Chapter 13 — Persistent Data

Introduction

Stand alone programs that do not save and use saved data are interesting, but most programs need to store and remember them. This chapter will introduce the UNIX DB and the Shelve Libraries to create stores for data. These are different from files, introduced in a previous chapter, in that

Objectives

Upon completion of this chapter's exercises, you should be able to:

- Blah de blah.
- Baz and Barf.

Prerequisites

This Chapter requires...

The UNIX DB

The UNIX DB module has a confusing name because of its origin, but it is available on virtually all Python implementations. The UNIX DB (the `dbm` module) acts very much like a dictionary, except that key value pairs you set are stored on the hard drive for later.

The actual method and operating system library used to store the data varies between implementations. On UNIX and LINUX systems the GNU gdbm or the UNIX ndb libraries are used. On Windows systems a hashed file or "dumb" file is used to store the key value pairs. The files stored on the systems may not be readable on other systems, but using them to store values is the same.

The `dbm` module has a limitation that all of the keys and values must be collections of bytes. You may use strings, but the values when they are returned will be collections of bytes and need to be encoded back to their original format (if they include Unicode characters); This also means that you will need to encode/decode numbers into collections of bytes or to strings.

Using dbm

This sample program uses a `dbm` object to store a value that the user enters. When the user runs the program again they can either change the value, or retrieve the old value. This program demonstrates that if a Unicode character outside the range of 0-127 is entered, the result will be returned as a collection of bytes. Like in file-system IO, it is strongly suggested that you use the `with` statement to manage the context of the module.

To use the `dbm` module, you first must `import dbm` into your program. This will load the best library for key value storage you have available to you.

Now all you need to do is "open it". The open method `dbm.open()` usually requires two values, file name and mode. The file name may be a simple name or the path to a file and does not need to include a file extension. The mode may be: `'r'` -read only, `'w'` -reading and writing, `'c'` — create a new database if it does not exist for reading and writing, and `'n'` — always create a new database. The 'r' mode is default (if you do not specify a mode).

If you are using the `dbm` module in a `with` context manager, it will be closed for you automatically when you leave the suite of code. If you are accessing dbm outside a context manager, you will need to close your `dbm` object using the `.close()` method.

dbm	Module
The `dbm` module will create a persistent key-value storage of byte arrays for your program to use REF	

dbm.open(*file_name, mode*)	Method of dbm
Open a UNIX database with the file name specified and return an object that can be used to access it.	

Mode	Description
“r”	Open the database for “read-only” access. This is the default action of mode is not specified.
“w”	Open the database for “write-only”. Use this mode is you are populating a database with large quantities of data and you are not going to be reading from it.
“c”	Create a database if it does not exist and then open it in read and write

	mode.	
“n”	Delete the old database create a new one. Open it in read and write mode.	
REF		

dbm_object.**close()**	Method of dbm
Closes a `dbm` database. If a database is not closed, changes may not be committed to the disk. It is recommended that you use a `with` context manager, and allow it to automatically close your database. REF	

```
1| import dbm
2| key = b'uservalue'
3|
4| a = input("enter a value to save or press enter to see old value
    -->")
5|
6| with dbm.open('demodbm', 'c') as d:
7|     if not a:
8|         value = d.get(key,"NOTHING")
9|         print("your value was", value)
10|    else:
11|        d[key] = a
12|        print("your value was saved")
```

```
enter a value to save or press enter to see old value -->
your value was NOTHING

enter a value to save or press enter to see old value --
>abcd 😆 efgh
your value was saved

enter a value to save or press enter to see old value -->
your value was b'abcd \xf0\x9f\x98\x85 efgh'
```

Using Shelve for Persistent Values

The `shelve` library actually combines the `pickle` library and the `dbm` library to create an easy way to store virtually anything. With `shelve` you do not have to convert the data to a string or a collection of bytes. The `pickle` library is insecure and may expose your system to malicious code. It is recommended that you do not use `shelve` data files from untrustworthy sources. As with `dbm` it is strongly recommended that you execute the `.close()` method or execute it as a context manager using the `with` statement.

shelve	Module
The `shelve` module combines the `dbm` module with the `pickle` module to create a key-value persistent store that will return values of various types, and nor just byte arrays. Data files, created by `shelve`, from unknown sources are not secure and should not be used. REF	

shelve.open(*file_name*)	Method of shelve
Closes a `shelve` for reading and writing. Create a new one of it does not exist. REF	

```
1| iimport shelve
2|
3| # shelve data
4| with shelve.open('demoshelve') as dat:
5|     dat['some_int'] = 999
6|     dat['a_float'] = 87.654
7|     dat['message'] = "hi there"
8|     dat['car_collection'] = [["Volvo", "240", 1993],
   ["Ford","Mustang",1972],["VW","Beetle",1978]]
9|
10|    print("sample data was written")

sample data was written
```

```
1| import shelve
2|
3| # get data from the shelve
4| with shelve.open('demoshelve') as stuff:
```

```
    print("an integer ->", stuff['some_int'])
    print("a float ->", stuff['a_float'])
    print("a string ->", stuff['message'])
    print("a list ->", stuff['car_collection'])
```

```
an integer -> 999
a float -> 87.654
a string -> hi there
a list -> [['Volvo', '240', 1993], ['Ford', 'Mustang', 1972],
['VW', 'Beetle', 1978]]
```

NOTE: It is recommended that when accessing values from either a `dbm` or `shelve` database that you trap for errors; check to see if the key exists (using the `in` operator); or use the `.get(key, default)` method with a default value.

Sample Program — Number Guess with History

This sample program plays the classic number guessing game where after each guess you are given a clue; higher or lower. It uses the shelve module to store how many times the game has been played and the fewest number of guesses that were made to win. It creates a local file called "numberguess.dat" to store this persistent information.

```
import random
import shelve

# save constants into variales
DATABASENAME = "numberguess"
PLAYED = "played_times"
PLAYED_DEFAULT = 0
FEWEST = "fewest_guesses"
FEWEST_DEFAULT = 100

print("number guessing game")
with shelve.open(DATABASENAME) as db:
    played_times = db.get(PLAYED, PLAYED_DEFAULT)
    fewest_guesses = db.get(FEWEST, FEWEST_DEFAULT)

    print('this game has been played', played_times, 'times')
    print("can you beat", fewest_guesses, 'guesses?')

#
```

```
20| n = random.randint(1,100)
21| print("i am thinking about a number from 1 to 100.")
22| print("can you guess it?")
23| print("you will get a clue after each guess.")
24| #
25| guesses = 0
26| while True:
27|     try:
28|         guess = int(input("enter your guess (or -1 to quit)?"))
29|         if guess== -1:
30|             print('good bye')
31|             break
32|         if (guess >= 1 and guess <= 100):
33|             guesses = guesses + 1
34|
35|             if guess == n:
36|                 print('you got it in', guesses, 'guesses.')
37|                 # reopen shelve and update
38|                 with shelve.open(DATABASENAME) as db:
39|                     db[PLAYED] = db.get(PLAYED, PLAYED_DEFAULT)
   + 1
40|                     if guesses < db.get(FEWEST, FEWEST_DEFAULT):
41|                         db[FEWEST] = guesses
42|                 break
43|             elif guess < n:
44|                 print('you need to guess higher')
45|             else:
46|                 print('you need to guess lower')
47|         else:
48|             print("your guess should be between 1 and 100")
49|     except:
50|         print("you need to enter an integer.")
51| print('thanks for playing')
```

```
number guessing game
this game has been played 4 times
can you beat 6 guesses?
i am thinking about a number from 1 to 100.
can you guess it?
you will get a clue after each guess.
enter your guess (or -1 to quit)?59
you need to guess higher
enter your guess (or -1 to quit)?77
```

```
you need to guess lower
enter your guess (or -1 to quit)?68
you got it in 3 guesses.
thanks for playing
```


Summary

Goes here

Important Terms

here

Exercises

Here

Word Search

References