Theory: Shelve

25 minutes 0 / 5 problems solved

Imagine you have a big and beautiful library with all kinds of books. But it is a Schrödinger kind of library, once you leave the room, it disappears. You can't give a book to a friend or enjoy reading it in another room. This metaphor explains what happens to most Python objects — they are accessible only when the program is running, and afterwards, they just cease to exist and, if needed, have to be created again.

One of the basic and convenient data types for storing various kinds of objects in Python are dictionaries. A dictionary object shares the same destiny with many other objects in Python — it's not available after a program has been executed. But we have a workaround called `shelve`.

`shelve` is a Python module that lets you create persistent dictionaries that are pretty similar to databases by their nature. They can be stored locally on your machine, and used later with the same structure and functionality. The only limitation is that only strings can be keys, but as values, you can store any Python objects. Cool, isn't it? Let's dig deeper and create a library that will be available anywhere, anytime.

§1. Opening

To start with, we need to import the module.

```
import shelve
```

Now, we can create a new persistent dictionary or open the existing one using the `open()` function:

```
lib = shelve.open("my_library", flag="c", writeback=False)
```

An instance like this is called a **shelf**. Different platforms may give different results. Windows would generate three files: "`my_library.dir`", "`my_library.dat`", "`my_library.bak`"; on Linux there will only be "`my_library`", macOS would have only "`my_library.db`". This is due to the specifics of the operating systems, so don't think of this too much right now; you should just be able to recognize these files in the folder.

You may notice that some parameters are passed to the function when creating the object in the example above. Let's go through them one by one.

The first one is the string containing a filename. Note that you shouldn't add any extension to the filename. The second one, called a flag, specifies the opening mode. There are four values you can pass to the flag parameter:

Value	Meaning
"r"	opens an existing file for reading
"w"	opens an existing file for reading and writing
"c"	opens an existing file for reading and writing, creating it if it doesn't exist
"n"	creates a new empty file, opens it for reading and writing

The function creates a shelf file by default, it is also automatically opened for reading and writing, so, in our case, we do not need to specify the "c" flag value.

There is also a "writeback" parameter which, if `True`, is responsible for storing all values in memory and for writing them back when the file is closed. We'll discuss this parameter in detail later.

Current topic:

Shelve

Topic depends on:

Stage 1

Dictionary methods

Table of contents:

↑ Shelve

§1. Opening

§2. What can you do with a persistent dictionary?

§3. Writeback and sync()

§4. Closing

§5. Summary

Feedback & Comments

§2. What can you do with a persistent dictionary?

As an answer to this question — everything you may want to do with a "usual" dictionary. Let's, for example, add some books to our library:

```
lib["A Song of Ice and Fire"] = ["A Game of Thrones", "A Clash of Kings", "A Storm of Swords", "A Feast for Crows", "A Dance with Dragons", "The Winds of Winter", "A Dream of Spring"]
lib["The Hunger Games"] = ["The Hunger Games", "Catching Fire", "Mockingjay"]
lib["The Girl with the Dragon Tattoo"] = ["The Girl with the Dragon Tattoo", "The Girl Who Played with Fire", "The Girl Who Kicked the Hornets"]
```

Now, imagine that last week your friend had a birthday and you gave them "The Girl with the Dragon Tattoo" as a present. It's not in the library anymore, so let's delete it and compare the lengths before and after to check if the job was done:

```
print(len(lib))  # 3
del lib["The Girl with the Dragon Tattoo"]
print(len(lib))  # 2
```

As an alternative, you can also do the following:

```
print("The Girl with the Dragon Tattoo" in lib)  # False
```

In case you want to print your persistent dictionary to see the contents, keep in mind this kind of dictionary is different so you can't achieve it by printing:

```
print(lib)  # this will print:
#  <shelve.DbfilenameShelf object at 0x000001FF6AD48308>
```

What you see here is actually a memory address, where the object is stored. There are many ways to get what you want depending on your particular needs. You can, for example, go through the persistent dictionary and print it key by key and value by value:

```
for key in lib:
    print(key + ": ", lib[key])


# A Song of Ice and Fire: ['A Game of Thrones', 'A Clash of Kings', 'A Storm of Swords', 'A Feast
# for Crows', 'A Dance with Dragons', 'The Winds of Winter', 'A Dream of Spring']
# The Hunger Games: ['The Hunger Games', 'Catching Fire', 'Mockingjay']
```

Don't forget that iterating through a large amount of data might take a lot of time.

Speaking of time, let's look closer at the "writeback" parameter.

§3. Writeback and sync()

We added "A Song of Ice and Fire" book series key as values, but the last two books aren't out yet, so you can't have them in your library. In order to correct that, you would normally do something like this:

```
del lib["A Song of Ice and Fire"][6]
del lib["A Song of Ice and Fire"][5]
```

But, if you check the length of the list stored as values, you are going to get:

```
len(lib["A Song of Ice and Fire"])  # 7
```

Why so? The thing is, every time you'd like to get a value from the shelf, Python returns a copy of that value. So, in the example above, you actually modify a copy but not the value itself. As for our shelf, this happens because the "writeback" parameter was set to `False` (which is by default), so the accessed values won't be written back and, to modify a value, you will have to reassign it to the shelf explicitly:

```
temp = lib["A Song of Ice and Fire"]  # a copy is made
temp.pop(6)  # modify the copy
temp.pop(5)
lib["A Song of Ice and Fire"] = temp  # assign a new value to the key
print(len(lib["A Song of Ice and Fire"]))  # check the result
# 5
```

It's worth noticing that it is not a problem for immutable objects (like strings and integers, for example). If we stored just the number of books as values instead of their names, a new book could be added like this:

```
lib["The Godfather"] = 4
lib["The Godfather"] += 1
print(lib["The Godfather"])  # 5
```

This is because Python cannot modify an immutable object, so it recounts and reassigns a new value itself.

Now let's see how the same can be achieved with the "writeback" parameter set to `True`:

```
lib = shelve.open("my_library", flag="c", writeback=True)
print(len(lib["A Song of Ice and Fire"]))  # 7
del lib["A Song of Ice and Fire"][6]
del lib["A Song of Ice and Fire"][5]
print(len(lib["A Song of Ice and Fire"]))  # 5
```

The difference is pretty clear. Although, it may seem that using the `writeback=True` is more logical and easier, note that to be able to modify an object in the shelf, Python needs to store all the items of the program in memory. When closing the shelf, if `writeback` is `True`, all the accessed data is written back to the shelf. If the shelf is large, this procedure may significantly lower the program performance.

If `writeback` is `True`, you may explicitly write back to the shelf all the modified entries and locally synchronize the persistent dictionary whenever you like. To do so, call `lib.sync()`: it will update the values in the shelf. When closing the file, `sync()` is called automatically. Don't forget, however, that if `writeback` was `False`, you wouldn't be able to call `sync()` at all.

§4. Closing

The official Python Shelve Documentation recommends closing the shelf explicitly with the `close()` method, `lib.close()` instead of relying on the automatic closing feature. Do that either at the very end of the program code or when you're sure that you won't need it anymore, since you will get *ValueError* if you try to operate with the closed shelf.

If you don't want to worry about closing the shelf, you may use the context manager. It'll do the closing automatically after you're done:

```
with shelve.open("my_library") as lib:
    # do all the stuff here
```

§5. Summary

In this topic, we learned about the Python `shelve` module and how we can use it. The main `shelve` methods are `open()`, `sync()` and `close()`, however, shelf objects support all dictionary methods and also allow you to store your data as a file on disk persistently. If you're interested in a more detailed explanation, feel free to check the official Python documentation.

Report a typo

22 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (5) Hints (0) Useful links (0) Show discussion