cbytesparse

Release 0.0.5

Andrea Zoppi

Feb 22, 2022

CONTENTS

CHAPTER

ONE

OVERVIEW

docs	
tests	
package	

Library to handle sparse bytes within a virtual memory space.

- Free software: BSD 2-Clause License

1.1 Objectives

This library aims to provide utilities to work with a *virtual memory*, which constsis in a virtual addressing space where sparse *chunks* of data can be stored.

In order to be easy to use, its interface should be close to that of a `bytearray`, which is the closest pythonic way to store dynamic data. The main downside of a `bytearray` is that it requires a contiguous data allocation starting from address 0. This is not good when sparse data have to be stored, such as when emulating the addressing space of a generic microcontroller.

The main idea is to provide a `bytearray`-like class with the possibility to internally hold the sparse *blocks* of data. A *block* is ideally a tuple `(start, data)` where *start* is the start address and *data* is the container of data items (e.g. `bytearray`). The length of the block is `len(data)`. Those blocks are usually not overlapping nor contiguous, and sorted by start address.

1.2 Python implementation

This library is the Cython complement to the Python implementation provided by the `bytesparse` Python package. Please refer to its own documentation for more details.

1.3 Cython implementation

The library provides an experimental *Cython* implementation. It tries to mimic the same algorithms of the Python implementation, while exploiting the speedup of compiled *C* code.

Beware that the Cython implementation is meant to be potentially faster than the pure Python one, but there might be even faster *ad-hoc* implementations of virtual memory highly optimized for the underlying hardware.

The addressing space is limited to that of an `uint_fast64_t` (typically 32-bit or 64-bit as per the hosting machine), so it is not possible to have an infinite addressing space, nor negative addresses. To keep the implementation code simple enough, the highest address (e.g. `0xFFFFFFFF` on a 32-bit machine) is reserved.

Block data chunks cannot be greater than the maximum `ssize_t` value (typically half of the addressing space).

The Cython implementation is optional, and potentially useful only when the Python implementation seems too slow for the user's algorithms, within the limits stated above.

If in doubt about using the Cython implementation, just stick with the Python one, which is much easier to integrate and debug.

More details can be found within `cbytesparse.c`.

1.4 Documentation

For the full documentation, please refer to:

https://cbytesparse.readthedocs.io/

1.5 Installation

From PyPI (might not be the latest version found on *github*):

```
$ pip install cbytesparse
```

From the source code root directory:

```
$ pip install .
```

1.6 Development

To run the all the tests:

```
$ pip install tox
$ tox
```

To regenerate the Cython files manually, run the following commands:

```
$ python scripts/cython_build_src.py
$ python scripts/cython_build_tests.py
```

or alternatively:

```
$ tox -e cythonize
```

CHAPTER

TWO

INSTALLATION

2.1 From *PyPI*

At the command line:

```
$ pip install cbytesparse
```

The package found on *PyPI* might be outdated with respect to the source repository.

2.2 From source

At the command line, at the root of the source directory:

```
$ pip install .
```

CHAPTER

THREE

REFERENCE

cbytesparse	Utilities for sparse blocks of bytes.
cbytesparse.c	Cython implementation.

3.1 cbytesparse

Utilities for sparse blocks of bytes.

Blocks are a useful way to describe sparse linear data.

The audience of this module are most importantly those who have to manage sparse blocks of bytes, where a very broad addressing space (*e.g.* 4 GiB) is used only in some sparse parts (*e.g.* physical memory addressing in a microcontroller).

This module also provides the `Memory` class, which is a handy wrapper around blocks, giving the user the flexibility of most operations of a `bytearray` on sparse byte-like chunks.

A *block* is a tuple `(start, data)` where *start* is the start address and *data* is the container of data items (e.g. `bytearray`). The length of the block is `len(data)`. Actually, the module uses lists instead of tuples, because the latter are mutables, thus can be changed in-place, without reallocation.

In this module it is common to require *spaces* blocks, *i.e.* blocks in which a block `b` does not start immediately after block `a`:

0	1	2	3	4	5	6	7	8
	[A	B	C]					
					[x	y	z]	

```
>>> a = [1, b'ABC']
>>> b = [5, b'xyz']
```

Instead, *overlapping* blocks have at least an addressed cell occupied by more items:

0	1	2	3	4	5	6	7	8
	[A	B	C]					
			[x	y	z]			
[#	#]							
		[!]						

```
>>> a = [1, b'ABC']
>>> b = [3, b'xyz']
>>> c = [0, b'##']
>>> d = [2, b'!']
```

Contiguous blocks are *non-overlapping*.

0	1	2	3	4	5	6	7	8
	[A	B	C]					
				[x	y	z]		

```
>>> a = [1, b'ABC']
>>> b = [4, b'xyz']
```

This module often deals with *sequences* of blocks, typically `list` objects containing blocks:

```
>>> seq = [[1, b'ABC'], [5, b'xyz']]
```

Sometimes *sequence generators* are allowed, in that blocks of the sequence are yielded on-the-fly by a generator, like *seq_gen*:

```
>>> seq_gen = ([i, (i + 0x21).to_bytes(1, 'little') * 3] for i in range(0, 15, 5))
>>> list(seq_gen)
[[0, b'!!!'], [5, b'&&&'], [10, b'+++']]
```

It is required that sequences are ordered, which means that a block `b` must follow a block `a` which end address is lesser than the *start* of `b`, like in:

```
>>> a = [1, b'ABC']
>>> b = [5, b'xyz']
>>> a[0] + len(a[1]) <= b[0]
True
```

cbytesparse.c	Cython implementation.

3.1.1 cbytesparse.c

Cython implementation.

Functions

collapse_blocks	Collapses a generic sequence of blocks.

cbytesparse.c.collapse_blocks

cbytesparse.c.**collapse_blocks**(*blocks*)

Collapses a generic sequence of blocks.

Given a generic sequence of blocks, writes them in the same order, generating a new sequence of non-contiguous blocks, sorted by address.

Parameters **blocks** (*sequence of blocks*) – Sequence of blocks to collapse.

Returns *list of blocks* – Collapsed block list.

Examples

0	1	2	3	4	5	6	7	8	9
[0	1	2	3	4	5	6	7	8	9]
[A	B	C	D]						
			[E	F]					
[$]									
						[x	y	z]	
[$	B	C	E	F	5	x	y	z	9]

```
>>> blocks = [
...     [0, b'0123456789'],
...     [0, b'ABCD'],
...     [3, b'EF'],
...     [0, b'$'],
...     [6, b'xyz'],
... ]
>>> collapse_blocks(blocks)
[[0, b'$BCEF5xyz9']]
```

~~~

0	1	2	3	4	5	6	7	8	9
[0	1	2]							
				[A	B]				
						[x	y	z]	
	[$]								
[0	$	2]		[A	B	x	y	z]	

```
>>> blocks = [
...     [0, b'012'],
...     [4, b'AB'],
...     [6, b'xyz'],
...     [1, b'$'],
... ]
>>> collapse_blocks(blocks)
[[0, b'0$2'], [4, b'ABxyz']]
```
~~~

Classes

BlockView	Block viewer.
Memory	Virtual memory.
bytesparse	Wrapper for more *bytearray* compatibility.

cbytesparse.c.BlockView

class cbytesparse.c.**BlockView**

Block viewer.

Memory view around an underlying block slice, implementing Python's *buffer* protocol API.

Methods

`__init__`

check	Checks for data consistency.
release	Forces object disposal.

Attributes

acquired	Underlying block currently acquired.
endex	Slice exclusive end address.
endin	Slice inclusive end address.
memview	Python `memoryview` wrapper.
start	Slice inclusive start address.

__bool__()

self != 0

__bytes__()

Converts into bytes.

Returns *bytes* – `bytes` clone of the viewed slice.

__eq__(*value*, /)

Return self==value.

__ge__(*value*, /)

Return self>=value.

__getattribute__(*name*, /)

Return getattr(self, name).

__getitem__(*key*, /)

Return self[key].

__gt__(*value*, /)

Return self>value.

__hash__ = None

__le__(*value*, /)
Return self<=value.

__len__()
int: Slice length.

__lt__(*value*, /)
Return self<value.

__ne__(*value*, /)
Return self!=value.

__new__(***kwargs*)

__reduce__()
BlockView.__reduce_cython__(self)

__repr__()
Return repr(self).

__setstate__()
BlockView.__setstate_cython__(self, __pyx_state)

__str__()
Return str(self).

acquired
Underlying block currently acquired.

Type bool

check()
Checks for data consistency.

endex
Slice exclusive end address.

Type int

endin
Slice inclusive end address.

Type int

memview
Python `memoryview` wrapper.

Type memoryview

release()
Forces object disposal.

Useful to make sure that any memory blocks are unreferenced before automatic garbage collection.

Any access to the object after calling this function could raise exceptions.

start
Slice inclusive start address.

Type int

cbytesparse.c.Memory

class cbytesparse.c.**Memory**(*start: Optional[Address] = None*, *endex: Optional[Address] = None*)

Virtual memory.

This class is a handy wrapper around *blocks*, so that it can behave mostly like a `bytearray`, but on sparse chunks of data.

Please look at examples of each method to get a glimpse of the features of this class.

On creation, at most one of *memory*, *blocks*, or *data* can be specified.

The Cython implementation limits the address range to that of the integral type `uint_fast64_t`.

Variables

- **~Memory._trim_start** (*int*) – Memory trimming start address. Any data before this address is automatically discarded; disabled if `None`.
- **~Memory._trim_endex** (*int*) – Memory trimming exclusive end address. Any data at or after this address is automatically discarded; disabled if `None`.

Parameters

- **start** (*int*) – Optional memory start address. Anything before will be trimmed away.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be trimmed away.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> memory = Memory()
>>> memory.to_blocks()
[]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.to_blocks()
[[5, b'Hello, World!']]
```

Methods

__init__

append	Appends a single item.
append_backup	Backups an *append()* operation.
append_restore	Restores an *append()* operation.
block_span	Span of block data.
blocks	Iterates over blocks.
bound	Bounds addresses.
clear	Clears an address range.
clear_backup	Backups a *clear()* operation.
clear_restore	Restores a *clear()* operation.

continues on next page

Table 7 – continued from previous page

content_items	Iterates over content address and value pairs.
content_keys	Iterates over content addresses.
content_values	Iterates over content values.
copy	Creates a shallow copy.
count	Counts items.
crop	Keeps data within an address range.
crop_backup	Backups a *crop()* operation.
crop_restore	Restores a *crop()* operation.
cut	Cuts a slice of memory.
delete	Deletes an address range.
delete_backup	Backups a *delete()* operation.
delete_restore	Restores a *delete()* operation.
equal_span	Span of homogeneous data.
extend	Concatenates items.
extend_backup	Backups an *extend()* operation.
extend_restore	Restores an *extend()* operation.
extract	Selects items from a range.
fill	Overwrites a range with a pattern.
fill_backup	Backups a *fill()* operation.
fill_restore	Restores a *fill()* operation.
find	Index of an item.
flood	Fills emptiness between non-touching blocks.
flood_backup	Backups a *flood()* operation.
flood_restore	Restores a *flood()* operation.
from_blocks	Creates a virtual memory from blocks.
from_bytes	Creates a virtual memory from a byte-like chunk.
from_items	Creates a virtual memory from a iterable address/byte mapping.
from_memory	Creates a virtual memory from another one.
from_values	Creates a virtual memory from a byte-like sequence.
fromhex	Creates a virtual memory from an hexadecimal string.
gaps	Iterates over block gaps.
get	Gets the item at an address.
hex	Converts into an hexadecimal string.
index	Index of an item.
insert	Inserts data.
insert_backup	Backups an *insert()* operation.
insert_restore	Restores an *insert()* operation.
intervals	Iterates over block intervals.
items	Iterates over address and value pairs.
keys	Iterates over addresses.
ofind	Index of an item.
peek	Gets the item at an address.
poke	Sets the item at an address.
poke_backup	Backups a *poke()* operation.
poke_restore	Restores a *poke()* operation.
pop	Takes a value away.
pop_backup	Backups a *pop()* operation.
pop_restore	Restores a *pop()* operation.

continues on next page

Table 7 – continued from previous page

popitem	Pops the last item.
popitem_backup	Backups a *popitem()* operation.
popitem_restore	Restores a *popitem()* operation.
remove	Removes an item.
remove_backup	Backups a *remove()* operation.
remove_restore	Restores a *remove()* operation.
reserve	Inserts emptiness.
reserve_backup	Backups a *reserve()* operation.
reserve_restore	Restores a *reserve()* operation.
reverse	Reverses the memory in-place.
rfind	Index of an item, reversed search.
rindex	Index of an item, reversed search.
rofind	Index of an item, reversed search.
rvalues	Iterates over values, reversed order.
setdefault	Defaults a value.
setdefault_backup	Backups a *setdefault()* operation.
setdefault_restore	Restores a *setdefault()* operation.
shift	Shifts the items.
shift_backup	Backups a *shift()* operation.
shift_restore	Restores an *shift()* operation.
to_blocks	Exports into blocks.
to_bytes	Exports into bytes.
update	Updates data.
update_backup	Backups an *update()* operation.
update_restore	Restores an *update()* operation.
validate	Validates internal structure.
values	Iterates over values.
view	Creates a view over a range.
write	Writes data.
write_backup	Backups a *write()* operation.
write_restore	Restores a *write()* operation.

Attributes

content_endex	Exclusive content end address.
content_endin	Inclusive content end address.
content_parts	Number of blocks.
content_size	Actual content size.
content_span	Memory content address span.
content_start	Inclusive content start address.
contiguous	Contains contiguous data.
endex	Exclusive end address.
endin	Inclusive end address.
span	Memory address span.
start	Inclusive start address.
trim_endex	Trimming exclusive end address.
trim_span	Trimming span addresses.
trim_start	Trimming start address.

__add__(*value*, */*)
Return self+value.

__bool__()
self != 0

__bytes__()
Creates a bytes clone.

Returns bytes – Cloned data.

Raises **ValueError** – Data not contiguous (see *contiguous*).

__contains__()
Checks if some items are contained.

Parameters **item** (*items*) – Items to find. Can be either some byte string or an integer.

Returns *bool* – Item is contained.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C]		[1	2	3]		[x	y	z]

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'123'], [9, b'xyz']])
>>> b'23' in memory
True
>>> ord('y') in memory
True
>>> b'$' in memory
False
```

__copy__()
Creates a shallow copy.

Note: The Cython implementation actually creates a deep copy.

Returns ImmutableMemory – Shallow copy.

__deepcopy__()
Creates a deep copy.

Returns ImmutableMemory – Deep copy.

__delitem__()
Deletes data.

Parameters **key** (*slice or int*) – Deletion range or address.

Note: This method is typically not optimized for a `slice` where its *step* is an integer greater than 1.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
	[A	B	C	y	z]						

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> del memory[4:9]
>>> memory.to_blocks()
[[1, b'ABCyz']]
```

~~~

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
	[A	B	C	D]		[$]		[x	z]		
	[A	B	D]		[$]		[x	z]			
	[A	D]			[x]						

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> del memory[9]
>>> memory.to_blocks()
[[1, b'ABCD'], [6, b'$'], [8, b'xz']]
>>> del memory[3]
>>> memory.to_blocks()
[[1, b'ABD'], [5, b'$'], [7, b'xz']]
>>> del memory[2:10:3]
>>> memory.to_blocks()
[[1, b'AD'], [5, b'x']]
```

__eq__()

Equality comparison.

Parameters **other** (Memory) – Data to compare with *self*.

If it is a `ImmutableMemory`, all of its blocks must match.

If it is a `bytes`, a `bytearray`, or a `memoryview`, it is expected to match the first and only stored block.

Otherwise, it must match the first and only stored block, via iteration over the stored values.

Returns *bool* – *self* is equal to *other*.
~~~

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> data = b'Hello, World!'
>>> memory = Memory.from_bytes(data)
>>> memory == data
True
>>> memory.shift(1)
>>> memory == data
True
```

```
>>> data = b'Hello, World!'
>>> memory = Memory.from_bytes(data)
>>> memory == list(data)
True
>>> memory.shift(1)
>>> memory == list(data)
True
```

__ge__(*value*, */*)
Return self>=value.

__getitem__()
Gets data.

Parameters **key** (*slice or int*) – Selection range or address. If it is a `slice` with bytes-like *step*, the latter is interpreted as the filling pattern.

Returns *items* – Items from the requested range.

Note: This method is typically not optimized for a `slice` where its *step* is an integer greater than 1.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B	C	D]		[$]		[x	y	z]
	65	66	67	68		36		120	121	122

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory[9]  # -> ord('y') = 121
121
>>> memory[:3].to_blocks()
[[1, b'AB']]
>>> memory[3:10].to_blocks()
[[3, b'CD'], [6, b'$'], [8, b'xy']]
>>> bytes(memory[3:10:b'.'])
b'CD.$.xy'
```

(continues on next page)

(continued from previous page)

```
>>> memory[memory.endex]
None
>>> bytes(memory[3:10:3])
b'C$y'
>>> memory[3:10:2].to_blocks()
[[3, b'C'], [6, b'y']]
>>> bytes(memory[3:10:2])
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

__gt__(*value*, /)
Return self>value.

__hash__ = None

__iadd__(*value*, /)
Return self+=value.

__imul__(*value*, /)
Return self*=value.

__init__(**args*, ***kwargs*)

__iter__()
Iterates over values.

Iterates over values between *start* and *endex*.

Yields *int* – Value as byte integer, or None.

__le__(*value*, /)
Return self<=value.

__len__()
Actual length.

Computes the actual length of the stored items, i.e. (*endex* - *start*). This will consider any trimmings being active.

Returns *int* – Memory length.

__lt__(*value*, /)
Return self<value.

__mul__(*value*, /)
Return self*value.

__ne__(*value*, /)
Return self!=value.

__new__(***kwargs*)

__radd__(*value*, /)
Return value+self.

__reduce__()
Memory.__reduce_cython__(self)

__repr__()
Return repr(self).

__reversed__()
Iterates over values, reversed order.

Iterates over values between *start* and *endex*, in reversed order.

Yields *int* – Value as byte integer, or None.

__rmul__(*value*, /)
Return value*self.

__setitem__()
Sets data.

Parameters

- **key** (*slice or int*) – Selection range or address.
- **value** (*items*) – Items to write at the selection address. If *value* is null, the range is cleared.

Examples

```
>>> from cbytesparse.c import Memory
```

4	5	6	7	8	9	10	11	12
	[A	B	C]		[x	y	z]	
	[A]					[y	z]	
	[A	B	C]		[x	y	z]	
	[A]		[C]			y	z]	
	[A	1	C]		[2	y	z]	

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory[7:10] = None
>>> memory.to_blocks()
[[5, b'AB'], [10, b'yz']]
>>> memory[7] = b'C'
>>> memory[9] = b'x'
>>> memory.to_blocks() == [[5, b'ABC'], [9, b'xyz']]
True
>>> memory[6:12:3] = None
>>> memory.to_blocks()
[[5, b'A'], [7, b'C'], [10, b'yz']]
>>> memory[6:13:3] = b'123'
>>> memory.to_blocks()
[[5, b'A1C'], [9, b'2yz3']]
```

~~~

0	1	2	3	4	5	6	7	8	9	10	11
					[A	B	C]		[x	y	z]
[$]		[A	B	C]		[x	y	z]			
[$]		[A	B	4	5	6	7	8	y	z]	
[$]		[A	B	4	5	<	>	8	y	z]	
~~~

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory[0:4] = b'$'
>>> memory.to_blocks()
[[0, b'$'], [2, b'ABC'], [6, b'xyz']]
>>> memory[4:7] = b'45678'
>>> memory.to_blocks()
[[0, b'$'], [2, b'AB45678yz']]
>>> memory[6:8] = b'<>'
>>> memory.to_blocks()
[[0, b'$'], [2, b'AB45<>8yz']]
```

__setstate__()
Memory.__setstate_cython__(self, __pyx_state)

__str__()
String representation.

If *content_size* is lesser than STR_MAX_CONTENT_SIZE, then the memory is represented as a list of blocks.

If exceeding, it is equivalent to *__repr__()*.

Returns *str* – String representation.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B	C]				[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [7, b'xyz']])
>>> str(memory)
<[[1, b'ABC'], [7, b'xyz']]>
```

_block_index_at(*address*)
Locates the block enclosing an address.

Returns the index of the block enclosing the given address.

Parameters **address** (*int*) – Address of the target item.

Returns *int* – Block index if found, None otherwise.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
	0	0	0	0		1		2	2	2	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> [memory._block_index_at(i) for i in range(12)]
[None, 0, 0, 0, 0, None, 1, None, 2, 2, 2, None]
```

_block_index_endex(*address*)

Locates the last block before an address range.

Returns the index of the last block whose end address is lesser than or equal to *address*.

Useful to find the termination block index in a ranged search.

Parameters **address** (*int*) – Exclusive end address of the scanned range.

Returns *int* – First block index before *address*.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
0	1	1	1	1	1	2	2	3	3	3	3

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> [memory._block_index_endex(i) for i in range(12)]
[0, 1, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3]
```

_block_index_start(*address*)

Locates the first block inside of an address range.

Returns the index of the first block whose start address is greater than or equal to *address*.

Useful to find the initial block index in a ranged search.

Parameters **address** (*int*) – Inclusive start address of the scanned range.

Returns *int* – First block index since *address*.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
0	0	0	0	0	1	1	2	2	2	2	3

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> [memory._block_index_start(i) for i in range(12)]
[0, 0, 0, 0, 0, 1, 1, 2, 2, 2, 2, 3]
```

_pretrim_endex(*start_min*, *size*)

Trims final data.

Low-level method to manage trimming of data starting from an address.

Parameters

- **start_min** (*int*) – Starting address of the erasure range. If None, *trim_endex* minus *size* is considered.
- **size** (*int*) – Size of the erasure range.

See also:

_pretrim_endex_backup()

_pretrim_endex_backup(*start_min*, *size*)
Backups a *_pretrim_endex()* operation.

Parameters

- **start_min** (*int*) – Starting address of the erasure range. If None, *trim_endex* minus *size* is considered.
- **size** (*int*) – Size of the erasure range.

Returns ImmutableMemory – Backup memory region.

See also:

_pretrim_endex()

_pretrim_start(*endex_max*, *size*)
Trims initial data.

Low-level method to manage trimming of data starting from an address.

Parameters

- **endex_max** (*int*) – Exclusive end address of the erasure range. If None, *trim_start* plus *size* is considered.
- **size** (*int*) – Size of the erasure range.

See also:

_pretrim_start_backup()

_pretrim_start_backup(*endex_max*, *size*)
Backups a *_pretrim_start()* operation.

Parameters

- **endex_max** (*int*) – Exclusive end address of the erasure range. If None, *trim_start* plus *size* is considered.
- **size** (*int*) – Size of the erasure range.

Returns ImmutableMemory – Backup memory region.

See also:

_pretrim_start()

append(*item*)
Appends a single item.

Parameters **item** (*int*) – Value to append. Can be a single byte string or integer.

See also:

append_backup() append_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> memory = Memory()
>>> memory.append(b'$')
>>> memory.to_blocks()
[[0, b'$']]
```

~~~

```
>>> memory = Memory()
>>> memory.append(3)
>>> memory.to_blocks()
[[0, b'\x03']]
```

append_backup()
Backups an *append()* operation.

Returns *None* – Nothing.

See also:

append() append_restore()

append_restore()
Restores an *append()* operation.

See also:

append() append_backup()

block_span(*address*)
Span of block data.

It searches for the biggest chunk of data adjacent to the given address.

If the address is within a gap, its bounds are returned, and its value is `None`.

If the address is before or after any data, bounds are `None`.

Parameters **address** (*int*) – Reference address.

Returns *tuple* – Start bound, exclusive end bound, and reference value.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> memory = Memory()
>>> memory.block_span(0)
(None, None, None)
```

~~~

0	1	2	3	4	5	6	7	8	9	10
[A	B	B	B	C]			[C	C	D]	
65	66	66	66	67			67	67	68	

```
>>> memory = Memory.from_blocks([[0, b'ABBBC'], [7, b'CCD']])
>>> memory.block_span(2)
(0, 5, 66)
>>> memory.block_span(4)
(0, 5, 67)
>>> memory.block_span(5)
(5, 7, None)
>>> memory.block_span(10)
(10, None, None)
```

blocks(*start*, *endex*)

Iterates over blocks.

Iterates over data blocks within an address range.

Parameters

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields *(start, memoryview)* – Start and data view of each block/slice.

See also:

intervals() to_blocks()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> [[s, bytes(d)] for s, d in memory.blocks()]
[[1, b'AB'], [5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.blocks(2, 9)]
[[2, b'B'], [5, b'x'], [7, b'12']]
>>> [[s, bytes(d)] for s, d in memory.blocks(3, 5)]
[]
```

bound(*start*, *endex*)

Bounds addresses.

It bounds the given addresses to stay within memory limits. None is used to ignore a limit for the *start* or *endex* directions.

In case of stored data, *content_start* and *content_endex* are used as bounds.

In case of trimming limits, *trim_start* or *trim_endex* are used as bounds, when not None.

In case *start* and *endex* are in the wrong order, one clamps the other if present (see the Python implementation for details).

Returns *tuple of int* – Bounded *start* and *endex*, closed interval.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().bound(None, None)
(0, 0)
>>> Memory().bound(None, 100)
(0, 100)
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.bound(0, 30)
(0, 30)
>>> memory.bound(2, 6)
(2, 6)
>>> memory.bound(None, 6)
(1, 6)
>>> memory.bound(2, None)
(2, 8)
```

~~~

0	1	2	3	4	5	6	7	8
	[[[		[A	B	C]			)))

```
>>> memory = Memory.from_blocks([[3, b'ABC']], start=1, endex=8)
>>> memory.bound(None, None)
(1, 8)
>>> memory.bound(0, 30)
(1, 8)
>>> memory.bound(2, 6)
(2, 6)
>>> memory.bound(2, None)
(2, 8)
>>> memory.bound(None, 6)
(1, 6)
```

clear(*start*, *endex*)

Clears an address range.

Parameters

- **start** (*int*) – Inclusive start address for clearing. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address for clearing. If `None`, *endex* is considered.

See also:

clear_backup() clear_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

4	5	6	7	8	9	10	11	12
	[A	B	C]		[x	y	z]	
	[A]					[y	z]	

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.clear(6, 10)
>>> memory.to_blocks()
[[5, b'A'], [10, b'yz']]
```

clear_backup(*start*, *endex*)

Backups a *clear()* operation.

Parameters

- **start** (*int*) – Inclusive start address for clearing. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for clearing. If None, *endex* is considered.

Returns ImmutableMemory – Backup memory region.

See also:

clear() *clear_restore()*

clear_restore(*backup*)

Restores a *clear()* operation.

Parameters **backup** (ImmutableMemory) – Backup memory region to restore.

See also:

clear() *clear_backup()*

content_endex

Exclusive content end address.

This property holds the exclusive end address of the memory content. By default, it is the current maximum exclusive end address of the last stored block.

If the memory has no data and no trimming, *start* is returned.

Trimming is considered only for an empty memory.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_endex
0
>>> Memory(endex=8).content_endex
0
>>> Memory(start=1, endex=8).content_endex
1
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_endex
8
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_endex
4
```

Type int

content_endin

Inclusive content end address.

This property holds the inclusive end address of the memory content. By default, it is the current maximum inclusive end address of the last stored block.

If the memory has no data and no trimming, *start* minus one is returned.

Trimming is considered only for an empty memory.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_endin
-1
>>> Memory(endex=8).content_endin
-1
>>> Memory(start=1, endex=8).content_endin
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_endin
7
```
~~~

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_endin
3
```

Type int

content_items(*start*, *endex*)

Iterates over content address and value pairs.

Parameters

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields *int* – Content address and value pairs.

See also:

meth:*content_keys* meth:*content_values*

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> dict(memory.content_items())
{1: 65, 2: 66, 5: 120, 7: 49, 8: 50, 9: 51}
>>> dict(memory.content_items(2, 9))
{2: 66, 5: 120, 7: 49, 8: 50}
>>> dict(memory.content_items(3, 5))
{}
```

content_keys(*start*, *endex*)

Iterates over content addresses.

Parameters

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields *int* – Content addresses.

See also:

meth:*content_items* meth:*content_values*
~~~

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> list(memory.content_keys())
[1, 2, 5, 7, 8, 9]
>>> list(memory.content_keys(2, 9))
[2, 5, 7, 8]
>>> list(memory.content_keys(3, 5))
[]
```

content_parts

Number of blocks.

Returns *int* – The number of blocks.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_parts
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_parts
2
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_parts
1
```

content_size

Actual content size.

Returns *int* – The sum of all block lengths.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_size
0
>>> Memory(start=1, endex=8).content_size
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_size
6
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_size
3
```

content_span

Memory content address span.

A `tuple` holding both *content_start* and *content_endex*.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_span
(0, 0)
>>> Memory(start=1).content_span
(1, 1)
>>> Memory(endex=8).content_span
(0, 0)
>>> Memory(start=1, endex=8).content_span
(1, 1)
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	
~~~

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_span
(1, 8)
```

Type tuple of int

content_start
Inclusive content start address.

This property holds the inclusive start address of the memory content. By default, it is the current minimum inclusive start address of the first stored block.

If the memory has no data and no trimming, 0 is returned.

Trimming is considered only for an empty memory.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_start
0
>>> Memory(start=1).content_start
1
>>> Memory(start=1, endex=8).content_start
1
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_start
1
```

~~~

0	1	2	3	4	5	6	7	8
	[[[				[x	y	z]	

```
>>> memory = Memory.from_blocks([[5, b'xyz']], start=1)
>>> memory.content_start
5
```

Type int

content_values(*start*, *endex*)
Iterates over content values.

Parameters

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields *int* – Content values.

See also:

meth:*content_items* meth:*content_keys*

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> list(memory.content_values())
[65, 66, 120, 49, 50, 51]
>>> list(memory.content_values(2, 9))
[66, 120, 49, 50]
>>> list(memory.content_values(3, 5))
[]
```

contiguous
Contains contiguous data.

The memory is considered to have contiguous data if there is no empty space between blocks.

If trimming is defined, there must be no empty space also towards it.

Type bool

copy()
Creates a shallow copy.

Note: The Cython implementation actually creates a deep copy.

Returns ImmutableMemory – Shallow copy.

count(*item*, *start*, *endex*)
Counts items.

Parameters

- **item** (*items*) – Reference value to count.
- **start** (*int*) – Inclusive start of the searched range. If None, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If None, *endex* is considered.

Returns *int* – The number of items equal to *value*.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C]		[B	a	t]		[t	a	b]

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'Bat'], [9, b'tab']])
>>> memory.count(b'a')
2
```

crop(*start*, *endex*)
Keeps data within an address range.

Parameters

- **start** (*int*) – Inclusive start address for cropping. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for cropping. If None, *endex* is considered.

See also:

crop_backup() crop_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

4	5	6	7	8	9	10	11	12
	[A	B	C]		[x	y	z]	
		[B	C]		[x]			

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.crop(6, 10)
>>> memory.to_blocks()
[[6, b'BC'], [9, b'x']]
```

crop_backup(*start*, *endex*)
Backups a *crop()* operation.

Parameters

- **start** (*int*) – Inclusive start address for cropping. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for cropping. If None, *endex* is considered.

Returns ImmutableMemory pair – Backup memory regions.

See also:

crop() crop_restore()

crop_restore(*backup_start*, *backup_endex*)
Restores a *crop()* operation.

Parameters

- **backup_start** (ImmutableMemory) – Backup memory region to restore at the beginning.
- **backup_endex** (ImmutableMemory) – Backup memory region to restore at the end.

See also:

crop() crop_backup()

cut(*start*, *endex*, *bound*)
Cuts a slice of memory.

Parameters

- **start** (*int*) – Inclusive start address for cutting. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for cutting. If None, *endex* is considered.
- **bound** (*bool*) – The selected address range is applied to the resulting memory as its trimming range. This retains information about any initial and final emptiness of that range, which would be lost otherwise.

Returns *Memory* – A copy of the memory from the selected range.

delete(*start*, *endex*)
Deletes an address range.

Parameters

- **start** (*int*) – Inclusive start address for deletion. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for deletion. If None, *endex* is considered.

See also:

delete_backup() delete_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

4	5	6	7	8	9	10	11	12	13
	[A	B	C]			[x	y	z]	
	[A	y	z]						

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.delete(6, 10)
>>> memory.to_blocks()
[[5, b'Ayz']]
```

delete_backup(*start*, *endex*)
Backups a *delete()* operation.

Parameters

- **start** (*int*) – Inclusive start address for deletion. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for deletion. If None, *endex* is considered.

Returns ImmutableMemory – Backup memory region.

See also:

delete() *delete_restore()*

delete_restore(*backup*)
Restores a *delete()* operation.

Parameters **backup** (ImmutableMemory) – Backup memory region

See also:

delete() *delete_backup()*

endex
Exclusive end address.

This property holds the exclusive end address of the virtual space. By default, it is the current maximmum exclusive end address of the last stored block.

If *trim_endex* not None, that is returned.

If the memory has no data and no trimming, *start* is returned.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().endex
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.endex
8
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.endex
8
```

Type int

endin
Inclusive end address.

This property holds the inclusive end address of the virtual space. By default, it is the current maximmum inclusive end address of the last stored block.

If *trim_endex* not None, that minus one is returned.

If the memory has no data and no trimming, *start* is returned.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().endin
-1
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.endin
7
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.endin
7
```

Type int

equal_span(*address*)

Span of homogeneous data.

It searches for the biggest chunk of data adjacent to the given address, with the same value at that address.

If the address is within a gap, its bounds are returned, and its value is None.

If the address is before or after any data, bounds are None.

Parameters **address** (*int*) – Reference address.

Returns *tuple* – Start bound, exclusive end bound, and reference value.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> memory = Memory()
>>> memory.equal_span(0)
(None, None, None)
```

~~~

0	1	2	3	4	5	6	7	8	9	10
[A	B	B	B	C]			[C	C	D]	
65	66	66	66	67			67	67	68	

```
>>> memory = Memory.from_blocks([[0, b'ABBBC'], [7, b'CCD']])
>>> memory.equal_span(2)
(1, 4, 66)
>>> memory.equal_span(4)
(4, 5, 67)
>>> memory.equal_span(5)
(5, 7, None)
>>> memory.equal_span(10)
(10, None, None)
```

extend(*items*, *offset*)

Concatenates items.

Equivalent to `self += items`.

Parameters

- **items** (*iterable*) – Items to append at the end of the current virtual space.
- **offset** (*int*) – Optional offset w.r.t. *content_endex*.

See also:

extend_backup() extend_restore()

extend_backup(*offset*)

Backups an *extend()* operation.

Parameters **offset** (*int*) – Optional offset w.r.t. *content_endex*.

Returns *int* – Content exclusive end address.

See also:

extend() extend_restore()

extend_restore(*content_endex*)

Restores an *extend()* operation.

Parameters **content_endex** (*int*) – Content exclusive end address to restore.

See also:

extend() extend_backup()

extract(*start*, *endex*, *pattern*, *step*, *bound*)

Selects items from a range.
~~~

Parameters

- **start** (*int*) – Inclusive start of the extracted range. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end of the extracted range. If `None`, *endex* is considered.
- **pattern** (*items*) – Optional pattern of items to fill the emptiness.
- **step** (*int*) – Optional address stepping between bytes extracted from the range. It has the same meaning of Python's `slice.step`, but negative steps are ignored. Please note that a *step* greater than 1 could take much more time to process than the default unitary step.
- **bound** (*bool*) – The selected address range is applied to the resulting memory as its trimming range. This retains information about any initial and final emptiness of that range, which would be lost otherwise.

Returns `ImmutableMemory` – A copy of the memory from the selected range.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.extract().to_blocks()
[[1, b'ABCD'], [6, b'$'], [8, b'xyz']]
>>> memory.extract(2, 9).to_blocks()
[[2, b'BCD'], [6, b'$'], [8, b'x']]
>>> memory.extract(start=2).to_blocks()
[[2, b'BCD'], [6, b'$'], [8, b'xyz']]
>>> memory.extract(endex=9).to_blocks()
[[1, b'ABCD'], [6, b'$'], [8, b'x']]
>>> memory.extract(5, 8).span
(5, 8)
>>> memory.extract(pattern=b'.').to_blocks()
[[1, b'ABCD.$.xyz']]
>>> memory.extract(pattern=b'.', step=3).to_blocks()
[[1, b'AD.z']]
```

fill(*start*, *endex*, *pattern*)

Overwrites a range with a pattern.

Parameters

- **start** (*int*) – Inclusive start address for filling. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address for filling. If `None`, *endex* is considered.
- **pattern** (*items*) – Pattern of items to fill the range.

See also:

fill_backup() *fill_restore()*

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	[1	2	3	1	2	3	1	2]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.fill(pattern=b'123')
>>> memory.to_blocks()
[[1, b'12312312']]
```

~~~

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	[A	B	1	2	3	1	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.fill(3, 7, b'123')
>>> memory.to_blocks()
[[1, b'AB1231yz']]
```

fill_backup(*start*, *endex*)
Backups a *fill()* operation.

Parameters

- **start** (*int*) – Inclusive start address for filling. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for filling. If None, *endex* is considered.

Returns ImmutableMemory – Backup memory region.

See also:

fill() fill_restore()

fill_restore(*backup*)
Restores a *fill()* operation.

Parameters **backup** (ImmutableMemory) – Backup memory region to restore.

See also:

fill() fill_backup()

find(*item*, *start*, *endex*)
Index of an item.

Parameters

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
- **start** (*int*) – Inclusive start of the searched range. If None, *endex* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If None, *endex* is considered.
~~~

Returns *int* – The index of the first item equal to *value*, or -1.

flood(*start*, *endex*, *pattern*)

Fills emptiness between non-touching blocks.

Parameters

- **start** (*int*) – Inclusive start address for flooding. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for flooding. If None, *endex* is considered.
- **pattern** (*items*) – Pattern of items to fill the range.

See also:

flood_backup() flood_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	[A	B	C	1	2	x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.flood(pattern=b'123')
>>> memory.to_blocks()
[[1, b'ABC12xyz']]
```

~~~

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	[A	B	C	2	3	x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.flood(3, 7, b'123')
>>> memory.to_blocks()
[[1, b'ABC23xyz']]
```

flood_backup(*start*, *endex*)

Backups a *flood()* operation.

Parameters

- **start** (*int*) – Inclusive start address for filling. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for filling. If None, *endex* is considered.

Returns *list of open intervals* – Backup memory gaps.

See also:

flood() flood_restore()

flood_restore(*gaps*)

Restores a *flood()* operation.
~~~

Parameters **gaps** (*list of open intervals*) – Backup memory gaps to restore.

See also:

flood() flood_backup()

classmethod from_blocks(*blocks*, *offset*, *start*, *endex*, *copy*, *validate*)

Creates a virtual memory from blocks.

Parameters

- **blocks** (*list of blocks*) – A sequence of non-overlapping blocks, sorted by address.
- **offset** (*int*) – Some address offset applied to all the blocks.
- **start** (*int*) – Optional memory start address. Anything before will be trimmed away.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be trimmed away.
- **copy** (*bool*) – Forces copy of provided input data.
- **validate** (*bool*) – Validates the resulting `ImmutableMemory` object.

Returns `ImmutableMemory` – The resulting memory object.

Raises **ValueError** – Some requirements are not satisfied.

See also:

to_blocks()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8
	[A	B	C]					
					[x	y	z]	

```
>>> blocks = [[1, b'ABC'], [5, b'xyz']]
>>> memory = Memory.from_blocks(blocks)
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz']]
>>> memory = Memory.from_blocks(blocks, offset=3)
>>> memory.to_blocks()
[[4, b'ABC'], [8, b'xyz']]
```

~~~

```
>>> # Loads data from an Intel HEX record file
>>> # NOTE: Record files typically require collapsing!
>>> import hexrec.records as hr
>>> blocks = hr.load_blocks('records.hex')
>>> memory = Memory.from_blocks(collapse_blocks(blocks))
>>> memory
    ...
```
~~~

classmethod from_bytes(*data*, *offset*, *start*, *endex*, *copy*, *validate*)
Creates a virtual memory from a byte-like chunk.

Parameters

- **data** (*byte-like data*) – A byte-like chunk of data (e.g. `bytes`, `bytearray`, `memoryview`).
- **offset** (*int*) – Start address of the block of data.
- **start** (*int*) – Optional memory start address. Anything before will be trimmed away.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be trimmed away.
- **copy** (*bool*) – Forces copy of provided input data into the underlying data structure.
- **validate** (*bool*) – Validates the resulting `ImmutableMemory` object.

Returns `ImmutableMemory` – The resulting memory object.

Raises **ValueError** – Some requirements are not satisfied.

See also:

to_bytes()

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> memory = Memory.from_bytes(b'')
>>> memory.to_blocks()
[]
```

~~~

0	1	2	3	4	5	6	7	8
		[A	B	C	x	y	z]	

```
>>> memory = Memory.from_bytes(b'ABCxyz', 2)
>>> memory.to_blocks()
[[2, b'ABCxyz']]
```

classmethod from_items(*items*, *offset*, *start*, *endex*, *validate*)
Creates a virtual memory from a iterable address/byte mapping.

Parameters

- **items** (*iterable address/byte mapping*) – An iterable mapping of address to byte values. Values of `None` are translated as gaps. When an address is stated multiple times, the last is kept.
- **offset** (*int*) – An address offset applied to all the values.
- **start** (*int*) – Optional memory start address. Anything before will be trimmed away.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be trimmed away.
~~~

- **validate** (*bool*) – Validates the resulting `ImmutableMemory` object.

Returns `ImmutableMemory` – The resulting memory object.

Raises **ValueError** – Some requirements are not satisfied.

See also:

to_bytes()

Examples

```
>>> from bytesparse.inplace import Memory
```

```
>>> memory = Memory.from_values({})
>>> memory.to_blocks()
[]
```

~~~

0	1	2	3	4	5	6	7	8
		[A	Z]		[x]			

```
>>> items = [
...     (0, ord('A')),
...     (1, ord('B')),
...     (3, ord('x')),
...     (1, ord('Z')),
... ]
>>> memory = Memory.from_items(items, offset=2)
>>> memory.to_blocks()
[[2, b'AZ'], [5, b'x']]
```

classmethod **from_memory**(*memory*, *offset*, *start*, *endex*, *copy*, *validate*)

Creates a virtual memory from another one.

Parameters

- **memory** (*Memory*) – A `ImmutableMemory` to copy data from.
- **offset** (*int*) – Some address offset applied to all the blocks.
- **start** (*int*) – Optional memory start address. Anything before will be trimmed away.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be trimmed away.
- **copy** (*bool*) – Forces copy of provided input data into the underlying data structure.
- **validate** (*bool*) – Validates the resulting `MemorImmutableMemory` object.

Returns `ImmutableMemory` – The resulting memory object.

Raises **ValueError** – Some requirements are not satisfied.
~~~

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC', 5)
>>> memory2 = Memory.from_memory(memory1)
>>> memory2.to_blocks()
[[5, b'ABC']]
>>> memory1 == memory2
True
>>> memory1 is memory2
False
>>> memory1.to_blocks() is memory2.to_blocks()
False
```

~~~

```
>>> memory1 = Memory.from_bytes(b'ABC', 10)
>>> memory2 = Memory.from_memory(memory1, -3)
>>> memory2.to_blocks()
[[7, b'ABC']]
>>> memory1 == memory2
False
```

~~~

```
>>> memory1 = Memory.from_bytes(b'ABC', 10)
>>> memory2 = Memory.from_memory(memory1, copy=False)
>>> all((b1[1] is b2[1])  # compare block data
...     for b1, b2 in zip(memory1.to_blocks(), memory2.to_blocks()))
True
```

classmethod from_values(*values*, *offset*, *start*, *endex*, *validate*)

Creates a virtual memory from a byte-like sequence.

Parameters

- **values** (*iterable byte-like sequence*) – An iterable sequence of byte values. Values of `None` are translated as gaps.
- **offset** (*int*) – An address offset applied to all the values.
- **start** (*int*) – Optional memory start address. Anything before will be trimmed away.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be trimmed away.
- **validate** (*bool*) – Validates the resulting `ImmutableMemory` object.

Returns `ImmutableMemory` – The resulting memory object.

Raises `ValueError` – Some requirements are not satisfied.

See also:

to_bytes()

Examples

```
>>> from bytesparse.inplace import Memory
```

```
>>> memory = Memory.from_values(range(0))
>>> memory.to_blocks()
[]
```

~~~

0	1	2	3	4	5	6	7	8
		[A	B	C	D	E]		

```
>>> memory = Memory.from_values(range(ord('A'), ord('F')), offset=2)
>>> memory.to_blocks()
[[2, b'ABCDE']]
```

classmethod **fromhex**(*string*)

Creates a virtual memory from an hexadecimal string.

Parameters **string** (*str*) – Hexadecimal string.

Returns ImmutableMemory – The resulting memory object.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> memory = Memory.fromhex('')
>>> bytes(memory)
b''
```

~~~

```
>>> memory = Memory.fromhex('48656C6C6F2C20576F726C6421')
>>> bytes(memory)
b'Hello, World!'
```

gaps(*start*, *endex*)

Iterates over block gaps.

Iterates over gaps emptiness bounds within an address range. If a yielded bound is None, that direction is infinitely empty (valid before or after global data bounds).

Parameters

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields *pair of addresses* – Block data interval boundaries.

See also:

intervals()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> list(memory.gaps())
[(None, 1), (3, 5), (6, 7), (10, None)]
>>> list(memory.gaps(0, 11))
[(0, 1), (3, 5), (6, 7), (10, 11)]
>>> list(memory.gaps(*memory.span))
[(3, 5), (6, 7)]
>>> list(memory.gaps(2, 6))
[(3, 5)]
```

get(*address*, *default*)

Gets the item at an address.

Returns *int* – The item at *address*, *default* if empty.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.get(3)  # -> ord('C') = 67
67
>>> memory.get(6)  # -> ord('$') = 36
36
>>> memory.get(10)  # -> ord('z') = 122
122
>>> memory.get(0)  # -> empty -> default = None
None
>>> memory.get(7)  # -> empty -> default = None
None
>>> memory.get(11)  # -> empty -> default = None
None
>>> memory.get(0, 123)  # -> empty -> default = 123
123
>>> memory.get(7, 123)  # -> empty -> default = 123
123
>>> memory.get(11, 123)  # -> empty -> default = 123
123
```

hex(**args*)

Converts into an hexadecimal string.

Parameters

- **sep** (*str*) – Separator string between bytes. Defaults to an emoty string if not provided. Available since Python 3.8.
- **bytes_per_sep** (*int*) – Number of bytes grouped between separators. Defaults to one byte per group. Available since Python 3.8.

Returns *str* – Hexadecimal string representation.

Raises **ValueError** – Data not contiguous (see *contiguous*).

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().hex() == ''
True
```

~~~

```
>>> memory = Memory.from_bytes(b'Hello, World!')
>>> memory.hex()
48656c6c6f2c20576f726c6421
>>> memory.hex('.')
48.65.6c.6c.6f.2c.20.57.6f.72.6c.64.21
>>> memory.hex('.', 4)
48.656c6c6f.2c20576f.726c6421
```

index(*item*, *start*, *endex*)

Index of an item.

Parameters

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
- **start** (*int*) – Inclusive start of the searched range. If None, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If None, *endex* is considered.

Returns *int* – The index of the first item equal to *value*.

Raises **ValueError** – Item not found.

insert(*address*, *data*)

Inserts data.

Inserts data, moving existing items after the insertion address by the size of the inserted data.

Arguments::

address (int): Address of the insertion point.

data (bytes): Data to insert.

See also:

insert_backup() insert_restore()
~~~

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C]			[x	y	z]			
	[A	B	C]			[x	y	z]		[$]	
	[A	B	C]			[x	y	1	z]		[$]

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.insert(10, b'$')
>>> memory.to_blocks()
[[1, b'ABC'], [6, b'xyz'], [10, b'$']]
>>> memory.insert(8, b'1')
>>> memory.to_blocks()
[[1, b'ABC'], [6, b'xy1z'], [11, b'$']]
```

insert_backup(*address*, *data*)
Backups an *insert()* operation.

Parameters

- **address** (*int*) – Address of the insertion point.
- **data** (*bytes*) – Data to insert.

Returns (int, ImmutableMemory) – Insertion address, backup memory region.

See also:

insert() insert_restore()

insert_restore(*address*, *backup*)
Restores an *insert()* operation.

Parameters

- **address** (*int*) – Address of the insertion point.
- **backup** (*Memory*) – Backup memory region to restore.

See also:

insert() insert_backup()

intervals(*start*, *endex*)
Iterates over block intervals.

Iterates over data boundaries within an address range.

Parameters

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields *pair of addresses* – Block data interval boundaries.

See also:

blocks() gaps()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> list(memory.intervals())
[(1, 3), (5, 6), (7, 10)]
>>> list(memory.intervals(2, 9))
[(2, 3), (5, 6), (7, 9)]
>>> list(memory.intervals(3, 5))
[]
```

items(*start*, *endex*, *pattern*)

Iterates over address and value pairs.

Iterates over address and value pairs, from *start* to *endex*. Implemets the interface of `dict`.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered. If `Ellipsis`, the iterator is infinite.
- **pattern** (*items*) – Pattern of values to fill emptiness.

Yields *int* – Range address and value pairs.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> from itertools import islice
>>> memory = Memory()
>>> list(memory.items(endex=8))
[(0, None), (1, None), (2, None), (3, None), (4, None), (5, None), (6, None),␣
↪(7, None)]
>>> list(memory.items(3, 8))
[(3, None), (4, None), (5, None), (6, None), (7, None)]
>>> list(islice(memory.items(3, ...), 7))
[(3, None), (4, None), (5, None), (6, None), (7, None), (8, None), (9, None)]
```

~~~

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	65	66	67			120	121	122	
~~~

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> list(memory.items())
[(1, 65), (2, 66), (3, 67), (4, None), (5, None), (6, 120), (7, 121), (8, 122)]
>>> list(memory.items(3, 8))
[(3, 67), (4, None), (5, None), (6, 120), (7, 121)]
>>> list(islice(memory.items(3, ...), 7))
[(3, 67), (4, None), (5, None), (6, 120), (7, 121), (8, 122), (9, None)]
```

keys(*start*, *endex*)

Iterates over addresses.

Iterates over addresses, from *start* to *endex*. Implemets the interface of `dict`.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered. If `Ellipsis`, the iterator is infinite.

Yields *int* – Range address.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> from itertools import islice
>>> memory = Memory()
>>> list(memory.keys())
[]
>>> list(memory.keys(endex=8))
[0, 1, 2, 3, 4, 5, 6, 7]
>>> list(memory.keys(3, 8))
[3, 4, 5, 6, 7]
>>> list(islice(memory.keys(3, ...), 7))
[3, 4, 5, 6, 7, 8, 9]
```

~~~

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> list(memory.keys())
[1, 2, 3, 4, 5, 6, 7, 8]
>>> list(memory.keys(endex=8))
[1, 2, 3, 4, 5, 6, 7]
>>> list(memory.keys(3, 8))
[3, 4, 5, 6, 7]
>>> list(islice(memory.keys(3, ...), 7))
[3, 4, 5, 6, 7, 8, 9]
```

ofind(*item*, *start*, *endex*)

Index of an item.
~~~

Parameters

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
- **start** (*int*) – Inclusive start of the searched range. If None, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If None, *endex* is considered.

Returns *int* – The index of the first item equal to *value*, or None.

peek(*address*)

Gets the item at an address.

Returns *int* – The item at *address*, None if empty.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.peek(3)  # -> ord('C') = 67
67
>>> memory.peek(6)  # -> ord('$') = 36
36
>>> memory.peek(10)  # -> ord('z') = 122
122
>>> memory.peek(0)
None
>>> memory.peek(7)
None
>>> memory.peek(11)
None
```

poke(*address*, *item*)

Sets the item at an address.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to set, None to clear the cell.

See also:

poke_backup() poke_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.poke(3, b'@')
>>> memory.peek(3)  # -> ord('@') = 64
64
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.poke(5, b'@')
>>> memory.peek(5)  # -> ord('@') = 64
64
```

poke_backup(*address*)
Backups a *poke()* operation.

Parameters **address** (*int*) – Address of the target item.

Returns *(int, int)* – *address*, item at *address* (None if empty).

See also:

poke() poke_restore()

poke_restore(*address*, *item*)
Restores a *poke()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to restore.

See also:

poke() poke_backup()

pop(*address*, *default*)
Takes a value away.

Parameters

- **address** (*int*) – Address of the byte to pop. If None, the very last byte is popped.
- **default** (*int*) – Value to return if *address* is within emptiness.

Returns *int* – Value at *address*; *default* within emptiness.

See also:

pop_backup() pop_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
	[A	B	C	D]		[$]		[x	y]		
	[A	B	D]		[$]		[x	y]			

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.pop()  # -> ord('z') = 122
122
>>> memory.pop(3)  # -> ord('C') = 67
67
>>> memory.pop(6, 63)  # -> ord('?') = 67
63
```

pop_backup(*address*)

Backups a *pop()* operation.

Parameters **address** (*int*) – Address of the byte to pop. If None, the very last byte is popped.

Returns *(int, int)* – *address*, item at *address* (None if empty).

See also:

pop() pop_restore()

pop_restore(*address*, *item*)

Restores a *pop()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to restore, None if empty.

See also:

pop() pop_backup()

popitem()

Pops the last item.

Returns *(int, int)* – Address and value of the last item.

See also:

popitem_backup() popitem_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A]								[y	z]	

```
>>> memory = Memory.from_blocks([[1, b'A'], [9, b'yz']])
>>> memory.popitem()  # -> ord('z') = 122
(10, 122)
>>> memory.popitem()  # -> ord('y') = 121
(9, 121)
>>> memory.popitem()  # -> ord('A') = 65
(1, 65)
>>> memory.popitem()
Traceback (most recent call last):
    ...
KeyError: empty
```

popitem_backup()

Backups a *popitem()* operation.

Returns *(int, int)* – Address and value of the last item.

See also:

popitem() popitem_restore()

popitem_restore(*address*, *item*)

Restores a *popitem()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int*) – Item to restore.

See also:

popitem() popitem_backup()

remove(*item*, *start*, *endex*)

Removes an item.

Searches and deletes the first occurrence of an item.

Parameters

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
- **start** (*int*) – Inclusive start of the searched range. If None, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If None, *endex* is considered.

Raises **ValueError** – Item not found.

See also:

remove_backup() remove_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
	[A	D]		[$]		[x	y	z]			
	[A	D]			[x	y	z]				

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.remove(b'BC')
>>> memory.to_blocks()
[[1, b'AD'], [4, b'$'], [6, b'xyz']]
>>> memory.remove(ord('$'))
>>> memory.to_blocks()
[[1, b'AD'], [5, b'xyz']]
>>> memory.remove(b'?')
Traceback (most recent call last):
    ...
ValueError: subsection not found
```

remove_backup(*item*, *start*, *endex*)
Backups a *remove()* operation.

Parameters

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
- **start** (*int*) – Inclusive start of the searched range. If None, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If None, *endex* is considered.

Returns *Memory* – Backup memory region.

See also:

remove() remove_restore()

remove_restore(*backup*)
Restores a *remove()* operation.

Parameters **backup** (*Memory*) – Backup memory region.

See also:

remove() remove_backup()

reserve(*address*, *size*)
Inserts emptiness.

Reserves emptiness at the provided address.

Parameters

- **address** (*int*) – Start address of the emptiness to insert.
- **size** (*int*) – Size of the emptiness to insert.

See also:

reserve_backup() reserve_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

2	3	4	5	6	7	8	9	10	11	12
	[A	B	C]		[x	y	z]			
	[A]			B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[3, b'ABC'], [7, b'xyz']])
>>> memory.reserve(4, 2)
>>> memory.to_blocks()
[[3, b'A'], [6, b'BC'], [9, b'xyz']]
```

~~~

2	3	4	5	6	7	8	9	10	11	12
			[A	B	C]		[x	y	z]	)))
								[A	B]	)))

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']], endex=12)
>>> memory.reserve(5, 5)
>>> memory.to_blocks()
[[10, b'AB']]
```

reserve_backup(*address*, *size*)
Backups a *reserve()* operation.

Parameters

- **address** (*int*) – Start address of the emptiness to insert.
- **size** (*int*) – Size of the emptiness to insert.

Returns (int, ImmutableMemory) – Reservation address, backup memory region.

See also:

reserve() reserve_restore()

reserve_restore(*address*, *backup*)
Restores a *reserve()* operation.

Parameters

- **address** (*int*) – Address of the reservation point.
- **backup** (ImmutableMemory) – Backup memory region to restore.

See also:

reserve() reserve_backup()

reverse()
Reverses the memory in-place.

Data is reversed within the memory *span*.
~~~

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
	[z	y	x]		[$]		[D	C	B	A]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.reverse()
>>> memory.to_blocks()
[[1, b'zyx'], [5, b'$'], [7, b'DCBA']]
```

~~~

0	1	2	3	4	5	6	7	8	9	10	11
		[[[		[A	B	C]				)))	
		[[[			[C	B	A]			)))	

```
>>> memory = Memory.from_bytes(b'ABCD', 3, start=2, endex=10)
>>> memory.reverse()
>>> memory.to_blocks()
[[5, b'CBA']]
```

rfind(*item*, *start*, *endex*)
Index of an item, reversed search.

Parameters

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
- **start** (*int*) – Inclusive start of the searched range. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If `None`, *endex* is considered.

Returns *int* – The index of the last item equal to *value*, or -1.

rindex(*item*, *start*, *endex*)
Index of an item, reversed search.

Parameters

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
- **start** (*int*) – Inclusive start of the searched range. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If `None`, *endex* is considered.

Returns *int* – The index of the last item equal to *value*.

Raises **ValueError** – Item not found.

rofind(*item*, *start*, *endex*)
Index of an item, reversed search.

Parameters

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
~~~

- **start** (*int*) – Inclusive start of the searched range. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If `None`, *endex* is considered.

Returns *int* – The index of the last item equal to *value*, or `None`.

rvalues(*start*, *endex*, *pattern*)

Iterates over values, reversed order.

Iterates over values, from *endex* to *start*.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered. If `Ellipsis`, the iterator is infinite.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.
- **pattern** (*items*) – Pattern of values to fill emptiness.

Yields *int* – Range values.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9
			A	B	C	D	A		
			65	66	67	68	65		

```
>>> from itertools import islice
>>> memory = Memory()
>>> list(memory.rvalues(endex=8))
[None, None, None, None, None, None, None, None]
>>> list(memory.rvalues(3, 8))
[None, None, None, None, None]
>>> list(islice(memory.rvalues(..., 8), 7))
[None, None, None, None, None, None, None]
>>> list(memory.rvalues(3, 8, b'ABCD'))
[65, 68, 67, 66, 65]
```

~~~

0	1	2	3	4	5	6	7	8	9
	[A	B	C]	<1	2>	[x	y	z]	
	65	66	67			120	121	122	
	65	66	67	49	50	120	121	122	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> list(memory.rvalues())
[122, 121, 120, None, None, 67, 66, 65]
>>> list(memory.rvalues(3, 8))
[121, 120, None, None, 67]
>>> list(islice(memory.rvalues(..., 8), 7))
```

(continues on next page)
~~~

(continued from previous page)

```
[121, 120, None, None, 67, 66, 65]
>>> list(memory.rvalues(3, 8, b'0123'))
[121, 120, 50, 49, 67]
```

setdefault(*address*, *default*)

Defaults a value.

Parameters

- **address** (*int*) – Address of the byte to set.
- **default** (*int*) – Value to set if *address* is within emptiness.

Returns *int* – Value at *address*; *default* within emptiness.

See also:

setdefault_backup() setdefault_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.setdefault(3, b'@')  # -> ord('C') = 67
67
>>> memory.peek(3)  # -> ord('C') = 67
67
>>> memory.setdefault(5, 64)  # -> ord('@') = 64
64
>>> memory.peek(5)  # -> ord('@') = 64
64
>>> memory.setdefault(9) is None
False
>>> memory.peek(9) is None
False
>>> memory.setdefault(7) is None
True
>>> memory.peek(7) is None
True
```

setdefault_backup(*address*)

Backups a *setdefault()* operation.

Parameters **address** (*int*) – Address of the byte to set.

Returns *(int, int)* – *address*, item at *address* (None if empty).

See also:

setdefault() setdefault_restore()

setdefault_restore(*address*, *item*)
Restores a *setdefault()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to restore, None if empty.

See also:

setdefault() setdefault_backup()

shift(*offset*)
Shifts the items.

Parameters **offset** (*int*) – Signed amount of address shifting.

See also:

shift_backup() shift_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

2	3	4	5	6	7	8	9	10	11	12
			[A	B	C]		[x	y	z]	
	[A	B	C]		[x	y	z]			

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.shift(-2)
>>> memory.to_blocks()
[[3, b'ABC'], [7, b'xyz']]
```

~~~

2	3	4	5	6	7	8	9	10	11	12
	[[[		[A	B	C]		[x	y	z]	
	[y	z]								

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']], start=3)
>>> memory.shift(-8)
>>> memory.to_blocks()
[[2, b'yz']]
```

shift_backup(*offset*)
Backups a *shift()* operation.

Parameters **offset** (*int*) – Signed amount of address shifting.

Returns (int, ImmutableMemory) – Shifting, backup memory region.

See also:

shift() shift_restore()
~~~

shift_restore(*offset*, *backup*)
Restores an *shift()* operation.

Parameters

- **offset** (*int*) – Signed amount of address shifting.
- **backup** (ImmutableMemory) – Backup memory region to restore.

See also:

shift() shift_backup()

span
Memory address span.

A tuple holding both *start* and *endex*.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().span
(0, 0)
>>> Memory(start=1, endex=8).span
(1, 8)
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.span
(1, 8)
```

Type tuple of int

start
Inclusive start address.

This property holds the inclusive start address of the virtual space. By default, it is the current minimum inclusive start address of the first stored block.

If *trim_start* not None, that is returned.

If the memory has no data and no trimming, 0 is returned.
~~~

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().start
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.start
1
```

~~~

0	1	2	3	4	5	6	7	8
	[[[				[x	y	z]	

```
>>> memory = Memory.from_blocks([[5, b'xyz']], start=1)
>>> memory.start
1
```

Type int

to_blocks(*start*, *endex*)

Exports into blocks.

Exports data blocks within an address range, converting them into standalone `bytes` objects.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.

Returns *list of blocks* – Exported data blocks.

See also:

blocks() from_blocks()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> memory.to_blocks()
[[1, b'AB'], [5, b'x'], [7, b'123']]
>>> memory.to_blocks(2, 9)
[[2, b'B'], [5, b'x'], [7, b'12']]
>>> memory.to_blocks(3, 5)]
[]
```

to_bytes(*start*, *endex*)

Exports into bytes.

Exports data within an address range, converting into a standalone `bytes` object.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.

Returns *bytes* – Exported data bytes.

See also:

from_bytes() view()

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> memory = Memory.from_bytes(b'')
>>> memory.to_bytes()
b''
```

~~~

0	1	2	3	4	5	6	7	8
		[A	B	C	x	y	z]	

```
>>> memory = Memory.from_bytes(b'ABCxyz', 2)
>>> memory.to_bytes()
b'ABCxyz'
>>> memory.to_bytes(start=4)
b'Cxyz'
>>> memory.to_bytes(endex=6)
b'ABCx'
>>> memory.to_bytes(4, 6)
b'Cx'
```

trim_endex

Trimming exclusive end address.

Any data at or after this address is automatically discarded. Disabled if `None`.

Type int
~~~

trim_span
Trimming span addresses.

A `tuple` holding *trim_start* and *trim_endex*.

Type tuple of int

trim_start
Trimming start address.

Any data before this address is automatically discarded. Disabled if `None`.

Type int

update(*data*, *clear*, ***kwargs*)
Updates data.

Parameters

- **data** (*iterable*) – Data to update with. Can be either another memory, an (address, value) mapping, or an iterable of (address, value) pairs.
- **clear** (*bool*) – Clears the target range before writing data. Useful only if *data* is a *Memory* with empty spaces.

See also:

update_backup() update_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
					[A	B	C]				
	[x	y]			[A	B	C]				
	[x	y	@]		[A	?	C]				

```
>>> memory = Memory()
>>> memory.update(Memory.from_bytes(b'ABC', 5))
>>> memory.to_blocks()
[[5, b'ABC']]
>>> memory.update({1: b'x', 2: ord('y')})
>>> memory.to_blocks()
[[1, b'xy'], [5, b'ABC']]
>>> memory.update([(6, b'?'), (3, ord('@'))])
>>> memory.to_blocks()
[[1, b'xy@'], [5, b'A?C']]
```

update_backup(*data*, *clear*, ***kwargs*)
Backups an *update()* operation.

Parameters

- **data** (*iterable*) – Data to update with. Can be either another memory, an (address, value) mapping, or an iterable of (address, value) pairs.
- **clear** (*bool*) – Clears the target range before writing data. Useful only if *data* is a *Memory* with empty spaces.

Returns list of `ImmutableMemory` – Backup memory regions.

See also:

update() update_restore()

`update_restore`(*backups*)
Restores an *update()* operation.

Parameters **backups** (list of `ImmutableMemory`) – Backup memory regions to restore.

See also:

update() update_backup()

`validate`()
Validates internal structure.

It makes sure that all the allocated blocks are sorted by block start address, and that all the blocks are non-overlapping.

Raises **ValueError** – Invalid data detected (see exception message).

`values`(*start*, *endex*, *pattern*)
Iterates over values.

Iterates over values, from *start* to *endex*. Implemets the interface of `dict`.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered. If `Ellipsis`, the iterator is infinite.
- **pattern** (*items*) – Pattern of values to fill emptiness.

Yields *int* – Range values.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9
			A	B	C	D	A		
			65	66	67	68	65		

```
>>> from itertools import islice
>>> memory = Memory()
>>> list(memory.values(endex=8))
[None, None, None, None, None, None, None, None]
>>> list(memory.values(3, 8))
[None, None, None, None, None]
>>> list(islice(memory.values(3, ...), 7))
[None, None, None, None, None, None, None]
>>> list(memory.values(3, 8, b'ABCD'))
[65, 66, 67, 68, 65]
```

~~~

0	1	2	3	4	5	6	7	8	9
	[A	B	C]	<1	2>	[x	y	z]	
	65	66	67			120	121	122	
	65	66	67	49	50	120	121	122	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> list(memory.values())
[65, 66, 67, None, None, 120, 121, 122]
>>> list(memory.values(3, 8))
[67, None, None, 120, 121]
>>> list(islice(memory.values(3, ...), 7))
[67, None, None, 120, 121, 122, None]
>>> list(memory.values(3, 8, b'0123'))
[67, 49, 50, 120, 121]
```

view(*start*, *endex*)

Creates a view over a range.

Creates a memory view over the selected address range. Data within the range is required to be contiguous.

Parameters

- **start** (*int*) – Inclusive start of the viewed range. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end of the viewed range. If `None`, *endex* is considered.

Returns `memoryview` – A view of the selected address range.

Raises **ValueError** – Data not contiguous (see *contiguous*).

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> bytes(memory.view(2, 5))
b'BCD'
>>> bytes(memory.view(9, 10))
b'y'
>>> memory.view()
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.view(0, 6)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```
~~~

write(*address*, *data*, *clear*)
Writes data.

Parameters

- **address** (*int*) – Address where to start writing data.
- **data** (*bytes*) – Data to write.
- **clear** (*bool*) – Clears the target range before writing data. Useful only if *data* is a `ImmutableMemory` with empty spaces.

See also:

write_backup() write_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	[A	B	C]		[1	2	3	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.write(5, b'123')
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'123z']]
```

write_backup(*address*, *data*, *clear*)
Backups a *write()* operation.

Parameters

- **address** (*int*) – Address where to start writing data.
- **data** (*bytes*) – Data to write.
- **clear** (*bool*) – Clears the target range before writing data. Useful only if *data* is a *Memory* with empty spaces.

Returns list of `ImmutableMemory` – Backup memory regions.

See also:

write() write_restore()

write_restore(*backups*)
Restores a *write()* operation.

Parameters **backups** (list of `ImmutableMemory`) – Backup memory regions to restore.

See also:

write() write_backup()

cbytesparse.c.bytesparse

class cbytesparse.c.**bytesparse**(*start: Optional[Address] = None*, **args: Any*, *endex: Optional[Address] = None*)

Wrapper for more *bytearray* compatibility.

This wrapper class can make *Memory* closer to the actual `bytearray` API.

For instantiation, please refer to `bytearray.__init__()`.

With respect to *Memory*, negative addresses are not allowed. Instead, negative addresses are to consider as referred to *endex*.

Parameters

- **source** – The optional *source* parameter can be used to initialize the array in a few different ways:
  - If it is a string, you must also give the *encoding* (and optionally, *errors*) parameters; it then converts the string to bytes using `str.encode()`.
  - If it is an integer, the array will have that size and will be initialized with null bytes.
  - If it is an object conforming to the buffer interface, a read-only buffer of the object will be used to initialize the byte array.
  - If it is an iterable, it must be an iterable of integers in the range 0 <= x < 256, which are used as the initial contents of the array.
- **encoding** (*str*) – Optional string encoding.
- **errors** (*str*) – Optional string error management.
- **start** (*int*) – Optional memory start address. Anything before will be trimmed away. If *source* is provided, its data start at this address (0 if *start* is `None`).
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be trimmed away.

Methods

__init__	
append	Appends a single item.
append_backup	Backups an *append()* operation.
append_restore	Restores an *append()* operation.
block_span	
blocks	
bound	
clear	
clear_backup	
clear_restore	Restores a *clear()* operation.
content_items	Iterates over content address and value pairs.

continues on next page

Table 9 – continued from previous page

content_keys	Iterates over content addresses.
content_values	Iterates over content values.
copy	Creates a shallow copy.
count	
crop	
crop_backup	
crop_restore	Restores a *crop()* operation.
cut	
delete	
delete_backup	
delete_restore	Restores a *delete()* operation.
equal_span	
extend	Concatenates items.
extend_backup	Backups an *extend()* operation.
extend_restore	Restores an *extend()* operation.
extract	
fill	
fill_backup	
fill_restore	Restores a *fill()* operation.
find	
flood	
flood_backup	
flood_restore	Restores a *flood()* operation.
from_blocks	
from_bytes	
from_items	Creates a virtual memory from a iterable address/byte mapping.
from_memory	
from_values	Creates a virtual memory from a byte-like sequence.
fromhex	Creates a virtual memory from an hexadecimal string.
gaps	
get	

continues on next page

Table 9 – continued from previous page

hex	Converts into an hexadecimal string.
index	
insert	
insert_backup	
insert_restore	Restores an *insert()* operation.
intervals	
items	
keys	
ofind	
peek	
poke	
poke_backup	
poke_restore	Restores a *poke()* operation.
pop	
pop_backup	
pop_restore	Restores a *pop()* operation.
popitem	Pops the last item.
popitem_backup	Backups a *popitem()* operation.
popitem_restore	Restores a *popitem()* operation.
remove	
remove_backup	
remove_restore	Restores a *remove()* operation.
reserve	
reserve_backup	
reserve_restore	Restores a *reserve()* operation.
reverse	Reverses the memory in-place.
rfind	
rindex	
rofind	
rvalues	

continues on next page

Table 9 – continued from previous page

setdefault	
setdefault_backup	
setdefault_restore	Restores a *setdefault()* operation.
shift	
shift_backup	
shift_restore	Restores an *shift()* operation.
to_blocks	Exports into blocks.
to_bytes	Exports into bytes.
update	Updates data.
update_backup	Backups an *update()* operation.
update_restore	Restores an *update()* operation.
validate	Validates internal structure.
values	
view	
write	
write_backup	
write_restore	Restores a *write()* operation.

Attributes

content_endex	Exclusive content end address.
content_endin	Inclusive content end address.
content_parts	Number of blocks.
content_size	Actual content size.
content_span	Memory content address span.
content_start	Inclusive content start address.
contiguous	Contains contiguous data.
endex	Exclusive end address.
endin	Inclusive end address.
span	Memory address span.
start	Inclusive start address.
trim_endex	
trim_span	
trim_start	

__add__(*value*, */*)
Return self+value.

__bool__()
self != 0

__bytes__()
Creates a bytes clone.

Returns bytes – Cloned data.

Raises ValueError – Data not contiguous (see *contiguous*).

__contains__()
Checks if some items are contained.

Parameters item (*items*) – Items to find. Can be either some byte string or an integer.

Returns *bool* – Item is contained.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C]		[1	2	3]		[x	y	z]

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'123'], [9, b'xyz']])
>>> b'23' in memory
True
>>> ord('y') in memory
True
>>> b'$' in memory
False
```

__copy__()
Creates a shallow copy.

Note: The Cython implementation actually creates a deep copy.

Returns ImmutableMemory – Shallow copy.

__deepcopy__()
Creates a deep copy.

Returns ImmutableMemory – Deep copy.

__delitem__(*key*, /)
Delete self[key].

__eq__()
Equality comparison.

Parameters other (Memory) – Data to compare with *self*.

If it is a ImmutableMemory, all of its blocks must match.

If it is a bytes, a bytearray, or a memoryview, it is expected to match the first and only stored block.

Otherwise, it must match the first and only stored block, via iteration over the stored values.

Returns *bool* – *self* is equal to *other*.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> data = b'Hello, World!'
>>> memory = Memory.from_bytes(data)
>>> memory == data
True
>>> memory.shift(1)
>>> memory == data
True
```

```
>>> data = b'Hello, World!'
>>> memory = Memory.from_bytes(data)
>>> memory == list(data)
True
>>> memory.shift(1)
>>> memory == list(data)
True
```

__ge__(*value*, /)
Return self>=value.

__getitem__(*key*, /)
Return self[key].

__gt__(*value*, /)
Return self>value.

__hash__ = None

__iadd__(*value*, /)
Return self+=value.

__imul__(*value*, /)
Return self*=value.

__init__(**args*, ***kwargs*)

__iter__()
Iterates over values.

Iterates over values between *start* and *endex*.

Yields *int* – Value as byte integer, or None.

__le__(*value*, /)
Return self<=value.

__len__()
Actual length.

Computes the actual length of the stored items, i.e. (*endex* - *start*). This will consider any trimmings being active.

Returns *int* – Memory length.

__lt__(*value*, /)
Return self<value.

__mul__(*value*, /)
Return self*value.

__ne__(*value*, /)
Return self!=value.

__new__(**kwargs*)

__radd__(*value*, /)
Return value+self.

__reduce__()
bytesparse.__reduce_cython__(self)

__repr__()
Return repr(self).

__reversed__()
Iterates over values, reversed order.

Iterates over values between *start* and *endex*, in reversed order.

Yields *int* – Value as byte integer, or `None`.

__rmul__(*value*, /)
Return value*self.

__setitem__(*key*, *value*, /)
Set self[key] to value.

__setstate__()
bytesparse.__setstate_cython__(self, __pyx_state)

__str__()
String representation.

If *content_size* is lesser than `STR_MAX_CONTENT_SIZE`, then the memory is represented as a list of blocks.

If exceeding, it is equivalent to *__repr__()*.

Returns *str* – String representation.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B	C]				[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [7, b'xyz']])
>>> str(memory)
<[[1, b'ABC'], [7, b'xyz']]>
```

_block_index_at(*address*)
Locates the block enclosing an address.

Returns the index of the block enclosing the given address.

Parameters **address** (*int*) – Address of the target item.

Returns *int* – Block index if found, None otherwise.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
	0	0	0	0		1		2	2	2	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> [memory._block_index_at(i) for i in range(12)]
[None, 0, 0, 0, 0, None, 1, None, 2, 2, 2, None]
```

_block_index_endex(*address*)

Locates the last block before an address range.

Returns the index of the last block whose end address is lesser than or equal to *address*.

Useful to find the termination block index in a ranged search.

Parameters **address** (*int*) – Exclusive end address of the scanned range.

Returns *int* – First block index before *address*.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
0	1	1	1	1	1	2	2	3	3	3	3

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> [memory._block_index_endex(i) for i in range(12)]
[0, 1, 1, 1, 1, 1, 2, 2, 3, 3, 3, 3]
```

_block_index_start(*address*)

Locates the first block inside of an address range.

Returns the index of the first block whose start address is greater than or equal to *address*.

Useful to find the initial block index in a ranged search.

Parameters **address** (*int*) – Inclusive start address of the scanned range.

Returns *int* – First block index since *address*.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
0	0	0	0	0	1	1	2	2	2	2	3

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> [memory._block_index_start(i) for i in range(12)]
[0, 0, 0, 0, 0, 1, 1, 2, 2, 2, 2, 3]
```

_pretrim_endex(*start_min*, *size*)
Trims final data.

Low-level method to manage trimming of data starting from an address.

Parameters

- **start_min** (*int*) – Starting address of the erasure range. If `None`, *trim_endex* minus *size* is considered.
- **size** (*int*) – Size of the erasure range.

See also:

_pretrim_endex_backup()

_pretrim_endex_backup(*start_min*, *size*)
Backups a *_pretrim_endex()* operation.

Parameters

- **start_min** (*int*) – Starting address of the erasure range. If `None`, *trim_endex* minus *size* is considered.
- **size** (*int*) – Size of the erasure range.

Returns `ImmutableMemory` – Backup memory region.

See also:

_pretrim_endex()

_pretrim_start(*endex_max*, *size*)
Trims initial data.

Low-level method to manage trimming of data starting from an address.

Parameters

- **endex_max** (*int*) – Exclusive end address of the erasure range. If `None`, *trim_start* plus *size* is considered.
- **size** (*int*) – Size of the erasure range.

See also:

_pretrim_start_backup()

_pretrim_start_backup(*endex_max*, *size*)
Backups a *_pretrim_start()* operation.

Parameters

- **endex_max** (*int*) – Exclusive end address of the erasure range. If None, *trim_start* plus *size* is considered.
- **size** (*int*) – Size of the erasure range.

Returns ImmutableMemory – Backup memory region.

See also:

_pretrim_start()

_rectify_address(*address*)

Rectifies an address.

In case the provided *address* is negative, it is recomputed as referred to *endex*.

In case the rectified address would still be negative, an exception is raised.

Parameters **address** (int) – Address to be rectified.

Returns *int* – Rectified address.

Raises **IndexError** – The rectified address would still be negative.

_rectify_span(*start*, *endex*)

Rectifies an address span.

In case a provided address is negative, it is recomputed as referred to *endex*.

In case the rectified address would still be negative, it is clamped to address zero.

Parameters

- **start** (*int*) – Inclusive start address for rectification. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for rectification. If None, *endex* is considered.

Returns *pair of int* – Rectified address span.

append(*item*)

Appends a single item.

Parameters **item** (*int*) – Value to append. Can be a single byte string or integer.

See also:

append_backup() append_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> memory = Memory()
>>> memory.append(b'$')
>>> memory.to_blocks()
[[0, b'$']]
```

~~~
~~~

```
>>> memory = Memory()
>>> memory.append(3)
>>> memory.to_blocks()
[[0, b'\x03']]
```

append_backup()
Backups an *append()* operation.

Returns *None* – Nothing.

See also:

append() append_restore()

append_restore()
Restores an *append()* operation.

See also:

append() append_backup()

block_span(*address*)

blocks(*start*, *endex*)

bound(*start*, *endex*)

clear(*start*, *endex*)

clear_backup(*start*, *endex*)

clear_restore(*backup*)
Restores a *clear()* operation.

Parameters **backup** (ImmutableMemory) – Backup memory region to restore.

See also:

clear() clear_backup()

content_endex
Exclusive content end address.

This property holds the exclusive end address of the memory content. By default, it is the current maximum exclusive end address of the last stored block.

If the memory has no data and no trimming, *start* is returned.

Trimming is considered only for an empty memory.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_endex
0
>>> Memory(endex=8).content_endex
0
>>> Memory(start=1, endex=8).content_endex
1
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_endex
8
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_endex
4
```

Type int

content_endin

Inclusive content end address.

This property holds the inclusive end address of the memory content. By default, it is the current maximum inclusive end address of the last stored block.

If the memory has no data and no trimming, *start* minus one is returned.

Trimming is considered only for an empty memory.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_endin
-1
>>> Memory(endex=8).content_endin
-1
>>> Memory(start=1, endex=8).content_endin
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_endin
7
```
~~~

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_endin
3
```

Type int

content_items(*start*, *endex*)

Iterates over content address and value pairs.

Parameters

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields *int* – Content address and value pairs.

See also:

meth:*content_keys* meth:*content_values*

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> dict(memory.content_items())
{1: 65, 2: 66, 5: 120, 7: 49, 8: 50, 9: 51}
>>> dict(memory.content_items(2, 9))
{2: 66, 5: 120, 7: 49, 8: 50}
>>> dict(memory.content_items(3, 5))
{}
```

content_keys(*start*, *endex*)

Iterates over content addresses.

Parameters

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields *int* – Content addresses.

See also:

meth:*content_items* meth:*content_values*
~~~

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> list(memory.content_keys())
[1, 2, 5, 7, 8, 9]
>>> list(memory.content_keys(2, 9))
[2, 5, 7, 8]
>>> list(memory.content_keys(3, 5))
[]
```

content_parts

Number of blocks.

Returns *int* – The number of blocks.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_parts
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_parts
2
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_parts
1
```

content_size

Actual content size.

Returns *int* – The sum of all block lengths.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_size
0
>>> Memory(start=1, endex=8).content_size
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_size
6
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_size
3
```

content_span

Memory content address span.

A `tuple` holding both *content_start* and *content_endex*.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_span
(0, 0)
>>> Memory(start=1).content_span
(1, 1)
>>> Memory(endex=8).content_span
(0, 0)
>>> Memory(start=1, endex=8).content_span
(1, 1)
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	
~~~

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_span
(1, 8)
```

Type tuple of int

content_start
Inclusive content start address.

This property holds the inclusive start address of the memory content. By default, it is the current minimum inclusive start address of the first stored block.

If the memory has no data and no trimming, 0 is returned.

Trimming is considered only for an empty memory.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().content_start
0
>>> Memory(start=1).content_start
1
>>> Memory(start=1, endex=8).content_start
1
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_start
1
```

~~~

0	1	2	3	4	5	6	7	8
	[[[				[x	y	z]	

```
>>> memory = Memory.from_blocks([[5, b'xyz']], start=1)
>>> memory.content_start
5
```

Type int

content_values(*start*, *endex*)
Iterates over content values.

Parameters

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields *int* – Content values.

See also:

meth:*content_items* meth:*content_keys*

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> list(memory.content_values())
[65, 66, 120, 49, 50, 51]
>>> list(memory.content_values(2, 9))
[66, 120, 49, 50]
>>> list(memory.content_values(3, 5))
[]
```

contiguous
Contains contiguous data.

The memory is considered to have contiguous data if there is no empty space between blocks.

If trimming is defined, there must be no empty space also towards it.

Type bool

copy()
Creates a shallow copy.

Note: The Cython implementation actually creates a deep copy.

Returns ImmutableMemory – Shallow copy.

count(*item*, *start*, *endex*)

crop(*start*, *endex*)

crop_backup(*start*, *endex*)

crop_restore(*backup_start*, *backup_endex*)
Restores a *crop()* operation.

Parameters

- **backup_start** (ImmutableMemory) – Backup memory region to restore at the beginning.
- **backup_endex** (ImmutableMemory) – Backup memory region to restore at the end.

See also:

crop() crop_backup()

cut(*start*, *endex*, *bound*)

delete(*start*, *endex*)

delete_backup(*start*, *endex*)

delete_restore(*backup*)
Restores a *delete()* operation.

Parameters **backup** (ImmutableMemory) – Backup memory region

See also:

delete() delete_backup()

endex
Exclusive end address.

This property holds the exclusive end address of the virtual space. By default, it is the current maximmum exclusive end address of the last stored block.

If *trim_endex* not None, that is returned.

If the memory has no data and no trimming, *start* is returned.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().endex
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.endex
8
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.endex
8
```

Type int

endin
Inclusive end address.

This property holds the inclusive end address of the virtual space. By default, it is the current maximmum inclusive end address of the last stored block.

If *trim_endex* not `None`, that minus one is returned.

If the memory has no data and no trimming, *start* is returned.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().endin
-1
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.endin
7
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.endin
7
```

Type int

equal_span(*address*)

extend(*items*, *offset*)
Concatenates items.

Equivalent to `self += items`.

Parameters

- **items** (*items*) – Items to append at the end of the current virtual space.
- **offset** (*int*) – Optional offset w.r.t. *content_endex*.

See also:

extend_backup() extend_restore()

extend_backup(*offset*)
Backups an *extend()* operation.

Parameters **offset** (*int*) – Optional offset w.r.t. *content_endex*.

Returns *int* – Content exclusive end address.

See also:

extend() extend_restore()

extend_restore(*content_endex*)
Restores an *extend()* operation.

Parameters **content_endex** (*int*) – Content exclusive end address to restore.

See also:

extend() extend_backup()

extract(*start*, *endex*, *pattern*, *step*, *bound*)

fill(*start*, *endex*, *pattern*)

fill_backup(*start*, *endex*)

fill_restore(*backup*)
Restores a *fill()* operation.

Parameters **backup** (ImmutableMemory) – Backup memory region to restore.

See also:

fill() fill_backup()

find(*item*, *start*, *endex*)

flood(*start*, *endex*, *pattern*)

flood_backup(*start*, *endex*)

flood_restore(*gaps*)
Restores a *flood()* operation.

Parameters **gaps** (*list of open intervals*) – Backup memory gaps to restore.

See also:

flood() flood_backup()

classmethod from_blocks(*blocks*, *offset*, *start*, *endex*, *copy*, *validate*)

classmethod from_bytes(*data*, *offset*, *start*, *endex*, *copy*, *validate*)

classmethod from_items(*items*, *offset*, *start*, *endex*, *validate*)
Creates a virtual memory from a iterable address/byte mapping.

Parameters

- **items** (*iterable address/byte mapping*) – An iterable mapping of address to byte values. Values of None are translated as gaps. When an address is stated multiple times, the last is kept.
- **offset** (*int*) – An address offset applied to all the values.
- **start** (*int*) – Optional memory start address. Anything before will be trimmed away.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be trimmed away.
- **validate** (*bool*) – Validates the resulting ImmutableMemory object.

Returns ImmutableMemory – The resulting memory object.

Raises **ValueError** – Some requirements are not satisfied.

See also:

to_bytes()

Examples

```
>>> from bytesparse.inplace import Memory
```

```
>>> memory = Memory.from_values({})
>>> memory.to_blocks()
[]
```

~~~

0	1	2	3	4	5	6	7	8
		[A	Z]		[x]			

```
>>> items = [
...     (0, ord('A')),
...     (1, ord('B')),
...     (3, ord('x')),
...     (1, ord('Z')),
... ]
>>> memory = Memory.from_items(items, offset=2)
>>> memory.to_blocks()
[[2, b'AZ'], [5, b'x']]
```

classmethod **from_memory**(*memory*, *offset*, *start*, *endex*, *copy*, *validate*)

classmethod **from_values**(*values*, *offset*, *start*, *endex*, *validate*)

Creates a virtual memory from a byte-like sequence.

Parameters

- **values** (*iterable byte-like sequence*) – An iterable sequence of byte values. Values of None are translated as gaps.
- **offset** (*int*) – An address offset applied to all the values.
- **start** (*int*) – Optional memory start address. Anything before will be trimmed away.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be trimmed away.
- **validate** (*bool*) – Validates the resulting ImmutableMemory object.

Returns ImmutableMemory – The resulting memory object.

Raises **ValueError** – Some requirements are not satisfied.

See also:

to_bytes()
~~~

Examples

```
>>> from bytesparse.inplace import Memory
```

```
>>> memory = Memory.from_values(range(0))
>>> memory.to_blocks()
[]
```

~~~

0	1	2	3	4	5	6	7	8
		[A	B	C	D	E]		

```
>>> memory = Memory.from_values(range(ord('A'), ord('F')), offset=2)
>>> memory.to_blocks()
[[2, b'ABCDE']]
```

classmethod **fromhex**(*string*)

Creates a virtual memory from an hexadecimal string.

Parameters **string** (*str*) – Hexadecimal string.

Returns ImmutableMemory – The resulting memory object.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> memory = Memory.fromhex('')
>>> bytes(memory)
b''
```

~~~

```
>>> memory = Memory.fromhex('48656C6C6F2C20576F726C6421')
>>> bytes(memory)
b'Hello, World!'
```

gaps(*start*, *endex*)

get(*address*, *default*)

hex(**args*)

Converts into an hexadecimal string.

Parameters

- **sep** (*str*) – Separator string between bytes. Defaults to an emoty string if not provided. Available since Python 3.8.
- **bytes_per_sep** (*int*) – Number of bytes grouped between separators. Defaults to one byte per group. Available since Python 3.8.

Returns *str* – Hexadecimal string representation.

Raises **ValueError** – Data not contiguous (see *contiguous*).

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().hex() == ''
True
```

~~~

```
>>> memory = Memory.from_bytes(b'Hello, World!')
>>> memory.hex()
48656c6c6f2c20576f726c6421
>>> memory.hex('.')
48.65.6c.6c.6f.2c.20.57.6f.72.6c.64.21
>>> memory.hex('.', 4)
48.656c6c6f.2c20576f.726c6421
```

index(*item*, *start*, *endex*)

insert(*address*, *data*)

insert_backup(*address*, *data*)

insert_restore(*address*, *backup*)
Restores an *insert()* operation.

Parameters

- **address** (*int*) – Address of the insertion point.
- **backup** (*Memory*) – Backup memory region to restore.

See also:

insert() insert_backup()

intervals(*start*, *endex*)

items(*start*, *endex*, *pattern*)

keys(*start*, *endex*)

ofind(*item*, *start*, *endex*)

peek(*address*)

poke(*address*, *item*)

poke_backup(*address*)

poke_restore(*address*, *item*)
Restores a *poke()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to restore.

See also:

poke() poke_backup()

pop(*address*, *default*)
~~~

pop_backup(*address*)

pop_restore(*address*, *item*)
Restores a *pop()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to restore, None if empty.

See also:

pop() pop_backup()

popitem()
Pops the last item.

Returns *(int, int)* – Address and value of the last item.

See also:

popitem_backup() popitem_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A]								[y	z]	

```
>>> memory = Memory.from_blocks([[1, b'A'], [9, b'yz']])
>>> memory.popitem()  # -> ord('z') = 122
(10, 122)
>>> memory.popitem()  # -> ord('y') = 121
(9, 121)
>>> memory.popitem()  # -> ord('A') = 65
(1, 65)
>>> memory.popitem()
Traceback (most recent call last):
    ...
KeyError: empty
```

popitem_backup()
Backups a *popitem()* operation.

Returns *(int, int)* – Address and value of the last item.

See also:

popitem() popitem_restore()

popitem_restore(*address*, *item*)
Restores a *popitem()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int*) – Item to restore.

See also:

popitem() popitem_backup()

remove(*item*, *start*, *endex*)

remove_backup(*item*, *start*, *endex*)

remove_restore(*backup*)
Restores a *remove()* operation.

Parameters **backup** (*Memory*) – Backup memory region.

See also:

remove() remove_backup()

reserve(*address*, *size*)

reserve_backup(*address*, *size*)

reserve_restore(*address*, *backup*)
Restores a *reserve()* operation.

Parameters

- **address** (*int*) – Address of the reservation point.
- **backup** (ImmutableMemory) – Backup memory region to restore.

See also:

reserve() reserve_backup()

reverse()
Reverses the memory in-place.

Data is reversed within the memory *span*.

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
	[z	y	x]		[$]		[D	C	B	A]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.reverse()
>>> memory.to_blocks()
[[1, b'zyx'], [5, b'$'], [7, b'DCBA']]
```

~~~

0	1	2	3	4	5	6	7	8	9	10	11
		[[[		[A	B	C]				)))	
		[[[			[C	B	A]			)))	
~~~

```
>>> memory = Memory.from_bytes(b'ABCD', 3, start=2, endex=10)
>>> memory.reverse()
>>> memory.to_blocks()
[[5, b'CBA']]
```

rfind(*item*, *start*, *endex*)

rindex(*item*, *start*, *endex*)

rofind(*item*, *start*, *endex*)

rvalues(*start*, *endex*, *pattern*)

setdefault(*address*, *default*)

setdefault_backup(*address*)

setdefault_restore(*address*, *item*)
Restores a *setdefault()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to restore, None if empty.

See also:

setdefault() setdefault_backup()

shift(*offset*)

shift_backup(*offset*)

shift_restore(*offset*, *backup*)
Restores an *shift()* operation.

Parameters

- **offset** (*int*) – Signed amount of address shifting.
- **backup** (ImmutableMemory) – Backup memory region to restore.

See also:

shift() shift_backup()

span
Memory address span.

A tuple holding both *start* and *endex*.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().span
(0, 0)
>>> Memory(start=1, endex=8).span
(1, 8)
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.span
(1, 8)
```

Type tuple of int

start
Inclusive start address.

This property holds the inclusive start address of the virtual space. By default, it is the current minimum inclusive start address of the first stored block.

If *trim_start* not `None`, that is returned.

If the memory has no data and no trimming, 0 is returned.

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> Memory().start
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.start
1
```

~~~

0	1	2	3	4	5	6	7	8
	[[[				[x	y	z]	

```
>>> memory = Memory.from_blocks([[5, b'xyz']], start=1)
>>> memory.start
1
```

Type int

to_blocks(*start*, *endex*)
Exports into blocks.

Exports data blocks within an address range, converting them into standalone `bytes` objects.
~~~

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.

Returns *list of blocks* – Exported data blocks.

See also:

blocks() from_blocks()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> memory.to_blocks()
[[1, b'AB'], [5, b'x'], [7, b'123']]
>>> memory.to_blocks(2, 9)
[[2, b'B'], [5, b'x'], [7, b'12']]
>>> memory.to_blocks(3, 5)]
[]
```

`to_bytes`(*start*, *endex*)

Exports into bytes.

Exports data within an address range, converting into a standalone `bytes` object.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.

Returns *bytes* – Exported data bytes.

See also:

from_bytes() view()

Examples

```
>>> from cbytesparse.c import Memory
```

```
>>> memory = Memory.from_bytes(b'')
>>> memory.to_bytes()
b''
```

~~~

0	1	2	3	4	5	6	7	8
		[A	B	C	x	y	z]	
~~~

```
>>> memory = Memory.from_bytes(b'ABCxyz', 2)
>>> memory.to_bytes()
b'ABCxyz'
>>> memory.to_bytes(start=4)
b'Cxyz'
>>> memory.to_bytes(endex=6)
b'ABCx'
>>> memory.to_bytes(4, 6)
b'Cx'
```

trim_endex

trim_span

trim_start

update(*data*, *clear*, ***kwargs*)

Updates data.

Parameters

- **data** (*iterable*) – Data to update with. Can be either another memory, an (address, value) mapping, or an iterable of (address, value) pairs.
- **clear** (*bool*) – Clears the target range before writing data. Useful only if *data* is a *Memory* with empty spaces.

See also:

update_backup() update_restore()

Examples

```
>>> from cbytesparse.c import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
					[A	B	C]				
	[x	y]			[A	B	C]				
	[x	y	@]		[A	?	C]				

```
>>> memory = Memory()
>>> memory.update(Memory.from_bytes(b'ABC', 5))
>>> memory.to_blocks()
[[5, b'ABC']]
>>> memory.update({1: b'x', 2: ord('y')})
>>> memory.to_blocks()
[[1, b'xy'], [5, b'ABC']]
>>> memory.update([(6, b'?'), (3, ord('@'))])
>>> memory.to_blocks()
[[1, b'xy@'], [5, b'A?C']]
```

update_backup(*data*, *clear*, ***kwargs*)

Backups an *update()* operation.

Parameters

- **data** (*iterable*) – Data to update with. Can be either another memory, an (address, value) mapping, or an iterable of (address, value) pairs.
- **clear** (*bool*) – Clears the target range before writing data. Useful only if *data* is a *Memory* with empty spaces.

Returns list of ImmutableMemory – Backup memory regions.

See also:

update() update_restore()

update_restore(*backups*)
Restores an *update()* operation.

Parameters **backups** (list of ImmutableMemory) – Backup memory regions to restore.

See also:

update() update_backup()

validate()
Validates internal structure.

It makes sure that all the allocated blocks are sorted by block start address, and that all the blocks are non-overlapping.

Raises **ValueError** – Invalid data detected (see exception message).

values(*start*, *endex*, *pattern*)

view(*start*, *endex*)

write(*address*, *data*, *clear*)

write_backup(*address*, *data*, *clear*)

write_restore(*backups*)
Restores a *write()* operation.

Parameters **backups** (list of ImmutableMemory) – Backup memory regions to restore.

See also:

write() write_backup()

CHAPTER

FOUR

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.2 Documentation improvements

cbytesparse could always use more documentation, whether as part of the official cbytesparse docs, in docstrings, or even on the web in blog posts, articles, and such.

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/TexZK/cbytesparse/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

4.4 Development

To set up *cbytesparse* for local development:

1. Fork cbytesparse (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/cbytesparse.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

4.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`).
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

4.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

CHAPTER

FIVE

AUTHORS

- Andrea Zoppi - main developer - https://github.com/TexZK

CHAPTER

SIX

CHANGELOG

6.1 0.0.5 (2022-02-22)

- Following the `bytesparse` Python package, version `0.0.5`.
- Added `bytesparse` class, closer to `bytearray` than `Memory`.
- Added registration to the abstract base classes.
- Added missing abstract and ported methods.
- Added cut feature.
- Added more helper methods.
- Fixed values iteration.
- Improved extraction performance.
- Improved testing.

6.2 0.0.4 (2022-01-09)

- Following the `bytesparse` Python package, version `0.0.4`.
- Refactored current implementation as the `c` sub-module.
- Removed experimental backup feature.
- Added dedicated methods to backup/restore mutated state.
- Fixed some write/insert bugs.
- Fixed some trim/bound bugs.
- Methods sorted by name.
- Removed useless functions.

6.3 0.0.2 (2022-01-03)

- Forced extension compilation.
- Using explicit factory methods instead of constructor arguments.
- Added block collapsing helper function.
- Minor fixes.
- Improved test suite.

6.4 0.0.1 (2021-12-27)

- First release on PyPI.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

M

O

P

R

S

T

U

V

W