

---

# **smmap Documentation**

***Release 0.8.0***

**Sebastian Thiel**

**Oct 15, 2021**



---

## Contents

---

<b>1</b>	<b>Motivation</b>	<b>3</b>
<b>2</b>	<b>Overview</b>	<b>5</b>
<b>3</b>	<b>Prerequisites</b>	<b>7</b>
<b>4</b>	<b>Limitations</b>	<b>9</b>
<b>5</b>	<b>Installing smmap</b>	<b>11</b>
<b>6</b>	<b>Homepage and Links</b>	<b>13</b>
<b>7</b>	<b>License Information</b>	<b>15</b>
<b>8</b>	<b>Usage Guide</b>	<b>17</b>
8.1	Design . . . . .	17
8.2	Memory Managers . . . . .	17
8.3	Buffers . . . . .	19
<b>9</b>	<b>API Reference</b>	<b>21</b>
9.1	Mapped Memory Managers . . . . .	21
9.2	Buffers . . . . .	24
9.3	Utilities . . . . .	25
<b>10</b>	<b>Changelog</b>	<b>27</b>
10.1	v5.0.0 . . . . .	27
10.2	v4.0.0 . . . . .	27
10.3	v3.0.5 . . . . .	27
10.4	v3.0.4 . . . . .	27
10.5	v3.0.2 . . . . .	28
10.6	v3.0.1 . . . . .	28
10.7	v0.9.0 . . . . .	28
10.8	v0.8.5 . . . . .	28
10.9	v0.8.4 . . . . .	28
10.10	v0.8.3 . . . . .	28
10.11	v0.8.1 . . . . .	28
10.12	v0.8.0 . . . . .	28

<b>11 Indices and tables</b>	<b>29</b>
<b>Python Module Index</b>	<b>31</b>
<b>Index</b>	<b>33</b>

**smmap** is a pure python implementation of a sliding memory map to help unifying memory mapped access on 32 and 64 bit systems and to help managing resources more efficiently.

Contents:



# CHAPTER 1

---

## Motivation

---

When reading from many possibly large files in a fashion similar to random access, it is usually the fastest and most efficient to use memory maps.

Although memory maps have many advantages, they represent a very limited system resource as every map uses one file descriptor, whose amount is limited per process. On 32 bit systems, the amount of memory you can have mapped at a time is naturally limited to theoretical 4GB of memory, which may not be enough for some applications.





## CHAPTER 2

---

### Overview

---

Smmmap wraps an interface around mmap and tracks the mapped files as well as the amount of clients who use it. If the system runs out of resources, or if a memory limit is reached, it will automatically unload unused maps to allow continued operation.

To allow processing large files even on 32 bit systems, it allows only portions of the file to be mapped. Once the user reads beyond the mapped region, smmap will automatically map the next required region, unloading unused regions using a LRU algorithm.

Although the library can be used most efficiently with its native interface, a Buffer implementation is provided to hide these details behind a simple string-like interface.

For performance critical 64 bit applications, a simplified version of memory mapping is provided which always maps the whole file, but still provides the benefit of unloading unused mappings on demand.



## CHAPTER 3

---

### Prerequisites

---

- Python 3.6+
- OSX, Windows or Linux

The package was tested on all of the previously mentioned configurations.



## CHAPTER 4

---

### Limitations

---

- The memory access is read-only by design.



---

### Installing smmap

---

Its easiest to install smmap using the *pip* program:

```
$ pip install smmap
```

As the command will install smmap in your respective python distribution, you will most likely need root permissions to authorize the required changes.

If you have downloaded the source archive, the package can be installed by running the `setup.py` script:

```
$ python setup.py install
```

It is advised to have a look at the [Usage Guide](#) for a brief introduction on the different database implementations.





## CHAPTER 6

---

### Homepage and Links

---

The project is home on github at <https://github.com/gitpython-developers/smmap>.

The latest source can be cloned from github as well:

- <git://github.com/gitpython-developers/smmap.git>

For support, please use the git-python mailing list:

- <http://groups.google.com/group/git-python>

Issues can be filed on github:

- <https://github.com/gitpython-developers/smmap/issues>



## CHAPTER 7

---

### License Information

---

*smmap* is licensed under the New BSD License.



This text briefly introduces you to the basic design decisions and accompanying classes.

### 8.1 Design

Per application, there is *MemoryManager* which is held as static instance and used throughout the application. It can be configured to keep your resources within certain limits.

To access mapped regions, you require a cursor. Cursors point to exactly one file and serve as handles into it. As long as it exists, the respective memory region will remain available.

For convenience, a buffer implementation is provided which handles cursors and resource allocation behind its simple buffer like interface.

### 8.2 Memory Managers

There are two types of memory managers, one uses *static* windows, the other one uses *sliding* windows. A window is a region of a file mapped into memory. Although the names might be somewhat misleading as technically windows are always static, the *sliding* version will allocate relatively small windows whereas the *static* version will always map the whole file.

The *static* manager does nothing more than keeping a client count on the respective memory maps which always map the whole file, which allows to make some assumptions that can lead to simplified data access and increased performance, but reduces the compatibility to 32 bit systems or giant files.

The *sliding* memory manager therefore should be the default manager when preparing an application for handling huge amounts of data on 32 bit and 64 bit platforms:

```
import smmap
# This instance should be globally available in your application
# It is configured to be well suitable for 32-bit or 64 bit applications.
mman = smmap.SlidingWindowMapManager()
```

(continues on next page)

(continued from previous page)

```
# the manager provides much useful information about its current state
# like the amount of open file handles or the amount of mapped memory
mman.num_file_handles()
mman.mapped_memory_size()
# and many more ...
```

## 8.2.1 Cursors

*Cursors* are handles that point onto a window, i.e. a region of a file mapped into memory. From them you may obtain a buffer through which the data of that window can actually be accessed:

```
import smmap.test.lib
fc = smmap.test.lib.FileCreator(1024*1024*8, "test_file")

# obtain a cursor to access some file.
c = mman.make_cursor(fc.path)

# the cursor is now associated with the file, but not yet usable
assert c.is_associated()
assert not c.is_valid()

# before you can use the cursor, you have to specify a window you want to
# access. The following just says you want as much data as possible starting
# from offset 0.
# To be sure your region could be mapped, query for validity
assert c.use_region().is_valid()          # use_region returns self

# once a region was mapped, you must query its dimension regularly
# to assure you don't try to access its buffer out of its bounds
assert c.size()
c.buffer()[0]                # first byte
c.buffer()[1:10]             # first 9 bytes
c.buffer()[c.size()-1]       # last byte

# its recommended not to create big slices when feeding the buffer
# into consumers (e.g. struct or zlib).
# Instead, either give the buffer directly, or use python's buffer command.
buffer(c.buffer(), 1, 9)     # first 9 bytes without copying them

# you can query absolute offsets, and check whether an offset is included
# in the cursor's data.
assert c ofs_begin() < c ofs_end()
assert c.includes_ofs(100)

# If you are over out of bounds with one of your region requests, the
# cursor will be come invalid. It cannot be used in that state
assert not c.use_region(fc.size, 100).is_valid()
# map as much as possible after skipping the first 100 bytes
assert c.use_region(100).is_valid()

# You can explicitly free cursor resources by unusing the cursor's region
c.unuse_region()
assert not c.is_valid()
```

Now you would have to write your algorithms around this interface to properly slide through huge amounts of data.

Alternatively you can use a convenience interface.

## 8.3 Buffers

To make first use easier, at the expense of performance, there is a Buffer implementation which uses a cursor underneath.

With it, you can access all data in a possibly huge file without having to take care of setting the cursor to different regions yourself:

```
# Create a default buffer which can operate on the whole file
buf = smmap.SlidingWindowMapBuffer(mman.make_cursor(fc.path))

# you can use it right away
assert buf.cursor().is_valid()

buf[0]      # access the first byte
buf[-1]     # access the last ten bytes on the file
buf[-10:]   # access the last ten bytes

# If you want to keep the instance between different accesses, use the
# dedicated methods
buf.end_access()
assert not buf.cursor().is_valid() # you cannot use the buffer anymore
assert buf.begin_access(offset=10) # start using the buffer at an offset

# it will stop using resources automatically once it goes out of scope
```

### 8.3.1 Disadvantages

Buffers cannot be used in place of strings or maps, hence you have to slice them to have valid input for the sorts of struct and zlib. A slice means a lot of data handling overhead which makes buffers slower compared to using cursors directly.





## 9.1 Mapped Memory Managers

Module containing a memory memory manager which provides a sliding window on a number of memory mapped files

**class** `smmap.mman.StaticWindowMapManager` (*window\_size=0*, *max\_memory\_size=0*,  
*max\_open\_handles=9223372036854775807*)

Provides a manager which will produce single size cursors that are allowed to always map the whole file.

Clients must be written to specifically know that they are accessing their data through a StaticWindowMapManager, as they otherwise have to deal with their window size.

These clients would have to use a SlidingWindowMapBuffer to hide this fact.

This type will always use a maximum window size, and optimize certain methods to accommodate this fact

**MapRegionCls**

alias of `smmap.util.MapRegion`

**MapRegionListCls**

alias of `smmap.util.MapRegionList`

**MapWindowCls**

alias of `smmap.util.MapWindow`

**WindowCursorCls**

alias of `WindowCursor`

**collect ()**

Collect all available free-to-collect mapped regions :return: Amount of freed handles

**force\_map\_handle\_removal\_win** (*base\_path*)

ONLY AVAILABLE ON WINDOWS On windows removing files is not allowed if anybody still has it opened. If this process is ourselves, and if the whole process uses this memory manager (as far as the parent framework is concerned) we can enforce closing all memory maps whose path matches the given base path to allow the respective operation after all. The respective system must NOT access the closed

memory regions anymore ! This really may only be used if you know that the items which keep the cursors alive will not be using it anymore. They need to be recreated ! :return: Amount of closed handles

**Note:** does nothing on non-windows platforms

**make\_cursor** (*path\_or\_fd*)

**Returns** a cursor pointing to the given path or file descriptor. It can be used to map new regions of the file into memory

**Note:** if a file descriptor is given, it is assumed to be open and valid, but may be closed afterwards. To refer to the same file, you may reuse your existing file descriptor, but keep in mind that new windows can only be mapped as long as it stays valid. This is why the using actual file paths are preferred unless you plan to keep the file descriptor open.

**Note:** file descriptors are problematic as they are not necessarily unique, as two different files opened and closed in succession might have the same file descriptor id.

**Note:** Using file descriptors directly is faster once new windows are mapped as it prevents the file to be opened again just for the purpose of mapping it.

**mapped\_memory\_size** ()

**Returns** amount of bytes currently mapped in total

**max\_file\_handles** ()

**Returns** maximum amount of handles we may have opened

**max\_mapped\_memory\_size** ()

**Returns** maximum amount of memory we may allocate

**num\_file\_handles** ()

**Returns** amount of file handles in use. Each mapped region uses one file handle

**num\_open\_files** ()

Amount of opened files in the system

**window\_size** ()

**Returns** size of each window when allocating new regions

**class** smmap.mman.**SlidingWindowMapManager** (*window\_size=-1*, *max\_memory\_size=0*,  
*max\_open\_handles=9223372036854775807*)

Maintains a list of ranges of mapped memory regions in one or more files and allows to easily obtain additional regions assuring there is no overlap. Once a certain memory limit is reached globally, or if there cannot be more open file handles which result from each mmap call, the least recently used, and currently unused mapped regions are unloaded automatically.

**Note:** currently not thread-safe !

**Note:** in the current implementation, we will automatically unload windows if we either cannot create more memory maps (as the open file handles limit is hit) or if we have allocated more than a safe amount of memory already, which would possibly cause memory allocations to fail as our address space is full.

**class** smmap.mman.**WindowCursor** (*manager=None*, *regions=None*)

Pointer into the mapped region of the memory manager, keeping the map alive until it is destroyed and no other client uses it.

Cursors should not be created manually, but are instead returned by the SlidingWindowMapManager

**Note::** The current implementation is suited for static and sliding window managers, but it also means that it must be suited for the somewhat quite different sliding manager. It could be improved, but I see no real need to do so.

**assign** (*rhs*)

Assign *rhs* to this instance. This is required in order to get a real copy. Alternatively, you can copy an existing instance using the copy module

**buffer** ()

Return a buffer object which allows access to our memory region from our offset to the window size. Please note that it might be smaller than you requested when calling `use_region()`

**Note:** You can only obtain a buffer if this instance is `_valid()` !

**Note:** buffers should not be cached passed the duration of your access as it will prevent resources from being freed even though they might not be accounted for anymore !

**fd** ()

**Returns** file descriptor used to create the underlying mapping.

**Note:** it is not required to be valid anymore :raise `ValueError`: if the mapping was not created by a file descriptor

**file\_size** ()

**Returns** size of the underlying file

**includes\_ofs** (*ofs*)

**Returns** True if the given absolute offset is contained in the cursors current region

**Note:** cursor must be valid for this to work

**is\_associated** ()

**Returns** True if we are associated with a specific file already

**is\_valid** ()

**Returns** True if we have a valid and usable region

**map** ()

**Returns** the underlying raw memory map. Please not that the offset and size is likely to be different to what you set as offset and size. Use it only if you are sure about the region it maps, which is the whole file in case of `StaticWindowMapManager`

**ofs\_begin** ()

**Returns** offset to the first byte pointed to by our cursor

**Note:** only if `is_valid()` is True

**ofs\_end** ()

**Returns** offset to one past the last available byte

**path** ()

**Returns** path of the underlying mapped file

**Raises** `ValueError` – if attached path is not a path

**path\_or\_fd** ()

**Returns** path or file descriptor of the underlying mapped file

**region** ()

**Returns** our mapped region, or `None` if nothing is mapped yet

**Raises** `AssertionError` – if we have no current region. This is only useful for debugging

**size()**

**Returns** amount of bytes we point to

**unuse\_region()**

Unuse the current region. Does nothing if we have no current region

**Note:** the cursor unuses the region automatically upon destruction. It is recommended to un-use the region once you are done reading from it in persistent cursors as it helps to free up resource more quickly

**use\_region** (*offset=0, size=0, flags=0*)

Assure we point to a window which allows access to the given offset into the file

**Parameters**

- **offset** – absolute offset in bytes into the file
- **size** – amount of bytes to map. If 0, all available bytes will be mapped
- **flags** – additional flags to be given to `os.open` in case a file handle is initially opened for mapping. Has no effect if a region can actually be reused.

**Returns** this instance - it should be queried for whether it points to a valid memory region. This is not the case if the mapping failed because we reached the end of the file

**Note::** The size actually mapped may be smaller than the given size. If that is the case, either the file has reached its end, or the map was created between two existing regions

## 9.2 Buffers

Module with a simple buffer implementation using the memory manager

**class** `smmap.buf.SlidingWindowMapBuffer` (*cursor=None, offset=0, size=9223372036854775807, flags=0*)

A buffer like object which allows direct byte-wise object and slicing into memory of a mapped file. The mapping is controlled by the provided cursor.

The buffer is relative, that is if you map an offset, index 0 will map to the first byte at the offset you used during initialization or `begin_access`

**Note:** Although this type effectively hides the fact that there are mapped windows underneath, it can unfortunately not be used in any non-pure python method which needs a buffer or string

**begin\_access** (*cursor=None, offset=0, size=9223372036854775807, flags=0*)

Call this before the first use of this instance. The method was already called by the constructor in case sufficient information was provided.

For more information no the parameters, see the `__init__` method :param path: if cursor is None the existing one will be used. :return: True if the buffer can be used

**cursor()**

**Returns** the currently set cursor which provides access to the data

**end\_access()**

Call this method once you are done using the instance. It is automatically called on destruction, and should be called just in time to allow system resources to be freed.

Once you called `end_access`, you must call `begin access` before reusing this instance!

## 9.3 Utilities

Module containing a memory memory manager which provides a sliding window on a number of memory mapped files

`smmap.util.align_to_mmap(num, round_up)`

Align the given integer number to the closest page offset, which usually is 4096 bytes.

**Parameters** `round_up` – if True, the next higher multiple of page size is used, otherwise the lower `page_size` will be used (i.e. if True, 1 becomes 4096, otherwise it becomes 0)

**Returns** num rounded to closest page

`smmap.util.is_64_bit()`

**Returns** True if the system is 64 bit. Otherwise it can be assumed to be 32 bit

**class** `smmap.util.MapWindow(offset, size)`

Utility type which is used to snap windows towards each other, and to adjust their size

**align()**

Assures the previous window area is contained in the new one

**extend\_left\_to(window, max\_size)**

Adjust the offset to start where the given window on our left ends if possible, but don't make yourself larger than `max_size`. The resize will assure that the new window still contains the old window area

**extend\_right\_to(window, max\_size)**

Adjust the size to make our window end where the right window begins, but don't get larger than `max_size`

**classmethod from\_region(region)**

**Returns** new window from a region

**ofs\_end()**

**class** `smmap.util.MapRegion(path_or_fd, ofs, size, flags=0)`

Defines a mapped region of memory, aligned to pagesizes

**Note:** deallocates used region automatically on destruction

**buffer()**

**Returns** a buffer containing the memory

**client\_count()**

**Returns** number of clients currently using this region

**includes\_ofs(ofs)**

**Returns** True if the given offset can be read in our mapped region

**increment\_client\_count(ofs=1)**

Adjust the usage count by the given positive or negative offset. If usage count equals 0, we will auto-release our resources :return: True if we released resources, False otherwise. In the latter case, we can still be used

**map()**

**Returns** a memory map containing the memory

**ofs\_begin()**

**Returns** absolute byte offset to the first byte of the mapping

**ofs\_end()**

**Returns** Absolute offset to one byte beyond the mapping into the file

**release()**

Release all resources this instance might hold. Must only be called if there usage\_count() is zero

**size()**

**Returns** total size of the mapped region in bytes

**class** smmap.util.**MapRegionList** (*path\_or\_fd*)

List of MapRegion instances associating a path with a list of regions.

**file\_size()**

**Returns** size of file we manager

**path\_or\_fd()**

**Returns** path or file descriptor we are attached to

# CHAPTER 10

---

## Changelog

---

### 10.1 v5.0.0

- Dropped support 3.5
- Added support for Python 3.10

### 10.2 v4.0.0

- Dropped support for Python 2.7 and 3.4
- Added support for Python 3.7, 3.8, and 3.9
- Removed unused `exc.MemoryManagerError` and `exc.RegionCollectionError`

### 10.3 v3.0.5

- Restored Python 2 support removed in v3.0.2
- Changed release signature key to `27C50E7F590947D7273A741E85194C08421980C9`. See <https://keybase.io/byronbates> for proof of ownership.

### 10.4 v3.0.4

- Signed release (with correct key this time)

## 10.5 v3.0.2

- Signed release
- Switched to GitHub Actions for CI

## 10.6 v3.0.1

- Switched back to the smmap package name on PyPI and fixed the smmap2 mirror package (#44)
- Fixed setup.py long\_description rendering (#40)

## 10.7 v0.9.0

- Fixed issue with resources never being freed as mmmaps were never closed.
- Client counting is now done manually, instead of relying on python's reference count

## 10.8 v0.8.5

- Fixed Python 3.0-3.3 regression, which also causes smmap to become about 3 times slower depending on the code path. It's related to this bug (<http://bugs.python.org/issue15958>), which was fixed in python 3.4

## 10.9 v0.8.4

- Fixed Python 3 performance regression

## 10.10 v0.8.3

- Cleaned up code and assured it works sufficiently well with python 3

## 10.11 v0.8.1

- A single bugfix

## 10.12 v0.8.0

- Initial Release



# CHAPTER 11

---

## Indices and tables

---

- `genindex`
- `modindex`
- `search`



### S

`smmap.buf`, [24](#)  
`smmap.mman`, [21](#)  
`smmap.util`, [25](#)



## A

`align()` (*smmap.util.MapWindow method*), 25  
`align_to_mmap()` (*in module smmap.util*), 25  
`assign()` (*smmap.mman.WindowCursor method*), 22

## B

`begin_access()` (*smmap.buf.SlidingWindowMapBuffer method*), 24  
`buffer()` (*smmap.mman.WindowCursor method*), 23  
`buffer()` (*smmap.util.MapRegion method*), 25

## C

`client_count()` (*smmap.util.MapRegion method*), 25  
`collect()` (*smmap.mman.StaticWindowMapManager method*), 21  
`cursor()` (*smmap.buf.SlidingWindowMapBuffer method*), 24

## E

`end_access()` (*smmap.buf.SlidingWindowMapBuffer method*), 24  
`extend_left_to()` (*smmap.util.MapWindow method*), 25  
`extend_right_to()` (*smmap.util.MapWindow method*), 25

## F

`fd()` (*smmap.mman.WindowCursor method*), 23  
`file_size()` (*smmap.mman.WindowCursor method*), 23  
`file_size()` (*smmap.util.MapRegionList method*), 26  
`force_map_handle_removal_win()` (*smmap.mman.StaticWindowMapManager method*), 21  
`from_region()` (*smmap.util.MapWindow class method*), 25

## I

`includes_ofs()` (*smmap.mman.WindowCursor method*), 23  
`includes_ofs()` (*smmap.util.MapRegion method*), 25  
`increment_client_count()` (*smmap.util.MapRegion method*), 25  
`is_64_bit()` (*in module smmap.util*), 25  
`is_associated()` (*smmap.mman.WindowCursor method*), 23  
`is_valid()` (*smmap.mman.WindowCursor method*), 23

## M

`make_cursor()` (*smmap.mman.StaticWindowMapManager method*), 22  
`map()` (*smmap.mman.WindowCursor method*), 23  
`map()` (*smmap.util.MapRegion method*), 25  
`mapped_memory_size()` (*smmap.mman.StaticWindowMapManager method*), 22  
`MapRegion` (*class in smmap.util*), 25  
`MapRegionCls` (*smmap.mman.StaticWindowMapManager attribute*), 21  
`MapRegionList` (*class in smmap.util*), 26  
`MapRegionListCls` (*smmap.mman.StaticWindowMapManager attribute*), 21  
`MapWindow` (*class in smmap.util*), 25  
`MapWindowCls` (*smmap.mman.StaticWindowMapManager attribute*), 21  
`max_file_handles()` (*smmap.mman.StaticWindowMapManager method*), 22  
`max_mapped_memory_size()` (*smmap.mman.StaticWindowMapManager method*), 22

## N

`num_file_handles()`

*(smmap.mman.StaticWindowMapManager  
method)*, [22](#)  
`num_open_files()` (*smmap.mman.StaticWindowMapManager  
method*), [22](#)

## O

`ofs_begin()` (*smmap.mman.WindowCursor method*),  
[23](#)  
`ofs_begin()` (*smmap.util.MapRegion method*), [25](#)  
`ofs_end()` (*smmap.mman.WindowCursor method*), [23](#)  
`ofs_end()` (*smmap.util.MapRegion method*), [25](#)  
`ofs_end()` (*smmap.util.MapWindow method*), [25](#)

## P

`path()` (*smmap.mman.WindowCursor method*), [23](#)  
`path_or_fd()` (*smmap.mman.WindowCursor  
method*), [23](#)  
`path_or_fd()` (*smmap.util.MapRegionList method*),  
[26](#)

## R

`region()` (*smmap.mman.WindowCursor method*), [23](#)  
`release()` (*smmap.util.MapRegion method*), [26](#)

## S

`size()` (*smmap.mman.WindowCursor method*), [23](#)  
`size()` (*smmap.util.MapRegion method*), [26](#)  
`SlidingWindowMapBuffer` (*class in smmap.buf*),  
[24](#)  
`SlidingWindowMapManager` (*class in  
smmap.mman*), [22](#)  
`smmap.buf` (*module*), [24](#)  
`smmap.mman` (*module*), [21](#)  
`smmap.util` (*module*), [25](#)  
`StaticWindowMapManager` (*class in  
smmap.mman*), [21](#)

## U

`unuse_region()` (*smmap.mman.WindowCursor  
method*), [24](#)  
`use_region()` (*smmap.mman.WindowCursor  
method*), [24](#)

## W

`window_size()` (*smmap.mman.StaticWindowMapManager  
method*), [22](#)  
`WindowCursor` (*class in smmap.mman*), [22](#)  
`WindowCursorCls` (*smmap.mman.StaticWindowMapManager  
attribute*), [21](#)