18.9. `mmap` — Memory-mapped file support

Memory-mapped file objects behave like both `bytearray` and like file objects. You can use mmap objects in most places where `bytearray` are expected; for example, you can use the `re` module to search through a memory-mapped file. You can also change a single byte by doing `obj[index] = 97`, or change a subsequence by assigning to a slice: `obj[i1:i2] = b'...'`. You can also read and write data starting at the current file position, and `seek()` through the file to different positions.

A memory-mapped file is created by the `mmap` constructor, which is different on Unix and on Windows. In either case you must provide a file descriptor for a file opened for update. If you wish to map an existing Python file object, use its `fileno()` method to obtain the correct value for the *fileno* parameter. Otherwise, you can open the file using the `os.open()` function, which returns a file descriptor directly (the file still needs to be closed when done).

Note: If you want to create a memory-mapping for a writable, buffered file, you should `flush()` the file first. This is necessary to ensure that local modifications to the buffers are actually available to the mapping.

For both the Unix and Windows versions of the constructor, *access* may be specified as an optional keyword parameter. *access* accepts one of three values: `ACCESS_READ`, `ACCESS_WRITE`, or `ACCESS_COPY` to specify read-only, write-through or copy-on-write memory respectively. *access* can be used on both Unix and Windows. If *access* is not specified, Windows mmap returns a write-through mapping. The initial memory values for all three access types are taken from the specified file. Assignment to an `ACCESS_READ` memory map raises a `TypeError` exception. Assignment to an `ACCESS_WRITE` memory map affects both memory and the underlying file. Assignment to an `ACCESS_COPY` memory map affects memory but does not update the underlying file.

To map anonymous memory, -1 should be passed as the fileno along with the length.

class `mmap.`**`mmap`**(*fileno*, *length*, *tagname=None*, *access=ACCESS_DEFAULT*[, *offset*])

(Windows version) Maps *length* bytes from the file specified by the file handle *fileno*, and creates a mmap object. If *length* is larger than the current size of the file, the file is extended to contain *length* bytes. If *length* is `0`, the maximum length of the map is the current size of the file, except that if the file is empty

Windows raises an exception (you cannot create an empty mapping on Windows).

tagname, if specified and not `None`, is a string giving a tag name for the mapping. Windows allows you to have many different mappings against the same file. If you specify the name of an existing tag, that tag is opened, otherwise a new tag of this name is created. If this parameter is omitted or `None`, the mapping is created without a name. Avoiding the use of the tag parameter will assist in keeping your code portable between Unix and Windows.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of the ALLOCATIONGRANULARITY.

class `mmap.`**`mmap`**(*fileno*, *length*, *flags=MAP_SHARED*, *prot=PROT_WRITE|PROT_READ*, *access=ACCESS_DEFAULT*[, *offset*])

(Unix version) Maps *length* bytes from the file specified by the file descriptor *fileno*, and returns a mmap object. If *length* is `0`, the maximum length of the map will be the current size of the file when `mmap` is called.

flags specifies the nature of the mapping. `MAP_PRIVATE` creates a private copy-on-write mapping, so changes to the contents of the mmap object will be private to this process, and `MAP_SHARED` creates a mapping that's shared with all other processes mapping the same areas of the file. The default value is `MAP_SHARED`.

prot, if specified, gives the desired memory protection; the two most useful values are `PROT_READ` and `PROT_WRITE`, to specify that the pages may be read or written. *prot* defaults to `PROT_READ | PROT_WRITE`.

access may be specified in lieu of *flags* and *prot* as an optional keyword parameter. It is an error to specify both *flags*, *prot* and *access*. See the description of *access* above for information on how to use this parameter.

offset may be specified as a non-negative integer offset. mmap references will be relative to the offset from the beginning of the file. *offset* defaults to 0. *offset* must be a multiple of the PAGESIZE or ALLOCATIONGRANULARITY.

To ensure validity of the created memory mapping the file specified by the descriptor *fileno* is internally automatically synchronized with physical backing store on Mac OS X and OpenVMS.

This example shows a simple way of using `mmap`:

```
import mmap

# write a simple example file
```

```
with open("hello.txt", "wb") as f:
    f.write(b"Hello Python!\n")

with open("hello.txt", "r+b") as f:
    # memory-map the file, size 0 means whole file
    mm = mmap.mmap(f.fileno(), 0)
    # read content via standard file methods
    print(mm.readline())  # prints b"Hello Python!\n"
    # read content via slice notation
    print(mm[:5])  # prints b"Hello"
    # update content using slice notation;
    # note that new content must have same size
    mm[6:] = b" world!\n"
    # ... and read again using standard file methods
    mm.seek(0)
    print(mm.readline())  # prints b"Hello  world!\n"
    # close the map
    mm.close()
```

`mmap` can also be used as a context manager in a `with` statement:

```
import mmap

with mmap.mmap(-1, 13) as mm:
    mm.write(b"Hello world!")
```

New in version 3.2: Context manager support.

The next example demonstrates how to create an anonymous map and exchange data between the parent and child processes:

```
import mmap
import os

mm = mmap.mmap(-1, 13)
mm.write(b"Hello world!")

pid = os.fork()

if pid == 0:  # In a child process
    mm.seek(0)
    print(mm.readline())

    mm.close()
```

Memory-mapped file objects support the following methods:

`close`()

Closes the mmap. Subsequent calls to other methods of the object will result in a ValueError exception being raised. This will not close the open file.

`closed`

`True` if the file is closed.

New in version 3.2.

`find`(*sub*[, *start*[, *end*]])

Returns the lowest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

Changed in version 3.5: Writable bytes-like object is now accepted.

`flush`([*offset*[, *size*]])

Flushes changes made to the in-memory copy of a file back to disk. Without use of this call there is no guarantee that changes are written back before the object is destroyed. If *offset* and *size* are specified, only changes to the given range of bytes will be flushed to disk; otherwise, the whole extent of the mapping is flushed.

(Windows version) A nonzero value returned indicates success; zero indicates failure.

(Unix version) A zero value is returned to indicate success. An exception is raised when the call failed.

`move`(*dest*, *src*, *count*)

Copy the *count* bytes starting at offset *src* to the destination index *dest*. If the mmap was created with `ACCESS_READ`, then calls to move will raise a `TypeError` exception.

`read`([*n*])

Return a `bytes` containing up to *n* bytes starting from the current file position. If the argument is omitted, `None` or negative, return all bytes from the current file position to the end of the mapping. The file position is updated to point after the bytes that were returned.

Changed in version 3.3: Argument can be omitted or `None`.

`read_byte`()

Returns a byte at the current file position as an integer, and advances the file position by 1.

`readline`()

Returns a single line, starting at the current file position and up to the next newline.

`resize`(*newsize*)

Resizes the map and the underlying file, if any. If the mmap was created with `ACCESS_READ` or `ACCESS_COPY`, resizing the map will raise a `TypeError` exception.

`rfind`(*sub*[, *start*[, *end*]])

Returns the highest index in the object where the subsequence *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Returns `-1` on failure.

Changed in version 3.5: Writable bytes-like object is now accepted.

`seek`(*pos*[, *whence*])

Set the file's current position. *whence* argument is optional and defaults to `os.SEEK_SET` or `0` (absolute file positioning); other values are `os.SEEK_CUR` or `1` (seek relative to the current position) and `os.SEEK_END` or `2` (seek relative to the file's end).

`size`()

Return the length of the file, which can be larger than the size of the memory-mapped area.

`tell`()

Returns the current position of the file pointer.

`write`(*bytes*)

Write the bytes in *bytes* into memory at the current position of the file pointer and return the number of bytes written (never less than `len(bytes)`, since if the write fails, a `ValueError` will be raised). The file position is updated to point after the bytes that were written. If the mmap was created with `ACCESS_READ`, then writing to it will raise a `TypeError` exception.

Changed in version 3.5: Writable bytes-like object is now accepted.

Changed in version 3.6: The number of bytes written is now returned.

`write_byte`(*byte*)

Write the integer *byte* into memory at the current position of the file pointer; the file position is advanced by `1`. If the mmap was created with `ACCESS_READ`, then writing to it will raise a `TypeError` exception.