created by", result
    handler = __import__(result)
    db = handler.open(filename, "r")
    print db.keys()
else:
    # cannot identify data base
    if result is None:
        print "cannot read database file", filename
    else:
        print "cannot identify database file", filename
    db = None
```

This example used the **__import__** function to import a module with the given name.

The shelve module

This module uses the database handlers to implement persistent dictionaries. A shelve object uses string keys, but the value can be of any data type, as long as it can be handled by the **pickle** module.

Example: Using the shelve module

```
# File:shelve-example-1.py

import shelve

db = shelve.open("database", "c")
db["one"] = 1
db["two"] = 2
db["three"] = 3
db.close()

db = shelve.open("database", "r")
for key in db.keys():
    print repr(key), repr(db[key])
```

```
'one' 1
'three' 3
'two' 2
```

The following example shows how to use the **shelve** module with a given database driver.

Example: Using the shelve module with a given database

```
# File:shelve-example-3.py

import shelve
import gdbm

def gdbm_shelve(filename, flag="c"):
    return shelve.Shelf(gdbm.open(filename, flag))

db = gdbm_shelve("dbfile")
```

The dbhash module

(Optional). This module provides a **dbm**-compatible interface to the **bsddb** database handler.

Example: Using the dbhash module

```
# File:dbhash-example-1.py

import dbhash

db = dbhash.open("dbhash", "c")
db["one"] = "the foot"
db["two"] = "the shoulder"
db["three"] = "the other foot"
db["four"] = "the bridge of the nose"
db["five"] = "the naughty bits"
db["six"] = "just above the elbow"
db["seven"] = "two inches to the right of a very naughty bit indeed"
db["eight"] = "the kneecap"
db.close()

db = dbhash.open("dbhash", "r")
for key in db.keys():
    print repr(key), repr(db[key])
```

The dbm module

(Optional). This module provides an interface to the **dbm** database handler (available on many Unix platforms).

Example: Using the dbm module

```
# File:dbm-example-1.py

import dbm

db = dbm.open("dbm", "c")
db["first"] = "bruce"
db["second"] = "bruce"
db["third"] = "bruce"
db["fourth"] = "bruce"
db["fifth"] = "michael"
db["fifth"] = "bruce" # overwrite
db.close()

db = dbm.open("dbm", "r")
for key in db.keys():
    print repr(key), repr(db[key])
```

```
'first' 'bruce'
'second' 'bruce'
'fourth' 'bruce'
'third' 'bruce'
'fifth' 'bruce'
```

The dumbdbm module

This is a very simple database implementation, similar to **dbm** and friends, but written in pure Python. It uses two files; a binary file (.dat) which contains the data, and a text file (.dir) which contain data descriptors.

Example: Using the dumbdbm module

```
# File:dumbdbm-example-1.py

import dumbdbm

db = dumbdbm.open("dumbdbm", "c")
db["first"] = "fear"
db["second"] = "surprise"
db["third"] = "ruthless efficiency"
db["fourth"] = "an almost fanatical devotion to the Pope"
db["fifth"] = "nice red uniforms"
db.close()

db = dumbdbm.open("dumbdbm", "r")
for key in db.keys():
    print repr(key), repr(db[key])
```

```
'first' 'fear'
'third' 'ruthless efficiency'
'fifth' 'nice red uniforms'
'second' 'surprise'
'fourth' 'an almost fanatical devotion to the Pope'
```

The gdbm module

(Optional). This module provides an interface to the GNU **dbm** database handler.

Example: Using the gdbm module

```
# File:gdbm-example-1.py

import gdbm

db = gdbm.open("gdbm", "c")
db["1"] = "call"
db["2"] = "the"
db["3"] = "next"
db["4"] = "defendant"
db.close()

db = gdbm.open("gdbm", "r")

keys = db.keys()
keys.sort()
for key in keys:
    print db[key],
```

```
call the next defendant
```