mmap — Memory-map Files

Purpose: Memory-map files instead of reading the contents directly.

Memory-mapping a file uses the operating system virtual memory system to access the data on the file system directly, instead of using normal I/O functions. Memory-mapping typically improves I/O performance because it does not involve a separate system call for each access and it does not require copying data between buffers – the memory is accessed directly by both the kernel and the user application.

Memory-mapped files can be treated as mutable strings or file-like objects, depending on the need. A mapped file supports the expected file API methods, such as `close()`, `flush()`, `read()`, `readline()`, `seek()`, `tell()`, and `write()`. It also supports the string API, with features such as slicing and methods like `find()`.

All of the examples use the text file `lorem.txt`, containing a bit of Lorem Ipsum. For reference, the text of the file is

```
# lorem.txt

Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Donec egestas, enim et consectetuer ullamcorper, lectus ligula rutrum leo,
a elementum elit tortor eu quam. Duis tincidunt nisi ut ante. Nulla
facilisi. Sed tristique eros eu libero. Pellentesque vel
arcu. Vivamus purus orci, iaculis ac, suscipit sit amet, pulvinar eu,
lacus. Praesent placerat tortor sed nisl. Nunc blandit diam egestas
dui. Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Aliquam viverra fringilla
leo. Nulla feugiat augue eleifend nulla. Vivamus mauris. Vivamus sed
mauris in nibh placerat egestas. Suspendisse potenti. Mauris
massa. Ut eget velit auctor tortor blandit sollicitudin. Suspendisse
imperdiet justo.
```

Note

There are differences in the arguments and behaviors for `mmap()` between Unix and Windows, which are not fully discussed here. For more details, refer to the standard library documentation.

Reading

Use the `mmap()` function to create a memory-mapped file. The first argument is a file descriptor, either from the `fileno()` method of a `file` object or from `os.open()`. The caller is responsible for opening the file before invoking `mmap()`, and closing it after it is no longer needed.

The second argument to `mmap()` is a size in bytes for the portion of the file to map. If the value is `0`, the entire file is mapped. If the size is larger than the current size of the file, the file is extended.

Note

Windows does not support creating a zero-length mapping.

An optional keyword argument, `access`, is supported by both platforms. Use `ACCESS_READ` for read-only access, `ACCESS_WRITE` for write-through (assignments to the memory go directly to the file), or `ACCESS_COPY` for copy-on-write (assignments to memory are not written to the file).

```
# mmap_read.py

import mmap

with open('lorem.txt', 'r') as f:
    with mmap.mmap(f.fileno(), 0,
                   access=mmap.ACCESS_READ) as m:
        print('First 10 bytes via read :', m.read(10))
```

```
        print('First 10 bytes via slice:', m[:10])
        print('2nd   10 bytes via read :', m.read(10))
```

The file pointer tracks the last byte accessed through a slice operation. In this example, the pointer moves ahead 10 bytes after the first read. It is then reset to the beginning of the file by the slice operation, and moved ahead 10 bytes again by the slice. After the slice operation, calling `read()` again gives the bytes 11-20 in the file.

```
$ python3 mmap_read.py

First 10 bytes via read : b'Lorem ipsu'
First 10 bytes via slice: b'Lorem ipsu'
2nd   10 bytes via read : b'm dolor si'
```

Writing

To set up the memory mapped file to receive updates, start by opening it for appending with mode `'r+'` (not `'w'`) before mapping it. Then use any of the API methods that change the data (`write()`, assignment to a slice, etc.).

The next example uses the default access mode of `ACCESS_WRITE` and assigning to a slice to modify part of a line in place.

```
# mmap_write_slice.py

import mmap
import shutil

# Copy the example file
shutil.copyfile('lorem.txt', 'lorem_copy.txt')

word = b'consectetuer'
reversed = word[::-1]
print('Looking for    :', word)
print('Replacing with :', reversed)

with open('lorem_copy.txt', 'r+') as f:
    with mmap.mmap(f.fileno(), 0) as m:
        print('Before:\n{}'.format(m.readline().rstrip()))
        m.seek(0)  # rewind

        loc = m.find(word)
        m[loc:loc + len(word)] = reversed
        m.flush()

        m.seek(0)  # rewind
        print('After :\n{}'.format(m.readline().rstrip()))

        f.seek(0)  # rewind
        print('File  :\n{}'.format(f.readline().rstrip()))
```

The word “`consectetuer`” is replaced in the middle of the first line in memory and in the file.

```
$ python3 mmap_write_slice.py

Looking for    : b'consectetuer'
Replacing with : b'reutetcesnoc'
Before:
b'Lorem ipsum dolor sit amet, consectetuer adipiscing elit.'
After :
b'Lorem ipsum dolor sit amet, reutetcesnoc adipiscing elit.'
File  :
Lorem ipsum dolor sit amet, reutetcesnoc adipiscing elit.
```

Copy Mode

Using the access setting `ACCESS_COPY` does not write changes to the file on disk.

```
# mmap_write_copy.py

import mmap
```

```
import shutil

# Copy the example file
shutil.copyfile('lorem.txt', 'lorem_copy.txt')

word = b'consectetuer'
reversed = word[::-1]

with open('lorem_copy.txt', 'r+') as f:
    with mmap.mmap(f.fileno(), 0,
                   access=mmap.ACCESS_COPY) as m:
        print('Memory Before:\n{}'.format(
            m.readline().rstrip()))
        print('File Before  :\n{}\n'.format(
            f.readline().rstrip()))

        m.seek(0)  # rewind
        loc = m.find(word)
        m[loc:loc + len(word)] = reversed

        m.seek(0)  # rewind
        print('Memory After :\n{}'.format(
            m.readline().rstrip()))

        f.seek(0)
        print('File After   :\n{}'.format(
            f.readline().rstrip()))
```

It is necessary to rewind the file handle in this example separately from the `mmap` handle, because the internal state of the two objects is maintained separately.

```
$ python3 mmap_write_copy.py

Memory Before:
b'Lorem ipsum dolor sit amet, consectetuer adipiscing elit.'
File Before  :
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.

Memory After :
b'Lorem ipsum dolor sit amet, reutetcesnoc adipiscing elit.'
File After   :
Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
```

Regular Expressions

Since a memory mapped file can act like a string, it can be used with other modules that operate on strings, such as regular expressions. This example finds all of the sentences with “`nulla`” in them.

```
# mmap_regex.py

import mmap
import re

pattern = re.compile(rb'(\.\W+)?([^.]?nulla[^.]*?\.)',
                     re.DOTALL | re.IGNORECASE | re.MULTILINE)

with open('lorem.txt', 'r') as f:
    with mmap.mmap(f.fileno(), 0,
                   access=mmap.ACCESS_READ) as m:
        for match in pattern.findall(m):
            print(match[1].replace(b'\n', b' '))
```

Because the pattern includes two groups, the return value from `findall()` is a sequence of tuples. The `print` statement pulls out the matching sentence and replaces newlines with spaces so each result prints on a single line.

```
$ python3 mmap_regex.py

b'Nulla facilisi.'
b'Nulla feugiat augue eleifend nulla.'
```

See also

- Standard library documentation for mmap
- Python 2 to 3 porting notes for mmap
- os – The os module.
- re – Regular expressions.

filecmp — Compare Files

codecs — String Encoding and Decoding

This page was last updated 2016-12-28.

Navigation

filecmp — Compare Files
codecs — String Encoding and Decoding

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?