bytesparse

Release 0.1.1

Andrea Zoppi

Mar 07, 2024

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 consists of 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 provides a pure Python implementation, for maximum compatibility.

Its implementation should be correct and robust, whilst trying to be as fast as common sense suggests. This means that the code should be reasonably optimized for general use, while still providing features that are less likely to be used, yet compatible with the existing Python API (e.g. `bytearray` or `dict`).

The Python implementation can also leverage the capabilities of its powerful `int` type, so that a virtually infinite addressing space can be used, even with negative addresses!

Data chunks are stored as common mutable `bytearray` objects, whose size is limited by the Python engine (i.e. that of `size_t`).

The `bytesparse` package provides the following virtual memory types:

- `bytesparse.Memory`, a generic virtual memory with infinite address range.
- `bytesparse.bytesparse`, a subclass behaving more like `bytearray`.

All the implementations inherit the behavior of `collections.abc.MutableSequence` and `collections.abc.MutableMapping`. Please refer to the collections.abc reference manual for more information about the interface API methods and capabilities.

1.3 Cython implementation

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

Please refer to the `cbytesparse` Python package for more details.

1.4 Examples

Here's a quick usage example of `bytesparse` objects:

```
>>> from bytesparse import Memory
>>> from bytesparse import bytesparse
```

```
>>> m = bytesparse(b'Hello, World!')  # creates from bytes
>>> len(m)  # total length
13
>>> str(m)  # string representation, with bounds and data blocks
"<[[0, b'Hello, World!']]>"
>>> bytes(m)  # exports as bytes
b'Hello, World!'
>>> m.to_bytes()  # exports the whole range as bytes
b'Hello, World!'
```

```
>>> m.extend(b'!!!')  # more emphasis!!!
>>> bytes(m)
b'Hello, World!!!'
```

```
>>> i = m.index(b',')  # gets the address of the comma
>>> m[:i] = b'Ciao'  # replaces 'Hello' with 'Ciao'
>>> bytes(m)
b'Ciao, World!!!'
```

```
>>> i = m.index(b',')  # gets the address of the comma
>>> m.insert(i, b'ne')  # inserts 'ne' to make 'Ciaone' ("big ciao")
>>> bytes(m)
b'Ciaone, World!!!'
```

```
>>> i = m.index(b',')  # gets the address of the comma
>>> m[(i - 2):i] = b' ciao'  # makes 'Ciaone' --> 'Ciao ciao'
>>> bytes(m)
b'Ciao ciao, World!!!'
```

```
>>> m.pop()  # less emphasis --> 33 == ord('!')
33
>>> bytes(m)
b'Ciao ciao, World!!'
```

```
>>> del m[m.index(b'l')]  # makes 'World' --> 'Word'
>>> bytes(m)
b'Ciao ciao, Word!!'
```

```
>>> m.popitem()  # less emphasis --> pops 33 (== '!') at address 16
(16, 33)
>>> bytes(m)
b'Ciao ciao, Word!'
```

```
>>> m.remove(b' ciao')  # self-explanatory
>>> bytes(m)
b'Ciao, Word!'
```

```
>>> i = m.index(b',')  # gets the address of the comma
>>> m.clear(start=i, endex=(i + 2))  # makes empty space between the words
>>> m.to_blocks()  # exports as data block list
[[0, b'Ciao'], [6, b'Word!']]
>>> m.contiguous  # multiple data blocks (emptiness inbetween)
False
>>> m.content_parts  # two data blocks
2
>>> m.content_size  # excluding emptiness
9
>>> len(m)  # including emptiness
11
```

```
>>> m.flood(pattern=b'.')  # replaces emptiness with dots
>>> bytes(m)
b'Ciao..Word!'
>>> m[-2]  # 100 == ord('d')
100
```

```
>>> m.peek(-2)  # 100 == ord('d')
100
>>> m.poke(-2, b'k')  # makes 'Word' --> 'Work'
>>> bytes(m)
b'Ciao..Work!'
```

```
>>> m.crop(start=m.index(b'W'))  # keeps 'Work!'
>>> m.to_blocks()
[[6, b'Work!']]
>>> m.span  # address range of the whole memory
(6, 11)
>>> m.start, m.endex  # same as above
(6, 11)
```

```
>>> m.bound_span = (2, 10)  # sets memory address bounds
>>> str(m)
"<2, [[6, b'Work']], 10>"
>>> m.to_blocks()
[[6, b'Work']]
```

```
>>> m.shift(-6)  # shifts to the left; NOTE: address bounds will cut 2 bytes!
>>> m.to_blocks()
[[2, b'rk']]
>>> str(m)
"<2, [[2, b'rk']], 10>"
```

```
>>> a = bytesparse(b'Ma')
>>> a.write(0, m)  # writes [2, b'rk'] --> 'Mark'
>>> a.to_blocks()
[[0, b'Mark']]
```

```
>>> b = Memory.from_bytes(b'ing', offset=4)
>>> b.to_blocks()
[[4, b'ing']]
```

```
>>> a.write(0, b)  # writes [4, b'ing'] --> 'Marking'
>>> a.to_blocks()
[[0, b'Marking']]
```

```
>>> a.reserve(4, 2)  # inserts 2 empty bytes after 'Mark'
>>> a.to_blocks()
[[0, b'Mark'], [6, b'ing']]
```

```
>>> a.write(4, b'et')  # --> 'Marketing'
>>> a.to_blocks()
[[0, b'Marketing']]
```

```
>>> a.fill(1, -1, b'*')  # fills asterisks between the first and last letters
>>> a.to_blocks()
[[0, b'M*******g']]
```

```
>>> v = a.view(1, -1)  # creates a memory view spanning the asterisks
>>> v[::2] = b'1234'  # replaces even asterisks with numbers
>>> a.to_blocks()
[[0, b'M1*2*3*4g']]
>>> a.count(b'*')  # counts all the asterisks
3
>>> v.release()  # release memory view
```

```
>>> c = a.copy()  # creates a (deep) copy
>>> c == a
True
>>> c is a
False
```

```
>>> del a[a.index(b'*')::2]  # deletes every other byte from the first asterisk
>>> a.to_blocks()
[[0, b'M1234']]
```

```
>>> a.shift(3)  # moves away from the trivial 0 index
>>> a.to_blocks()
[[3, b'M1234']]
>>> list(a.keys())
[3, 4, 5, 6, 7]
>>> list(a.values())
[77, 49, 50, 51, 52]
>>> list(a.items())
[(3, 77), (4, 49), (5, 50), (6, 51), (7, 52)]
```

```
>>> c.to_blocks()  # reminder
[[0, b'M1*2*3*4g']]
>>> c[2::2] = None  # clears (empties) every other byte from the first asterisk
>>> c.to_blocks()
[[0, b'M1'], [3, b'2'], [5, b'3'], [7, b'4']]
>>> list(c.intervals())  # lists all the block ranges
[(0, 2), (3, 4), (5, 6), (7, 8)]
>>> list(c.gaps())  # lists all the empty ranges
[(None, 0), (2, 3), (4, 5), (6, 7), (8, None)]
```

```
>>> c.flood(pattern=b'xy')  # fills empty spaces
>>> c.to_blocks()
[[0, b'M1x2x3x4']]
```

```
>>> t = c.cut(c.index(b'1'), c.index(b'3'))  # cuts an inner slice
>>> t.to_blocks()
[[1, b'1x2x']]
>>> c.to_blocks()
[[0, b'M'], [5, b'3x4']]
>>> t.bound_span  # address bounds of the slice (automatically activated)
(1, 5)
```

```
>>> k = bytesparse.from_blocks([[4, b'ABC'], [9, b'xy']], start=2, endex=15)  # bounded
>>> str(k)  # shows summary
"<2, [[4, b'ABC'], [9, b'xy']], 15>"
>>> k.bound_span  # defined at creation
(2, 15)
>>> k.span  # superseded by bounds
(2, 15)
>>> k.content_span  # actual content span (min/max addresses)
(4, 11)
>>> len(k)  # superseded by bounds
13
>>> k.content_size  # actual content size
5
```

```
>>> k.flood(pattern=b'.')  # floods between span
>>> k.to_blocks()
[[2, b'..ABC..xy....']]
```

1.5 Background

This library started as a spin-off of `hexrec.blocks.Memory`. That was based on a simple Python implementation using immutable objects (i.e. `tuple` and `bytes`). While good enough to handle common hexadecimal files, it was totally unsuited for dynamic/interactive environments, such as emulators, IDEs, data editors, and so on. Instead, `bytesparse` should be more flexible and faster, hopefully suitable for generic usage.

While developing the Python implementation, why not also jump on the Cython bandwagon, which permits even faster algorithms? Moreover, Cython itself is an interesting intermediate language, which brings to the speed of C, whilst being close enough to Python for the like.

Too bad, one great downside is that debugging Cython-compiled code is quite an hassle – that is why I debugged it in a crude way I cannot even mention, and the reason why there must be dozens of bugs hidden around there, despite the test suite :-) Moreover, the Cython implementation is still experimental, with some features yet to be polished (e.g. reference counting).

1.6 Documentation

For the full documentation, please refer to:

https://bytesparse.readthedocs.io/

1.7 Installation

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

```
$ pip install bytesparse
```

From the source code root directory:

```
$ pip install .
```

1.8 Development

To run the all the tests:

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

CHAPTER

TWO

INSTALLATION

2.1 From *PyPI*

At the command line:

```
$ pip install bytesparse
```

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

bytesparse	Utilities for sparse blocks of bytes.
bytesparse.base	Common stuff, shared across modules.
bytesparse.inplace	In-place implementation.
bytesparse.io	Streaming utilities.

3.1 bytesparse

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
```

3.2 bytesparse.base

Common stuff, shared across modules.

Attributes

STR_MAX_CONTENT_SIZE	Maximum memory content size for string representation.
HUMAN_ASCII	Mapping from byte to human-readable ASCII characters.

3.2.1 STR_MAX_CONTENT_SIZE

bytesparse.base.**STR_MAX_CONTENT_SIZE: int = 1000**

Maximum memory content size for string representation.

3.2.2 HUMAN_ASCII

```
bytesparse.base.HUMAN_ASCII = '................................
!"#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\
]^_`abcdefghijklmnopqrstuvwxyz{|}~................................................
................................................................ ><'
```

Mapping from byte to human-readable ASCII characters.

Classes

ImmutableMemory	Immutable virtual memory.
MutableBytesparse	Wrapper for more *bytearray* compatibility.
MutableMemory	Mutable virtual memory.

3.2.3 ImmutableMemory

class bytesparse.base.**ImmutableMemory**(*start=None*, *endex=None*)

Immutable virtual memory.

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

Being immutable, only getters and queries can be performed against instances of this class.

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

Parameters

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

Examples

```
>>> from bytesparse import Memory
```

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

```
>>> memory = Memory(start=3, endex=10)
>>> memory.bound_span
(3, 10)
>>> memory.write(0, b'Hello, World!')
```

(continues on next page)

(continued from previous page)

```
>>> memory.to_blocks()
[[3, b'lo, Wor']]
```

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

Method Groups

- **Addressing** – *__len__()* *_block_index_at()* *_block_index_endex()* *_block_index_start()* *block_span()* *bound()* *bound_endex* *bound_span* *bound_start* *content_endex* *content_endin* *content_parts* *content_size* *content_span* *content_start* *count* *endex* *endin* *span* *start*
- **Comparison** – *__bool__()* *__eq__()* *contiguous* *validate()*
- **Creation** – *__copy__()* *__deepcopy__()* *__init__()* *copy()* *extract()* *from_blocks()* *from_bytes()* *from_items()* *from_memory()* *from_values()* *fromhex()*
- **Export** – *__bytes__()* *__repr__()* *__str__()* *hex()* *hexdump()* *read()* *readinto()* *to_blocks()* *to_bytes()* *view()*
- **Iteration** – *__iter__()* *__reversed__()* *blocks()* *chop()* *content_blocks()* *content_items()* *content_keys()* *content_values()* *equal_span()* *gaps()* *intervals()* *items()* *keys()* *rvalues()*
- **Search** – *__contains__()* *__getitem__()* *count()* *equal_span()* *find()* *index()* *rfind()* *rindex()*
- **Vector** – *__add__()* *__mul__()* *__or__()* *collapse_blocks()* *extract()* *get()* *peek()*

Attributes

bound_endex	Bounds exclusive end address.
bound_span	Bounds span addresses.
bound_start	Bounds start address.
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.

Methods

__init__	
block_span	Span of block data.
blocks	Iterates over blocks.
bound	Bounds addresses.
chop	Iterates over chopped blocks.
collapse_blocks	Collapses a generic sequence of blocks.
content_blocks	Iterates over blocks.
content_items	Iterates over content address and value pairs.
content_keys	Iterates over content addresses.
content_values	Iterates over content values.
copy	Creates a deep copy.
count	Counts items.
equal_span	Span of homogeneous data.
extract	Selects items from a range.
find	Index of an item.
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.
hexdump	Textual hex dump.
index	Index of an item.
intervals	Iterates over block intervals.
items	Iterates over address and value pairs.
keys	Iterates over addresses.
peek	Gets the item at an address.
read	Reads data.
readinto	Reads data into a pre-allocated buffer.
rfind	Index of an item, reversed search.
rindex	Index of an item, reversed search.
rvalues	Iterates over values, reversed order.
to_blocks	Exports into blocks.
to_bytes	Exports into bytes.
validate	Validates internal structure.
values	Iterates over values.
view	Creates a view over a range.

abstract __add__(*value*)

Concatenates items.

Equivalent to `self.copy() += items` of a *MutableMemory*.

See also:

MutableMemory.__iadd__()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC')
>>> memory2 = memory1 + b'xyz'
>>> memory2.to_blocks()
[[0, b'ABCxyz']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.content_endex
4
>>> memory3 = memory1 + memory2
>>> memory3.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

abstract __bool__()

Has any items.

Returns

bool – Has any items.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> bool(memory)
False
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> bool(memory)
True
```

abstract __bytes__()

Creates a bytes clone.

Returns

bytes – Cloned data.

Raises

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

Examples

```
>>> from bytesparse import Memory
```

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

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

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> bytes(memory)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> bytes(memory)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

classmethod __class_getitem__()

Represent a PEP 585 generic type

E.g. for t = list[int], t.__origin__ is list and t.__args__ is (int,).

abstract __contains__(*item*)

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 bytesparse 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
```

(continues on next page)

(continued from previous page)

```
>>> b'$' in memory
False
```

abstract __copy__()

Creates a shallow copy.

Returns
ImmutableMemory – Shallow copy.

abstract __deepcopy__()

Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

abstract __eq__(*other*)

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 bytesparse 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
```

abstract __getitem__(*key*)

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 bytesparse 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'$'], [8, b'xyz']])
>>> memory[9]  # -> ord('y') = 121
121
>>> memory[:3]._blocks
[[1, b'AB']]
>>> memory[3:10]._blocks
[[3, b'CD'], [6, b'$'], [8, b'xy']]
>>> bytes(memory[3:10:b'.'])
b'CD.$.xy'
>>> memory[memory.endex]
None
>>> bytes(memory[3:10:3])
b'C$y'
>>> memory[3:10:2]._blocks
[[3, b'C'], [6, b'y']]
>>> bytes(memory[3:10:2])
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

__hash__ = None

abstract __init__(*start=None*, *endex=None*)

abstract __iter__()

Iterates over values.

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

Yields

int – Value as byte integer, or None.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> list(memory)
[65, 66, 67, None, 120, 121, 122]
```

abstract __len__()

Actual length.

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

Returns

int – Memory length.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> len(memory)
0
```

```
>>> memory = Memory(start=3, endex=10)
>>> len(memory)
7
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [9, b'xyz']])
>>> len(memory)
11
```

```
>>> memory = Memory.from_blocks([[3, b'ABC'], [9, b'xyz']], start=1, endex=15)
>>> len(memory)
14
```

abstract __mul__(*times*)

Concatenates a repeated copy.

Equivalent to `self.copy() *= items` of a *MutableMemory*.

See also:

MutableMemory.__imul__()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory2 = memory1 * 3
>>> memory2.to_blocks()
[[0, b'ABCABCABC']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = memory1 * 3
>>> memory2.to_blocks()
[[1, b'ABCABCABC']]
```

abstract __or__(*value*)

Merges memories.

Equivalent to `self.copy() |= items` of a *MutableMemory*.

See also:

MutableMemory.__ior__()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory3 = memory1 | memory2
>>> memory3.to_blocks()
[[1, b'ABC'], [5, b'xyz']]
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory2 = memory1 | b'xyz'
>>> memory2.to_blocks()
[[0, b'xyzBC']]
```

abstract __repr__()

Return repr(self).

abstract __reversed__()

Iterates over values, reversed order.

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

Yields

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> list(memory)
[65, 66, 67, None, 120, 121, 122]
>>> list(reversed(memory))
[122, 121, 120, None, 67, 66, 65]
```

abstract __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 bytesparse 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']]>
```

classmethod __subclasshook__(*C*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__

list of weak references to the object (if defined)

abstract _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 bytesparse 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]
```

abstract _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 bytesparse 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]
```

abstract _block_index_start(*address*)

Locates the first block inside 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 bytesparse 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]
```

abstract 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 bytesparse 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)
```
~~~

abstract blocks(*start=None*, *endex=None*)

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 bytesparse 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)]
[]
```

abstract 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 bounds limits, *bound_start* or *bound_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 bytesparse 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)
```

abstract property bound_endex: int | None

Bounds exclusive end address.

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_endex = 10
>>> memory.to_blocks()
[[5, b'Hello']]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, endex=10)
>>> memory.to_blocks()
[[5, b'Hello']]
```

Type
int

abstract property bound_span: Tuple[int | None, int | None]

Bounds span addresses.

A `tuple` holding *bound_start* and *bound_endex*.

Notes

Assigning `None` to *MutableMemory.bound_span* sets both *bound_start* and *bound_endex* to `None` (equivalent to `(None, None)`).

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_span = (7, 13)
>>> memory.to_blocks()
[[7, b'llo, W']]
>>> memory.bound_span = None
>>> memory.bound_span
(None, None)
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=7, endex=13)
>>> memory.to_blocks()
[[7, b'llo, W']]
```

Type
tuple of int

abstract property bound_start: int | None

Bounds start address.

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_start = 10
>>> memory.to_blocks()
[[10, b', World!']]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=10)
>>> memory.to_blocks()
[[10, b', World!']]
```

Type
int

abstract chop(*width*, *start=None*, *endex=None*, *align=False*)

Iterates over chopped blocks.

The provided range is split into sub-ranges of a fixed width. For each sub-range, it yields views of the contained block chunks.

Parameters

- **width** (*int*) – Sub-range width.
- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.
- **align** (*bool*) – Sub-ranges are aligned to *width*.

Examples

```
>>> from bytesparse import Memory
```

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

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> chopping = memory.chop(2, align=False)
>>> [(address, bytes(view)) for address, view in chopping]
[(1, b'AB'), (3, b'C'), (6, b'xy'), (8, b'z')]
>>> chopping = memory.chop(2, align=True)
>>> [(address, bytes(view)) for address, view in chopping]
[(1, b'A'), (2, b'BC'), (6, b'xy'), (8, b'z')]
```

abstract classmethod 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

```
>>> from bytesparse import Memory
```

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'$'],
...     [6, b'xyz'],
... ]
>>> Memory.collapse_blocks(blocks)
[[0, b'$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'$'],
... ]
```

(continues on next page)
~~~

(continued from previous page)

```
>>> Memory.collapse_blocks(blocks)
[[0, b'0$2'], [4, b'ABxyz']]
```

abstract content_blocks(*block_index_start=None*, *block_index_endex=None*, *block_index_step=None*)

Iterates over blocks.

Iterates over data blocks within a block index range.

Parameters

- **block_index_start** (*int*) – Inclusive block start index. A negative index is referred to *content_parts*. If None, `0` is considered.
- **block_index_endex** (*int*) – Exclusive block end index. A negative index is referred to *content_parts*. If None, *content_parts* is considered.
- **block_index_step** (*int*) – Block index step, which can be negative. If None, 1 is considered.

Yields

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

See also:

content_parts

Examples

```
>>> from bytesparse 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.content_blocks()]
[[1, b'AB'], [5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(1, 2)]
[[5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(3, 5)]
[]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_start=-2)]
[[5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_endex=-1)]
[[1, b'AB'], [5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_step=2)]
[[1, b'AB'], [7, b'123']]
```

abstract property content_endex: int

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 bounds, *start* is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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

abstract property content_endin: int

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 bounds, *start* minus one is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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

abstract content_items(*start=None*, *endex=None*)

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 bytesparse 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))
{}
```

abstract content_keys(*start=None*, *endex=None*)

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 bytesparse 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))
[]
```

abstract property content_parts: int

Number of blocks.

Returns

int – The number of blocks.

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().content_parts
∅
```

~~~

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
```

abstract property content_size: int

Actual content size.

Returns

int – The sum of all block lengths.

Examples

```
>>> from bytesparse import Memory
```

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

~~~

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
```

abstract property content_span: Tuple[int, int]

Memory content address span.

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

Examples

```
>>> from bytesparse 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

abstract property content_start: int

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 bounds, 0 is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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

abstract content_values(*start=None*, *endex=None*)

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 bytesparse 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))
[]
```

abstract property contiguous: bool

Contains contiguous data.

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

If bounds are defined, there must be no empty space also towards them.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory.contiguous
False
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.contiguous
False
```

Type
bool

abstract copy()

Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory()
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(None, None)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory(start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory2 = memory1.copy()
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
>>> memory2.bound_span = (2, 19)
>>> memory1 == memory2
True
```

```
>>> memory1 = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory2 = memory1.copy()
[[5, b'ABC'], [9, b'xyz']]
>>> memory1 == memory2
True
```

abstract count(*item*, *start=None*, *endex=None*)

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 bytesparse 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
```

abstract property endex: int

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 *bound_endex* not `None`, that is returned.

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

Examples

```
>>> from bytesparse 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

abstract property endin: int

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 *bound_endex* not `None`, that minus one is returned.

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

Examples

```
>>> from bytesparse 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

abstract 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 bytesparse 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)
```

abstract extract(*start=None*, *endex=None*, *pattern=None*, *step=None*, *bound=True*)

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

abstract find(*item*, *start=None*, *endex=None*)

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.

> **Warning:** If the memory allows negative addresses, *index()* is more appropriate, because it raises `ValueError` if the item is not found.

See also:

index()

abstract classmethod from_blocks(*blocks*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.

- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse 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  # noqa
>>> blocks = hr.load_blocks('records.hex')
>>> memory = Memory.from_blocks(Memory.collapse_blocks(blocks))
>>> memory
    ...
```

abstract classmethod from_bytes(*data*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
~~~

- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse 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']]
```

abstract classmethod from_items(*items*, *offset=0*, *start=None*, *endex=None*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 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']]
```

abstract classmethod from_memory(*memory*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse import Memory
```

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

~~~

```
>>> memory1 = Memory.from_bytes(b'ABC', 10)
>>> memory2 = Memory.from_memory(memory1, -3)
>>> memory2._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._blocks, memory2._blocks))
True
```

abstract classmethod from_values(*values*, *offset=0*, *start=None*, *endex=None*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 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']]
```

abstract classmethod fromhex(*string*)

Creates a virtual memory from an hexadecimal string.

Parameters

string (*str*) – Hexadecimal string.

Returns

ImmutableMemory – The resulting memory object.

Examples

```
>>> from bytesparse import Memory
```

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

~~~

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

abstract gaps(*start=None*, *endex=None*)

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 bytesparse 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)]
```

abstract get(*address*, *default=None*)

Gets the item at an address.

Returns

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

Examples

```
>>> from bytesparse 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
```

(continues on next page)

(continued from previous page)

```
123
>>> memory.get(7, 123)  # -> empty -> default = 123
123
>>> memory.get(11, 123)  # -> empty -> default = 123
123
```

abstract 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 bytesparse 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
```

abstract hexdump(*start=None*, *endex=None*, *columns=16*, *addrfmt='{:08X} '*, *bytefmt=' {:02X}'*, *headfmt=None*, *charmap='..... !"#$%&\\'()*+, -./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\\\]^_`abcdefghijklmnopqrstuvwxyz{|}~..><'*, *emptystr=' --'*, *beforestr=' >>'*, *afterstr=' <<'*, *charsep=' |'*, *charend='|'*, *stream=Ellipsis*)

Textual hex dump.

This function generates a hex dump of the bytes within the specified range.

If *stream* is not `None`, the hex dump is written on it, otherwise it is returned as a `str`.

The default output is similar to that of `hexdump` or `xxd` commands, with some degree of tweaking. In case more customized formatting is desired, a dedicated custom function can be written by carefully looping over *values()*.

Parameters
~~~

- **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.
- **columns** (*int*) – Number of byte columns per row.
- **addrfmt** (*str*) – Address formatting string.
- **bytefmt** (*str*) – Byte formatting string.
- **headfmt** (*str*) – Header offset formatting string. If Ellipsis, it applies that of *bytefmt*. If None, no header row is generated.
- **charmap** (*mapping*) – Mapping to convert a byte integer into a string character. If None, no character data are appended to each row.

  The table is structured this way:

  – The initial 256 bytes map actual byte values.

  – Index 0x100 represents an empty byte (None).

  – Index 0x101 represents a byte before *start*.

  – Index 0x102 represents a byte after *endex*.
- **emptystr** (*str*) – Placeholder for an empty byte (None value).
- **beforestr** (*str*) – Placeholder for a byte before *bound_start*.
- **afterstr** (*str*) – Placeholder for a byte after *bound_endex*.
- **charsep** (*str*) – Separator between byte data and character data.
- **charend** (*str*) – Separator after character data.
- **stream** (*IO stream*) – Stream to write text onto. If Ellipsis, it uses sys.stdout. If not None, the function returns None.

Returns

str – Textual hex dump, if *stream* is None.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']], offset=0xDA7A)
>>> memory.hexdump()
0000DA7B  41 42 43 -- -- 78 79 7A -- -- -- -- -- -- -- --  |ABC  xyz        |
>>> memory.hexdump(stream=None)
'0000DA7B  41 42 43 -- -- 78 79 7A -- -- -- -- -- -- -- --  |ABC  xyz        |'
>>> memory.hexdump(start=0xDA7A, charmap=None)
0000DA7A  -- 41 42 43 -- -- 78 79 7A -- -- -- -- -- -- --
>>> memory.hexdump(start=0xDA7A)
0000DA7A  -- 41 42 43 -- -- 78 79 7A -- -- -- -- -- -- --  | ABC  xyz       |
>>> memory.hexdump(start=0xDA70)
0000DA70  -- -- -- -- -- -- -- -- -- -- -- 41 42 43 -- --  |           ABC  |
0000DA80  78 79 7A -- -- -- -- -- -- -- -- -- -- -- -- --  |xyz             |
>>> memory.bound_span = (0xDA78, 0xDA88)
>>> memory.hexdump(start=0xDA70)
0000DA70  >> >> >> >> >> >> >> >> -- -- -- 41 42 43 -- --  |>>>>>>>>   ABC  |
```

(continues on next page)

(continued from previous page)

```
0000DA80  78 79 7A -- -- -- -- -- << << << << << << << <<  |xyz     <<<<<<<<|
>>> memory.hexdump(start=0xDA70, headfmt=...)
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000DA70  >> >> >> >> >> >> >> >> -- -- -- 41 42 43 -- --  |>>>>>>>>   ABC  |
0000DA80  78 79 7A -- -- -- -- -- << << << << << << << <<  |xyz     <<<<<<<<|
>>> memory.hexdump(start=0xDA78, endex=0xDA84, columns=4)
0000DA78  -- -- -- 41  |   A|
0000DA7C  42 43 -- --  |BC  |
0000DA80  78 79 7A --  |xyz |
```

abstract index(*item*, *start=None*, *endex=None*)

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.

See also:

find()

abstract intervals(*start=None*, *endex=None*)

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 bytesparse 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))
[]
```

abstract items(*start=None*, *endex=None*, *pattern=None*)

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 bytesparse 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)]
```
~~~

abstract keys(*start=None*, *endex=None*)

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 bytesparse 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]
```

abstract peek(*address*)

Gets the item at an address.

Returns

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

Examples

```
>>> from bytesparse 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
```

abstract read(*address*, *size*)

Reads data.

Reads a chunk of data from an address, with a given size. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.
- **size** (*int*) – Chunk size.

Returns

memoryview – A view over the addressed chunk.

Raises

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

Examples

```
>>> from bytesparse 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.read(2, 3))
b'BCD'
>>> bytes(memory.read(9, 1))
b'y'
```

(continues on next page)

(continued from previous page)

```
>>> memory.read(4, 3)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.read(0, 6)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

abstract readinto(*address*, *buffer*)

Reads data into a pre-allocated buffer.

Provided a pre-allocated writable buffer (*e.g.* a `bytearray` or a `memoryview` slice of it), this method reads a chunk of data from an address, with the size of the target buffer. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.
- **buffer** (*writable*) – Pre-allocated buffer to fill with data.

Returns

int – Number of bytes read.

Raises

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

Examples

```
>>> from bytesparse 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']])
>>> buffer = bytearray(3)
>>> memory.readinto(2, buffer)
3
>>> buffer
bytearray(b'BCD')
>>> view = memoryview(buffer)
>>> memory.readinto(9, view[1:2])
1
>>> buffer
bytearray(b'ByD')
>>> memory.readinto(4, buffer)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.readinto(0, bytearray(6))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
ValueError: non-contiguous data within range
```

abstract rfind(*item*, *start=None*, *endex=None*)

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.

> **Warning:** If the memory allows negative addresses, *rindex()* is more appropriate, because it raises `ValueError` if the item is not found.

See also:

rindex()

abstract rindex(*item*, *start=None*, *endex=None*)

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.

> **Warning:** If the memory allows negative addresses, *index()* is more appropriate, because it raises `ValueError` if the item is not found.

See also:

rfind()

abstract rvalues(*start=None*, *endex=None*, *pattern=None*)

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

abstract property span: Tuple[int, int]

Memory address span.

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

Examples

```
>>> from bytesparse 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

abstract property start: int

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 *bound_start* not `None`, that is returned.

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

Examples

```
>>> from bytesparse 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

abstract to_blocks(*start=None*, *endex=None*)

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 bytesparse 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)]
[]
```

abstract to_bytes(*start=None*, *endex=None*)

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 bytesparse 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'
```

abstract 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).

abstract values(*start=None*, *endex=None*, *pattern=None*)

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 bytesparse 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]
```

abstract view(*start=None*, *endex=None*)

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 bytesparse 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
```

3.2.4 MutableBytesparse

class bytesparse.base.**MutableBytesparse**(**args*, *start=None*, *endex=None*)

Wrapper for more *bytearray* compatibility.

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

For instantiation, please refer to *MutableBytesparse.__init__()*.

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

See also:

ImmutableMemory *MutableMemory*

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 deleted. 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 deleted.

Examples

```
>>> from bytesparse import bytesparse
```

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

```
>>> memory = bytesparse(start=3, endex=10)
>>> memory.bound_span
(3, 10)
>>> memory.write(0, b'Hello, World!')
>>> memory.to_blocks()
[[3, b'lo, Wor']]
```

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

```
>>> memory = bytesparse(b'Hello, World!')
>>> memory.to_blocks()
[[0, b'Hello, World!']]
```

```
>>> memory = bytesparse(3)
>>> memory.to_blocks()
[[0, b'\x00\x00\x00']]
```

```
>>> memory = bytesparse([65, 66, 67])
>>> memory.to_blocks()
[[0, b'ABC']]
```

```
>>> memory = bytesparse('ASCII string', 'ascii')
>>> memory.to_blocks()
[[0, b'ASCII string']]
```

```
>>> memory = bytesparse('Non-ASCII: \u2204', 'ascii', 'backslashreplace')
>>> memory.to_blocks()
[[0, b'Non-ASCII: \\u2204']]
```

```
>>> memory = bytesparse('Non-ASCII: \u2204', 'ascii', 'xmlcharrefreplace')
>>> memory.to_blocks()
[[0, b'Non-ASCII: &#8708;']]
```

```
>>> memory = bytesparse('Non-ASCII: \u2204', 'ascii', 'replace')
>>> memory.to_blocks()
[[0, b'Non-ASCII: ?']]
```

```
>>> memory = bytesparse('Non-ASCII: \u2204', 'ascii', 'ignore')
>>> memory.to_blocks()
[[0, b'Non-ASCII: ']]
```

```
>>> memory = bytesparse('Non-ASCII: \u2204', 'ascii', 'strict')
Traceback (most recent call last):
    ...
UnicodeEncodeError: 'ascii' codec can't encode character '\u2204' in position 11:␣
↪ordinal not in range(128)
```

```
>>> memory = bytesparse('Missing encoding')
Traceback (most recent call last):
    ...
TypeError: string argument without an encoding
```

Method Groups

- **Creation** – *__init__()*
- **Addressing** – *_rectify_address()* *_rectify_span()*

Attributes

bound_endex	Bounds exclusive end address.
bound_span	Bounds span addresses.
bound_start	Bounds start address.
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.

Methods

__init__	
align	Floods blocks to align their boundaries.
align_backup	Backups an *align()* operation.
align_restore	Restores an *align()* operation.
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.
chop	Iterates over chopped blocks.
clear	Clears an address range.
clear_backup	Backups a *clear()* operation.
clear_restore	Restores a *clear()* operation.
collapse_blocks	Collapses a generic sequence of blocks.
content_blocks	Iterates over blocks.
content_items	Iterates over content address and value pairs.
content_keys	Iterates over content addresses.
content_values	Iterates over content values.
copy	Creates a deep 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.

continues on next page

Table 2 – continued from previous page

hex	Converts into an hexadecimal string.
hexdump	Textual hex dump.
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.
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.
popitem	Pops the last item.
popitem_backup	Backups a *popitem()* operation.
popitem_restore	Restores a *popitem()* operation.
read	Reads data.
readinto	Reads data into a pre-allocated buffer.
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.
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.

abstract __add__(*value*)

Concatenates items.

Equivalent to `self.copy() += items` of a *MutableMemory*.

See also:

MutableMemory.__iadd__()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC')
>>> memory2 = memory1 + b'xyz'
>>> memory2.to_blocks()
[[0, b'ABCxyz']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.content_endex
4
>>> memory3 = memory1 + memory2
>>> memory3.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

abstract __bool__()

Has any items.

Returns
bool – Has any items.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> bool(memory)
False
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> bool(memory)
True
```

abstract __bytes__()

Creates a bytes clone.

Returns
bytes – Cloned data.

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

Examples

```
>>> from bytesparse import Memory
```

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

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

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> bytes(memory)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> bytes(memory)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

classmethod __class_getitem__()

Represent a PEP 585 generic type

E.g. for t = list[int], t.__origin__ is list and t.__args__ is (int,).

abstract __contains__(*item*)

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 bytesparse 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
```

(continues on next page)

(continued from previous page)

```
>>> b'$' in memory
False
```

abstract __copy__()

Creates a shallow copy.

Returns
ImmutableMemory – Shallow copy.

abstract __deepcopy__()

Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

abstract __delitem__(*key*)

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 bytesparse 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()
```

(continues on next page)
~~~

(continued from previous page)

```
[[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']]
```

abstract __eq__(*other*)

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 bytesparse 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
```

abstract __getitem__(*key*)

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 bytesparse 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'$'], [8, b'xyz']])
>>> memory[9]  # -> ord('y') = 121
121
>>> memory[:3]._blocks
[[1, b'AB']]
>>> memory[3:10]._blocks
[[3, b'CD'], [6, b'$'], [8, b'xy']]
>>> bytes(memory[3:10:b'.'])
b'CD.$.xy'
>>> memory[memory.endex]
None
>>> bytes(memory[3:10:3])
b'C$y'
>>> memory[3:10:2]._blocks
[[3, b'C'], [6, b'y']]
>>> bytes(memory[3:10:2])
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

__hash__ = None

abstract __iadd__(*value*)

Concatenates items.

Equivalent to `self.extend(value)`.

See also:

extend()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'ABC')
>>> memory += b'xyz'
>>> memory.to_blocks()
[[0, b'ABCxyz']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.content_endex
4
>>> memory1 += memory2
>>> memory1.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

abstract __imul__(*times*)

Concatenates a repeated copy.

Equivalent to `self.extend(items)` repeated *times* times.

See also:

extend()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'ABC')
>>> memory *= 3
>>> memory.to_blocks()
[[0, b'ABCABCABC']]
```

```
>>> memory = Memory.from_blocks([[1, b'ABC']])
>>> memory *= 3
>>> memory.to_blocks()
[[1, b'ABCABCABC']]
```

abstract __init__(**args*, *start=None*, *endex=None*)

abstract __ior__(*value*)

Merges memories.

Equivalent to `self.write(0, value)`.

See also:

extend()

See also:

MutableMemory.__ior__()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1 |= memory2
>>> memory1.to_blocks()
[[1, b'ABC'], [5, b'xyz']]
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory1 |= b'xyz'
>>> memory2.to_blocks()
[[0, b'xyzBC']]
```

abstract __iter__()

Iterates over values.

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

Yields

int – Value as byte integer, or None.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> list(memory)
[65, 66, 67, None, 120, 121, 122]
```

abstract __len__()

Actual length.

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

Returns

int – Memory length.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> len(memory)
0
```

```
>>> memory = Memory(start=3, endex=10)
>>> len(memory)
7
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [9, b'xyz']])
>>> len(memory)
11
```

```
>>> memory = Memory.from_blocks([[3, b'ABC'], [9, b'xyz']], start=1, endex=15)
>>> len(memory)
14
```

abstract __mul__(*times*)

Concatenates a repeated copy.

Equivalent to `self.copy() *= items` of a *MutableMemory*.

See also:

MutableMemory.__imul__()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory2 = memory1 * 3
>>> memory2.to_blocks()
[[0, b'ABCABCABC']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = memory1 * 3
>>> memory2.to_blocks()
[[1, b'ABCABCABC']]
```

abstract __or__(*value*)

Merges memories.

Equivalent to `self.copy() |= items` of a *MutableMemory*.

See also:

MutableMemory.__ior__()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory3 = memory1 | memory2
>>> memory3.to_blocks()
[[1, b'ABC'], [5, b'xyz']]
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory2 = memory1 | b'xyz'
>>> memory2.to_blocks()
[[0, b'xyzBC']]
```

abstract __repr__()

Return repr(self).

abstract __reversed__()

Iterates over values, reversed order.

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

Yields

int – Value as byte integer, or None.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> list(memory)
[65, 66, 67, None, 120, 121, 122]
>>> list(reversed(memory))
[122, 121, 120, None, 67, 66, 65]
```

abstract __setitem__(*key*, *value*)

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 bytesparse 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()
```

(continues on next page)

(continued from previous page)

```
[[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']]
```

abstract __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 bytesparse 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']]>
```

classmethod __subclasshook__(*C*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__

list of weak references to the object (if defined)

abstract _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 bytesparse 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]
```

abstract _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 bytesparse 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]
```

abstract _block_index_start(*address*)

Locates the first block inside 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 bytesparse 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]
```

abstract _prebound_endex(*start_min*, *size*)

Bounds final data.

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

Parameters

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

See also:

_prebound_endex_backup()

abstract _prebound_endex_backup(*start_min*, *size*)

Backups a *_prebound_endex()* operation.

Parameters

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

Returns

ImmutableMemory – Backup memory region.

See also:

_prebound_endex()

abstract _prebound_start(*endex_max*, *size*)

Bounds initial data.

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

Parameters

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

See also:

_prebound_start_backup()

abstract _prebound_start_backup(*endex_max*, *size*)

Backups a *_prebound_start()* operation.

Parameters

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

Returns

ImmutableMemory – Backup memory region.

See also:

_prebound_start()

abstract _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.

abstract _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.

abstract align(*modulo*, *start=None*, *endex=None*, *pattern=0*)

Floods blocks to align their boundaries.

Parameters

- **modulo** (*int*) – Alignment modulo.
- **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:

align_backup() align_restore() flood()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11	12
	[A	B	C]			[x	y	z]				
[0	A	B	C	0	1	x	y	z	1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.align(4, pattern=b'0123')
>>> memory.to_blocks()
[[0, b'0ABC01xyz123']]
```

~~~
~~~

0	1	2	3	4	5	6	7	8	9	10	11	12
	[A	B	C]				[0	1	2	3]		
[x	A	B	C]			[x	0	1	2	3	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [7, b'0123']])
>>> memory.align(2, pattern=b'xyz')
>>> memory.to_blocks()
[[0, b'xABC'], [6, b'x0123z']]
```

~~~

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']])
>>> memory.align(2, start=3, endex=7, pattern=b'.@')
>>> memory.to_blocks()
[[1, b'ABC'], [6, b'xyz']]
```

abstract align_backup(*modulo*, *start=None*, *endex=None*)

Backups an *align()* operation.

Parameters

- **modulo** (*int*) – Alignment modulo.
- **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:

align() align_restore()

abstract align_restore(*gaps*)

Restores an *align()* operation.

Parameters

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

See also:

align() align_backup()

abstract 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 bytesparse import Memory
```

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

~~~

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

abstract append_backup()

Backups an *append()* operation.

Returns
None – Nothing.

See also:

append() append_restore()

abstract append_restore()

Restores an *append()* operation.

See also:

append() append_backup()

abstract 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 bytesparse 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)
```

abstract blocks(*start=None*, *endex=None*)

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 bytesparse 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)]
[]
```

abstract 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 bounds limits, *bound_start* or *bound_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 bytesparse 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)
```

(continues on next page)

(continued from previous page)

```
(2, 8)
>>> memory.bound(None, 6)
(1, 6)
```

abstract property bound_endex: int | None

Bounds exclusive end address.

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_endex = 10
>>> memory.to_blocks()
[[5, b'Hello']]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, endex=10)
>>> memory.to_blocks()
[[5, b'Hello']]
```

Type
int

abstract property bound_span: Tuple[int | None, int | None]

Bounds span addresses.

A `tuple` holding *bound_start* and *bound_endex*.

Notes

Assigning `None` to *MutableMemory.bound_span* sets both *bound_start* and *bound_endex* to `None` (equivalent to `(None, None)`).

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_span = (7, 13)
>>> memory.to_blocks()
[[7, b'llo, W']]
>>> memory.bound_span = None
>>> memory.bound_span
(None, None)
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=7, endex=13)
>>> memory.to_blocks()
[[7, b'llo, W']]
```

Type
tuple of int

abstract property bound_start: int | None

Bounds start address.

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_start = 10
>>> memory.to_blocks()
[[10, b', World!']]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=10)
>>> memory.to_blocks()
[[10, b', World!']]
```

Type
int

abstract chop(*width*, *start=None*, *endex=None*, *align=False*)

Iterates over chopped blocks.

The provided range is split into sub-ranges of a fixed width. For each sub-range, it yields views of the contained block chunks.

Parameters

- **width** (*int*) – Sub-range width.
- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.
- **align** (*bool*) – Sub-ranges are aligned to *width*.

Examples

```
>>> from bytesparse import Memory
```

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

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> chopping = memory.chop(2, align=False)
>>> [(address, bytes(view)) for address, view in chopping]
[(1, b'AB'), (3, b'C'), (6, b'xy'), (8, b'z')]
>>> chopping = memory.chop(2, align=True)
>>> [(address, bytes(view)) for address, view in chopping]
[(1, b'A'), (2, b'BC'), (6, b'xy'), (8, b'z')]
```

abstract clear(*start=None*, *endex=None*)

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 bytesparse 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']]
```

abstract clear_backup(*start=None*, *endex=None*)

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()

abstract clear_restore(*backup*)

Restores a *clear()* operation.

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

See also:

clear() clear_backup()

abstract classmethod 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

```
>>> from bytesparse import Memory
```

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'$'],
...     [6, b'xyz'],
... ]
>>> Memory.collapse_blocks(blocks)
[[0, b'$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'$'],
... ]
>>> Memory.collapse_blocks(blocks)
[[0, b'0$2'], [4, b'ABxyz']]
```

abstract content_blocks(*block_index_start=None*, *block_index_endex=None*, *block_index_step=None*)

Iterates over blocks.

Iterates over data blocks within a block index range.

Parameters

- **block_index_start** (*int*) – Inclusive block start index. A negative index is referred to *content_parts*. If None, `0` is considered.
- **block_index_endex** (*int*) – Exclusive block end index. A negative index is referred to *content_parts*. If None, *content_parts* is considered.
- **block_index_step** (*int*) – Block index step, which can be negative. If None, `1` is considered.

Yields

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

See also:

content_parts

Examples

```
>>> from bytesparse 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.content_blocks()]
[[1, b'AB'], [5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(1, 2)]
[[5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(3, 5)]
```

(continues on next page)

(continued from previous page)

```
[]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_start=-2)]
[[5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_endex=-1)]
[[1, b'AB'], [5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_step=2)]
[[1, b'AB'], [7, b'123']]
```

abstract property content_endex: int

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 bounds, *start* is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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

abstract property content_endin: int

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 bounds, *start* minus one is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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

abstract content_items(*start=None*, *endex=None*)

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 bytesparse 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))
{}
```

abstract content_keys(*start=None*, *endex=None*)

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 bytesparse 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]
```

(continues on next page)

(continued from previous page)

```
>>> list(memory.content_keys(3, 5))
[]
```

abstract property content_parts: int

Number of blocks.

Returns

int – The number of blocks.

Examples

```
>>> from bytesparse 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
```

abstract property content_size: int

Actual content size.

Returns

int – The sum of all block lengths.

Examples

```
>>> from bytesparse 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
```

abstract property content_span: Tuple[int, int]

Memory content address span.

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

Examples

```
>>> from bytesparse 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

abstract property content_start: int

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 bounds, 0 is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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

abstract content_values(*start=None*, *endex=None*)

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 bytesparse 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))
[]
```

abstract property contiguous: bool

Contains contiguous data.

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

If bounds are defined, there must be no empty space also towards them.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory.contiguous
False
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.contiguous
False
```

Type
bool

abstract copy()

Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory()
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(None, None)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory(start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory2 = memory1.copy()
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
>>> memory2.bound_span = (2, 19)
>>> memory1 == memory2
True
```

```
>>> memory1 = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory2 = memory1.copy()
[[5, b'ABC'], [9, b'xyz']]
>>> memory1 == memory2
True
```

abstract count(*item*, *start=None*, *endex=None*)

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 bytesparse 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
```

abstract crop(*start=None*, *endex=None*)

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 bytesparse 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']]
```

abstract crop_backup(*start=None*, *endex=None*)

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()

abstract 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()

abstract cut(*start=None*, *endex=None*, *bound=True*)

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 bounds 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.

Examples

```
>>> from bytesparse import Memory
```

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

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

abstract delete(*start=None*, *endex=None*)

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 bytesparse 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']]
```

abstract delete_backup(*start=None*, *endex=None*)

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()

abstract delete_restore(*backup*)

Restores a *delete()* operation.

Parameters

backup (*ImmutableMemory*) – Backup memory region

See also:

delete() delete_backup()

abstract property endex: int

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 *bound_endex* not `None`, that is returned.

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

Examples

```
>>> from bytesparse 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

abstract property endin: int

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 *bound_endex* not `None`, that minus one is returned.

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

Examples

```
>>> from bytesparse 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

abstract 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 bytesparse 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)
```

abstract extend(*items*, *offset=0*)

Concatenates items.

Appends *items* after *content_endex*. 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:

__iadd__() *extend_backup()* *extend_restore()*

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.extend(b'123')
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz123']]
```

~~~

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.extend(range(49, 52), offset=4)
```

(continues on next page)

(continued from previous page)

```
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz'], [12, b'123']]
```

~~~

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.extend(memory2)
>>> memory1.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

abstract extend_backup(*offset=0*)

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()

abstract extend_restore(*content_endex*)

Restores an *extend()* operation.

Parameters

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

See also:

extend() extend_backup()

abstract extract(*start=None*, *endex=None*, *pattern=None*, *step=None*, *bound=True*)

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

abstract fill(*start=None*, *endex=None*, *pattern=0*)

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 bytesparse 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']]
```

abstract fill_backup(*start=None*, *endex=None*)

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()*

abstract fill_restore(*backup*)

Restores a *fill()* operation.

Parameters

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

See also:

fill() *fill_backup()*

abstract find(*item*, *start=None*, *endex=None*)

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.

> **Warning:** If the memory allows negative addresses, *index()* is more appropriate, because it raises `ValueError` if the item is not found.

See also:

index()
~~~

abstract flood(*start=None*, *endex=None*, *pattern=0*)

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 bytesparse 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(start=3, endex=7, pattern=b'123')
>>> memory.to_blocks()
[[1, b'ABC23xyz']]
```

abstract flood_backup(*start=None*, *endex=None*)

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()
~~~

abstract flood_restore(*gaps*)

Restores a *flood()* operation.

Parameters

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

See also:

flood() flood_backup()

abstract classmethod from_blocks(*blocks*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse 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  # noqa
>>> blocks = hr.load_blocks('records.hex')
>>> memory = Memory.from_blocks(Memory.collapse_blocks(blocks))
>>> memory
    ...
```

abstract classmethod from_bytes(*data*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse 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']]
```
~~~

abstract classmethod from_items(*items*, *offset=0*, *start=None*, *endex=None*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 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']]
```

abstract classmethod from_memory(*memory*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse import Memory
```

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

~~~

```
>>> memory1 = Memory.from_bytes(b'ABC', 10)
>>> memory2 = Memory.from_memory(memory1, -3)
>>> memory2._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._blocks, memory2._blocks))
True
```

abstract classmethod from_values(*values*, *offset=0*, *start=None*, *endex=None*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 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']]
```

abstract classmethod fromhex(*string*)

Creates a virtual memory from an hexadecimal string.

Parameters
string (*str*) – Hexadecimal string.

Returns
ImmutableMemory – The resulting memory object.
~~~

Examples

```
>>> from bytesparse import Memory
```

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

~~~

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

abstract gaps(*start=None*, *endex=None*)

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 bytesparse 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)]
```

abstract get(*address*, *default=None*)

Gets the item at an address.
~~~

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

Examples

```
>>> from bytesparse 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
```

abstract 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 bytesparse 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
```

abstract hexdump(*start=None*, *endex=None*, *columns=16*, *addrfmt='{:08X} '*, *bytefmt=' {:02X}'*, *headfmt=None*, *charmap='..... !"#$%&\\'()*+, -./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\\\]^_`abcdefghijklmnopqrstuvwxyz{|}~..><'*, *emptystr=' --'*, *beforestr=' >>'*, *afterstr=' <<'*, *charsep=' |'*, *charend='|'*, *stream=Ellipsis*)

Textual hex dump.

This function generates a hex dump of the bytes within the specified range.

If *stream* is not `None`, the hex dump is written on it, otherwise it is returned as a `str`.

The default output is similar to that of `hexdump` or `xxd` commands, with some degree of tweaking. In case more customized formatting is desired, a dedicated custom function can be written by carefully looping over *values()*.

Parameters

- **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.
- **columns** (*int*) – Number of byte columns per row.
- **addrfmt** (*str*) – Address formatting string.
- **bytefmt** (*str*) – Byte formatting string.
- **headfmt** (*str*) – Header offset formatting string. If `Ellipsis`, it applies that of *bytefmt*. If `None`, no header row is generated.
- **charmap** (*mapping*) – Mapping to convert a byte integer into a string character. If `None`, no character data are appended to each row.

  The table is structured this way:

  – The initial 256 bytes map actual byte values.

  – Index `0x100` represents an empty byte (`None`).

  – Index `0x101` represents a byte before *start*.

  – Index `0x102` represents a byte after *endex*.
- **emptystr** (*str*) – Placeholder for an empty byte (`None` value).
~~~

- **beforestr** (*str*) – Placeholder for a byte before *bound_start*.
- **afterstr** (*str*) – Placeholder for a byte after *bound_endex*.
- **charsep** (*str*) – Separator between byte data and character data.
- **charend** (*str*) – Separator after character data.
- **stream** (*IO stream*) – Stream to write text onto. If `Ellipsis`, it uses `sys.stdout`. If not `None`, the function returns `None`.

Returns

str – Textual hex dump, if *stream* is `None`.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']], offset=0xDA7A)
>>> memory.hexdump()
0000DA7B  41 42 43 -- -- 78 79 7A -- -- -- -- -- -- -- --  |ABC  xyz        |
>>> memory.hexdump(stream=None)
'0000DA7B  41 42 43 -- -- 78 79 7A -- -- -- -- -- -- -- --  |ABC  xyz        |'
>>> memory.hexdump(start=0xDA7A, charmap=None)
0000DA7A  -- 41 42 43 -- -- 78 79 7A -- -- -- -- -- -- --
>>> memory.hexdump(start=0xDA7A)
0000DA7A  -- 41 42 43 -- -- 78 79 7A -- -- -- -- -- -- --  | ABC  xyz       |
>>> memory.hexdump(start=0xDA70)
0000DA70  -- -- -- -- -- -- -- -- -- -- -- 41 42 43 -- --  |           ABC  |
0000DA80  78 79 7A -- -- -- -- -- -- -- -- -- -- -- -- --  |xyz             |
>>> memory.bound_span = (0xDA78, 0xDA88)
>>> memory.hexdump(start=0xDA70)
0000DA70  >> >> >> >> >> >> >> >> -- -- -- 41 42 43 -- --  |>>>>>>>>   ABC  |
0000DA80  78 79 7A -- -- -- -- -- << << << << << << << <<  |xyz     <<<<<<<<|
>>> memory.hexdump(start=0xDA70, headfmt=...)
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000DA70  >> >> >> >> >> >> >> >> -- -- -- 41 42 43 -- --  |>>>>>>>>   ABC  |
0000DA80  78 79 7A -- -- -- -- -- << << << << << << << <<  |xyz     <<<<<<<<|
>>> memory.hexdump(start=0xDA78, endex=0xDA84, columns=4)
0000DA78  -- -- -- 41  |   A|
0000DA7C  42 43 -- --  |BC  |
0000DA80  78 79 7A --  |xyz |
```

abstract index(*item*, *start=None*, *endex=None*)

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.

See also:

find()

abstract 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 bytesparse 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'$']]
```

abstract 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()

abstract 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()

abstract intervals(*start=None*, *endex=None*)

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 bytesparse 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))
[]
```

abstract items(*start=None*, *endex=None*, *pattern=None*)

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 bytesparse 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)]
```

abstract keys(*start=None*, *endex=None*)

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 bytesparse 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]
```

abstract peek(*address*)

Gets the item at an address.

Returns

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

Examples

```
>>> from bytesparse 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
```

(continues on next page)
~~~

(continued from previous page)

```
>>> memory.peek(10)  # -> ord('z') = 122
122
>>> memory.peek(0)
None
>>> memory.peek(7)
None
>>> memory.peek(11)
None
```

abstract 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 bytesparse 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
```

abstract 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()

abstract 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()

abstract pop(*address=None*, *default=None*)

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 bytesparse 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
```

abstract pop_backup(*address=None*)

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()

abstract 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()

abstract popitem()

Pops the last item.

Returns

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

See also:

popitem_backup() popitem_restore()

Examples

```
>>> from bytesparse 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
```

abstract popitem_backup()

Backups a *popitem()* operation.

Returns

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

See also:

popitem() popitem_restore()

abstract popitem_restore(*address*, *item*)

Restores a *popitem()* operation.

Parameters

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

See also:

popitem() popitem_backup()

abstract read(*address*, *size*)

Reads data.

Reads a chunk of data from an address, with a given size. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.
- **size** (*int*) – Chunk size.

Returns

memoryview – A view over the addressed chunk.

Raises

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

Examples

```
>>> from bytesparse 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.read(2, 3))
b'BCD'
>>> bytes(memory.read(9, 1))
b'y'
>>> memory.read(4, 3)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.read(0, 6)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

abstract readinto(*address*, *buffer*)

Reads data into a pre-allocated buffer.

Provided a pre-allocated writable buffer (*e.g.* a bytearray or a memoryview slice of it), this method reads a chunk of data from an address, with the size of the target buffer. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.

- **buffer** (*writable*) – Pre-allocated buffer to fill with data.

Returns
int – Number of bytes read.

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

Examples

```
>>> from bytesparse 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']])
>>> buffer = bytearray(3)
>>> memory.readinto(2, buffer)
3
>>> buffer
bytearray(b'BCD')
>>> view = memoryview(buffer)
>>> memory.readinto(9, view[1:2])
1
>>> buffer
bytearray(b'ByD')
>>> memory.readinto(4, buffer)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.readinto(0, bytearray(6))
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

abstract remove(*item*, *start=None*, *endex=None*)

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 bytesparse 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
```

abstract remove_backup(*item*, *start=None*, *endex=None*)

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()

abstract remove_restore(*backup*)

Restores a *remove()* operation.

Parameters

backup (`Memory`) – Backup memory region.

See also:

remove() remove_backup()

abstract 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 bytesparse 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']]
```

abstract 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()

abstract 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()

abstract reverse()

Reverses the memory in-place.

Data is reversed within the memory *span*.

Examples

```
>>> from bytesparse 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']]
```

abstract rfind(*item*, *start=None*, *endex=None*)

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.

Warning: If the memory allows negative addresses, *rindex()* is more appropriate, because it raises `ValueError` if the item is not found.
~~~

See also:

rindex()

abstract rindex(*item*, *start=None*, *endex=None*)

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.

> **Warning:** If the memory allows negative addresses, *index()* is more appropriate, because it raises `ValueError` if the item is not found.

See also:

rfind()

abstract rvalues(*start=None*, *endex=None*, *pattern=None*)

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 bytesparse 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))
```

(continues on next page)

(continued from previous page)

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

abstract setdefault(*address*, *default=None*)

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 bytesparse 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
```

abstract 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()

abstract 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()

abstract shift(*offset*)

Shifts the items.

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

See also:

shift_backup() shift_restore()

Examples

```
>>> from bytesparse 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']]
```

abstract 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()

abstract 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()

abstract property span: Tuple[int, int]

Memory address span.

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

Examples

```
>>> from bytesparse 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

abstract property start: int

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 *bound_start* not `None`, that is returned.

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

Examples

```
>>> from bytesparse 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

abstract to_blocks(*start=None*, *endex=None*)

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 bytesparse 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)]
[]
```

abstract to_bytes(*start=None*, *endex=None*)

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 bytesparse 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'
```

abstract update(*data*, *clear=False*, ***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 bytesparse 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']]
```

abstract update_backup(*data*, *clear=False*, ***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()

abstract update_restore(*backups*)

Restores an *update()* operation.

Parameters

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

See also:

update() update_backup()

abstract 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).

abstract values(*start=None*, *endex=None*, *pattern=None*)

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 bytesparse 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]
```
~~~

abstract view(*start=None*, *endex=None*)

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 bytesparse 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
```

abstract write(*address*, *data*, *clear=False*)

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 bytesparse 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']]
```

abstract write_backup(*address*, *data*, *clear=False*)

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()

abstract write_restore(*backups*)

Restores a *write()* operation.

Parameters

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

See also:

write() write_backup()

3.2.5 MutableMemory

class bytesparse.base.**MutableMemory**(*start=None*, *endex=None*)

Mutable 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.

Being mutable, instances of this class can be updated dynamically. All the methods and attributes of an *ImmutableMemory* are available as well.

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

See also:

ImmutableMemory

Method Groups

- **Setting** – *__setitem__() bound_endex bound_span bound_start poke() reverse() setdefault() shift()*
- **Merging** – *__ior__() align() fill() flood() update()*
- **Extension** – *__iadd__() __imul__() append() extend() insert() reserve() write()*
- **Deletion** – *__delitem__() clear() crop() cut() delete() pop() popitem() remove()*
- **Backup** – *align_backup() append_backup() clear_backup() crop_backup() delete_backup() extend_backup() fill_backup() flood_backup() insert_backup() poke_backup() pop_backup() popitem_backup() remove_backup() reserve_backup() setdefault_backup() shift_backup() update_backup() write_backup()*
- **Restore** – *align_restore() append_restore() clear_restore() crop_restore() delete_restore() extend_restore() fill_restore() flood_restore() insert_restore() poke_restore() pop_restore() popitem_restore() remove_restore() reserve_restore() setdefault_restore() shift_restore() update_restore() write_restore()*
- **Internal** – *_prebound_endex() _prebound_endex_backup() _prebound_start() _prebound_start_backup()*

Parameters

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

Examples

```
>>> from bytesparse import Memory
```

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

```
>>> memory = Memory(start=3, endex=10)
>>> memory.bound_span
(3, 10)
>>> memory.write(0, b'Hello, World!')
>>> memory.to_blocks()
[[3, b'lo, Wor']]
```

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

Attributes

bound_endex	Bounds exclusive end address.
bound_span	Bounds span addresses.
bound_start	Bounds start address.
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.

Methods

__init__	
align	Floods blocks to align their boundaries.
align_backup	Backups an *align()* operation.
align_restore	Restores an *align()* operation.
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.
chop	Iterates over chopped blocks.
clear	Clears an address range.
clear_backup	Backups a *clear()* operation.
clear_restore	Restores a *clear()* operation.
collapse_blocks	Collapses a generic sequence of blocks.
content_blocks	Iterates over blocks.
content_items	Iterates over content address and value pairs.
content_keys	Iterates over content addresses.
content_values	Iterates over content values.
copy	Creates a deep 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.

continues on next page

Table 3 – continued from previous page

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.
hexdump	Textual hex dump.
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.
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.
popitem	Pops the last item.
popitem_backup	Backups a *popitem()* operation.
popitem_restore	Restores a *popitem()* operation.
read	Reads data.
readinto	Reads data into a pre-allocated buffer.
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.
rvalues	Iterates over values, reversed order.
setdefault	Defaults a value.
setdefault_backup	Backups a *setdefault()* operation.

continues on next page

Table 3 – continued from previous page

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.

abstract __add__(*value*)

Concatenates items.

Equivalent to `self.copy() += items` of a *MutableMemory*.

See also:

MutableMemory.__iadd__()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC')
>>> memory2 = memory1 + b'xyz'
>>> memory2.to_blocks()
[[0, b'ABCxyz']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.content_endex
4
>>> memory3 = memory1 + memory2
>>> memory3.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

abstract __bool__()

Has any items.

Returns

bool – Has any items.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> bool(memory)
False
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> bool(memory)
True
```

abstract __bytes__()

Creates a bytes clone.

Returns
bytes – Cloned data.

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

Examples

```
>>> from bytesparse import Memory
```

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

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

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> bytes(memory)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> bytes(memory)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

classmethod __class_getitem__()

Represent a PEP 585 generic type

E.g. for t = list[int], t.__origin__ is list and t.__args__ is (int,).

abstract __contains__(*item*)

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 bytesparse 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
```

abstract __copy__()
Creates a shallow copy.

Returns
ImmutableMemory – Shallow copy.

abstract __deepcopy__()
Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

abstract __delitem__(*key*)
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 bytesparse 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']]
```

abstract __eq__(*other*)

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 bytesparse 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
```

abstract __getitem__(*key*)

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 bytesparse 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'$'], [8, b'xyz']])
>>> memory[9]  # -> ord('y') = 121
121
>>> memory[:3]._blocks
[[1, b'AB']]
>>> memory[3:10]._blocks
[[3, b'CD'], [6, b'$'], [8, b'xy']]
>>> bytes(memory[3:10:b'.'])
b'CD.$.xy'
>>> memory[memory.endex]
None
>>> bytes(memory[3:10:3])
b'C$y'
>>> memory[3:10:2]._blocks
[[3, b'C'], [6, b'y']]
>>> bytes(memory[3:10:2])
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

__hash__ = None

abstract __iadd__(*value*)

Concatenates items.

Equivalent to `self.extend(value)`.

See also:

extend()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'ABC')
>>> memory += b'xyz'
>>> memory.to_blocks()
[[0, b'ABCxyz']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.content_endex
4
>>> memory1 += memory2
>>> memory1.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

abstract __imul__(*times*)

Concatenates a repeated copy.

Equivalent to `self.extend(items)` repeated *times* times.

See also:

extend()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'ABC')
>>> memory *= 3
>>> memory.to_blocks()
[[0, b'ABCABCABC']]
```

```
>>> memory = Memory.from_blocks([[1, b'ABC']])
>>> memory *= 3
>>> memory.to_blocks()
[[1, b'ABCABCABC']]
```

abstract __init__(*start=None*, *endex=None*)

abstract __ior__(*value*)

Merges memories.

Equivalent to `self.write(0, value)`.

See also:

extend()

See also:

MutableMemory.__ior__()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1 |= memory2
>>> memory1.to_blocks()
[[1, b'ABC'], [5, b'xyz']]
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory1 |= b'xyz'
>>> memory2.to_blocks()
[[0, b'xyzBC']]
```

abstract __iter__()

Iterates over values.

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

Yields

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> list(memory)
[65, 66, 67, None, 120, 121, 122]
```

abstract __len__()

Actual length.

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

Returns

int – Memory length.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> len(memory)
0
```

```
>>> memory = Memory(start=3, endex=10)
>>> len(memory)
7
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [9, b'xyz']])
>>> len(memory)
11
```

```
>>> memory = Memory.from_blocks([[3, b'ABC'], [9, b'xyz']], start=1, endex=15)
>>> len(memory)
14
```

abstract __mul__(*times*)

Concatenates a repeated copy.

Equivalent to `self.copy() *= items` of a *MutableMemory*.

See also:

MutableMemory.__imul__()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory2 = memory1 * 3
>>> memory2.to_blocks()
[[0, b'ABCABCABC']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = memory1 * 3
>>> memory2.to_blocks()
[[1, b'ABCABCABC']]
```

abstract __or__(*value*)

Merges memories.

Equivalent to `self.copy() |= items` of a *MutableMemory*.

See also:

MutableMemory.__ior__()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory3 = memory1 | memory2
>>> memory3.to_blocks()
[[1, b'ABC'], [5, b'xyz']]
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory2 = memory1 | b'xyz'
>>> memory2.to_blocks()
[[0, b'xyzBC']]
```

abstract __repr__()

Return repr(self).

abstract __reversed__()

Iterates over values, reversed order.

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

Yields

int – Value as byte integer, or None.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> list(memory)
[65, 66, 67, None, 120, 121, 122]
>>> list(reversed(memory))
[122, 121, 120, None, 67, 66, 65]
```

abstract __setitem__(*key*, *value*)

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 bytesparse 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']]
```

abstract __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 bytesparse 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']]>
```

classmethod __subclasshook__(*C*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__

list of weak references to the object (if defined)

abstract _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 bytesparse 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]
```

abstract _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 bytesparse 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]
```

abstract _block_index_start(*address*)

Locates the first block inside 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 bytesparse 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]
```

abstract _prebound_endex(*start_min*, *size*)

Bounds final data.

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

Parameters

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

See also:

_prebound_endex_backup()

abstract _prebound_endex_backup(*start_min*, *size*)

Backups a *_prebound_endex()* operation.

Parameters

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

Returns

ImmutableMemory – Backup memory region.

See also:

_prebound_endex()

abstract _prebound_start(*endex_max*, *size*)

Bounds initial data.

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

Parameters

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

See also:

_prebound_start_backup()

abstract _prebound_start_backup(*endex_max*, *size*)

Backups a *_prebound_start()* operation.

Parameters

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

Returns

ImmutableMemory – Backup memory region.

See also:

_prebound_start()

abstract align(*modulo*, *start=None*, *endex=None*, *pattern=0*)

Floods blocks to align their boundaries.

Parameters

- **modulo** (*int*) – Alignment modulo.
- **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:

align_backup() align_restore() flood()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11	12
	[A	B	C]			[x	y	z]				
[0	A	B	C	0	1	x	y	z	1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.align(4, pattern=b'0123')
>>> memory.to_blocks()
[[0, b'0ABC01xyz123']]
```

~~~

0	1	2	3	4	5	6	7	8	9	10	11	12
	[A	B	C]				[0	1	2	3]		
[x	A	B	C]			[x	0	1	2	3	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [7, b'0123']])
>>> memory.align(2, pattern=b'xyz')
>>> memory.to_blocks()
[[0, b'xABC'], [6, b'x0123z']]
```

~~~

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']])
>>> memory.align(2, start=3, endex=7, pattern=b'.@')
>>> memory.to_blocks()
[[1, b'ABC'], [6, b'xyz']]
```

abstract align_backup(*modulo*, *start=None*, *endex=None*)

Backups an *align()* operation.

Parameters

- **modulo** (*int*) – Alignment modulo.
- **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:

align() align_restore()

abstract align_restore(*gaps*)

Restores an *align()* operation.

Parameters

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

See also:

align() align_backup()

abstract 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 bytesparse import Memory
```

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

~~~

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

abstract append_backup()

Backups an *append()* operation.

Returns

None – Nothing.
~~~

See also:

append() append_restore()

abstract append_restore()

Restores an *append()* operation.

See also:

append() append_backup()

abstract 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 bytesparse 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)
```

abstract blocks(*start=None*, *endex=None*)

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 bytesparse 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)]
[]
```

abstract 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 bounds limits, *bound_start* or *bound_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 bytesparse 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)
```

abstract property bound_endex: int | None

Bounds exclusive end address.

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_endex = 10
>>> memory.to_blocks()
[[5, b'Hello']]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, endex=10)
>>> memory.to_blocks()
[[5, b'Hello']]
```
~~~

Type
int

abstract property bound_span: Tuple[int | None, int | None]

Bounds span addresses.

A `tuple` holding *bound_start* and *bound_endex*.

Notes

Assigning `None` to *MutableMemory.bound_span* sets both *bound_start* and *bound_endex* to `None` (equivalent to `(None, None)`).

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_span = (7, 13)
>>> memory.to_blocks()
[[7, b'llo, W']]
>>> memory.bound_span = None
>>> memory.bound_span
(None, None)
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=7, endex=13)
>>> memory.to_blocks()
[[7, b'llo, W']]
```

Type
tuple of int

abstract property bound_start: int | None

Bounds start address.

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_start = 10
>>> memory.to_blocks()
[[10, b', World!']]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=10)
>>> memory.to_blocks()
[[10, b', World!']]
```

Type
int

abstract chop(*width*, *start=None*, *endex=None*, *align=False*)

Iterates over chopped blocks.

The provided range is split into sub-ranges of a fixed width. For each sub-range, it yields views of the contained block chunks.

Parameters

- **width** (*int*) – Sub-range width.
- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.
- **align** (*bool*) – Sub-ranges are aligned to *width*.

Examples

```
>>> from bytesparse import Memory
```

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

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> chopping = memory.chop(2, align=False)
>>> [(address, bytes(view)) for address, view in chopping]
[(1, b'AB'), (3, b'C'), (6, b'xy'), (8, b'z')]
>>> chopping = memory.chop(2, align=True)
>>> [(address, bytes(view)) for address, view in chopping]
[(1, b'A'), (2, b'BC'), (6, b'xy'), (8, b'z')]
```

abstract clear(*start=None*, *endex=None*)

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 bytesparse 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']]
```

abstract clear_backup(*start=None*, *endex=None*)

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()*

abstract clear_restore(*backup*)

Restores a *clear()* operation.

Parameters

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

See also:

clear() *clear_backup()*

abstract classmethod 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

```
>>> from bytesparse import Memory
```

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'$'],
...     [6, b'xyz'],
... ]
>>> Memory.collapse_blocks(blocks)
[[0, b'$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'$'],
... ]
>>> Memory.collapse_blocks(blocks)
[[0, b'0$2'], [4, b'ABxyz']]
```

abstract content_blocks(*block_index_start=None*, *block_index_endex=None*, *block_index_step=None*)

Iterates over blocks.

Iterates over data blocks within a block index range.

Parameters

- **block_index_start** (*int*) – Inclusive block start index. A negative index is referred to *content_parts*. If None, `0` is considered.
- **block_index_endex** (*int*) – Exclusive block end index. A negative index is referred to *content_parts*. If None, *content_parts* is considered.
~~~

- **block_index_step** (*int*) – Block index step, which can be negative. If None, 1 is considered.

Yields

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

See also:

content_parts

Examples

```
>>> from bytesparse 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.content_blocks()]
[[1, b'AB'], [5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(1, 2)]
[[5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(3, 5)]
[]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_start=-2)]
[[5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_endex=-1)]
[[1, b'AB'], [5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_step=2)]
[[1, b'AB'], [7, b'123']]
```

abstract property content_endex: int

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 bounds, *start* is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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

abstract property content_endin: int

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 bounds, *start* minus one is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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

abstract content_items(*start=None*, *endex=None*)

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 bytesparse 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))
{}
```

abstract content_keys(*start=None*, *endex=None*)

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 bytesparse 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))
[]
```

abstract property content_parts: int

Number of blocks.

Returns
int – The number of blocks.

Examples

```
>>> from bytesparse 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
```

abstract property content_size: int

Actual content size.

Returns

int – The sum of all block lengths.

Examples

```
>>> from bytesparse 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
```

abstract property content_span: Tuple[int, int]

Memory content address span.

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

Examples

```
>>> from bytesparse 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

abstract property content_start: int

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 bounds, 0 is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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

abstract content_values(*start=None*, *endex=None*)

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 bytesparse 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))
[]
```

abstract property contiguous: bool

Contains contiguous data.

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

If bounds are defined, there must be no empty space also towards them.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory.contiguous
False
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.contiguous
False
```

Type
bool

abstract copy()

Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory()
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(None, None)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory(start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory2 = memory1.copy()
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
>>> memory2.bound_span = (2, 19)
>>> memory1 == memory2
True
```

```
>>> memory1 = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory2 = memory1.copy()
[[5, b'ABC'], [9, b'xyz']]
>>> memory1 == memory2
True
```

abstract count(*item*, *start=None*, *endex=None*)

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 bytesparse 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
```

abstract crop(*start=None*, *endex=None*)

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 bytesparse 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']]
```

abstract crop_backup(*start=None*, *endex=None*)

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()

abstract 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()

abstract cut(*start=None*, *endex=None*, *bound=True*)

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 bounds 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.

Examples

```
>>> from bytesparse import Memory
```

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

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

abstract delete(*start=None*, *endex=None*)

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 bytesparse 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']]
```

abstract delete_backup(*start=None*, *endex=None*)

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()

abstract delete_restore(*backup*)

Restores a *delete()* operation.

Parameters
backup (*ImmutableMemory*) – Backup memory region

See also:

delete() delete_backup()

abstract property endex: int

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 *bound_endex* not None, that is returned.

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

Examples

```
>>> from bytesparse 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

abstract property endin: int

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 *bound_endex* not `None`, that minus one is returned.

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

Examples

```
>>> from bytesparse 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

abstract 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 bytesparse 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)
```

abstract extend(*items*, *offset=0*)

Concatenates items.

Appends *items* after *content_endex*. 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:

__iadd__() extend_backup() extend_restore()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.extend(b'123')
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz123']]
```

~~~

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.extend(range(49, 52), offset=4)
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz'], [12, b'123']]
```

~~~

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.extend(memory2)
>>> memory1.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

abstract extend_backup(*offset=0*)

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()

abstract extend_restore(*content_endex*)

Restores an *extend()* operation.

Parameters

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

See also:

extend() extend_backup()

abstract extract(*start=None*, *endex=None*, *pattern=None*, *step=None*, *bound=True*)

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

abstract fill(*start=None*, *endex=None*, *pattern=0*)

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 bytesparse 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']]
```

abstract fill_backup(*start=None*, *endex=None*)

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()*

abstract fill_restore(*backup*)

Restores a *fill()* operation.

Parameters

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

See also:

fill() *fill_backup()*

abstract find(*item*, *start=None*, *endex=None*)

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.

> **Warning:** If the memory allows negative addresses, *index()* is more appropriate, because it raises `ValueError` if the item is not found.

See also:

index()
~~~

abstract flood(*start=None*, *endex=None*, *pattern=0*)

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 bytesparse 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(start=3, endex=7, pattern=b'123')
>>> memory.to_blocks()
[[1, b'ABC23xyz']]
```

abstract flood_backup(*start=None*, *endex=None*)

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()
~~~

abstract flood_restore(*gaps*)

Restores a *flood()* operation.

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

See also:

flood() flood_backup()

abstract classmethod from_blocks(*blocks*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse 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  # noqa
>>> blocks = hr.load_blocks('records.hex')
>>> memory = Memory.from_blocks(Memory.collapse_blocks(blocks))
>>> memory
    ...
```

abstract classmethod from_bytes(*data*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse 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']]
```
~~~

abstract classmethod from_items(*items*, *offset=0*, *start=None*, *endex=None*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 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']]
```

abstract classmethod from_memory(*memory*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse import Memory
```

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

~~~

```
>>> memory1 = Memory.from_bytes(b'ABC', 10)
>>> memory2 = Memory.from_memory(memory1, -3)
>>> memory2._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._blocks, memory2._blocks))
True
```

abstract classmethod from_values(*values*, *offset=0*, *start=None*, *endex=None*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 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']]
```

abstract classmethod fromhex(*string*)

Creates a virtual memory from an hexadecimal string.

Parameters
string (*str*) – Hexadecimal string.

Returns
ImmutableMemory – The resulting memory object.
~~~

Examples

```
>>> from bytesparse import Memory
```

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

~~~

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

abstract gaps(*start=None*, *endex=None*)

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 bytesparse 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)]
```

abstract get(*address*, *default=None*)

Gets the item at an address.
~~~

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

Examples

```
>>> from bytesparse 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
```

abstract 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 bytesparse 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
```

abstract hexdump(*start=None*, *endex=None*, *columns=16*, *addrfmt='{:08X} '*, *bytefmt=' {:02X}'*, *headfmt=None*, *charmap='.............................. !"#$%&\\'()*+, -./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\\\]^_`abcdefghijklmnopqrstuvwxyz{|}~.. ><'*, *emptystr=' --'*, *beforestr=' >>'*, *afterstr=' <<'*, *charsep=' |'*, *charend='|'*, *stream=Ellipsis*)

Textual hex dump.

This function generates a hex dump of the bytes within the specified range.

If *stream* is not `None`, the hex dump is written on it, otherwise it is returned as a `str`.

The default output is similar to that of `hexdump` or `xxd` commands, with some degree of tweaking. In case more customized formatting is desired, a dedicated custom function can be written by carefully looping over *values()*.

Parameters

- **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.
- **columns** (*int*) – Number of byte columns per row.
- **addrfmt** (*str*) – Address formatting string.
- **bytefmt** (*str*) – Byte formatting string.
- **headfmt** (*str*) – Header offset formatting string. If `Ellipsis`, it applies that of *bytefmt*. If `None`, no header row is generated.
- **charmap** (*mapping*) – Mapping to convert a byte integer into a string character. If `None`, no character data are appended to each row.

  The table is structured this way:

  – The initial 256 bytes map actual byte values.

  – Index `0x100` represents an empty byte (`None`).

  – Index `0x101` represents a byte before *start*.

  – Index `0x102` represents a byte after *endex*.
- **emptystr** (*str*) – Placeholder for an empty byte (`None` value).
~~~

- **beforestr** (*str*) – Placeholder for a byte before *bound_start*.
- **afterstr** (*str*) – Placeholder for a byte after *bound_endex*.
- **charsep** (*str*) – Separator between byte data and character data.
- **charend** (*str*) – Separator after character data.
- **stream** (*IO stream*) – Stream to write text onto. If `Ellipsis`, it uses `sys.stdout`. If not `None`, the function returns `None`.

Returns

str – Textual hex dump, if *stream* is `None`.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']], offset=0xDA7A)
>>> memory.hexdump()
0000DA7B  41 42 43 -- -- 78 79 7A -- -- -- -- -- -- -- --  |ABC  xyz        |
>>> memory.hexdump(stream=None)
'0000DA7B  41 42 43 -- -- 78 79 7A -- -- -- -- -- -- -- --  |ABC  xyz        |'
>>> memory.hexdump(start=0xDA7A, charmap=None)
0000DA7A  -- 41 42 43 -- -- 78 79 7A -- -- -- -- -- -- --
>>> memory.hexdump(start=0xDA7A)
0000DA7A  -- 41 42 43 -- -- 78 79 7A -- -- -- -- -- -- --  | ABC  xyz       |
>>> memory.hexdump(start=0xDA70)
0000DA70  -- -- -- -- -- -- -- -- -- -- -- 41 42 43 -- --  |           ABC  |
0000DA80  78 79 7A -- -- -- -- -- -- -- -- -- -- -- -- --  |xyz             |
>>> memory.bound_span = (0xDA78, 0xDA88)
>>> memory.hexdump(start=0xDA70)
0000DA70  >> >> >> >> >> >> >> >> -- -- -- 41 42 43 -- --  |>>>>>>>>   ABC  |
0000DA80  78 79 7A -- -- -- -- -- << << << << << << << <<  |xyz     <<<<<<<<|
>>> memory.hexdump(start=0xDA70, headfmt=...)
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000DA70  >> >> >> >> >> >> >> >> -- -- -- 41 42 43 -- --  |>>>>>>>>   ABC  |
0000DA80  78 79 7A -- -- -- -- -- << << << << << << << <<  |xyz     <<<<<<<<|
>>> memory.hexdump(start=0xDA78, endex=0xDA84, columns=4)
0000DA78  -- -- -- 41  |   A|
0000DA7C  42 43 -- --  |BC  |
0000DA80  78 79 7A --  |xyz |
```

abstract index(*item*, *start=None*, *endex=None*)

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.

See also:

find()

abstract 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 bytesparse 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'$']]
```

abstract 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()

abstract 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()

abstract intervals(*start=None*, *endex=None*)

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 bytesparse 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))
[]
```

abstract items(*start=None*, *endex=None*, *pattern=None*)

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 bytesparse 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)]
```

abstract keys(*start=None*, *endex=None*)

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 bytesparse 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]
```

abstract peek(*address*)

Gets the item at an address.

Returns

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

Examples

```
>>> from bytesparse 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
```

(continues on next page)
~~~

(continued from previous page)

```
>>> memory.peek(10)  # -> ord('z') = 122
122
>>> memory.peek(0)
None
>>> memory.peek(7)
None
>>> memory.peek(11)
None
```

abstract 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 bytesparse 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
```

abstract 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()

abstract 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()

abstract pop(*address=None*, *default=None*)

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 bytesparse 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
```

abstract pop_backup(*address=None*)

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()

abstract 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()

abstract popitem()

Pops the last item.

Returns

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

See also:

popitem_backup() popitem_restore()

Examples

```
>>> from bytesparse 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
```

abstract popitem_backup()

Backups a *popitem()* operation.

Returns

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

See also:

popitem() popitem_restore()

abstract popitem_restore(*address*, *item*)

Restores a *popitem()* operation.

Parameters

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

See also:

popitem() popitem_backup()

abstract read(*address*, *size*)

Reads data.

Reads a chunk of data from an address, with a given size. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.
- **size** (*int*) – Chunk size.

Returns

memoryview – A view over the addressed chunk.

Raises

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

Examples

```
>>> from bytesparse 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.read(2, 3))
b'BCD'
>>> bytes(memory.read(9, 1))
b'y'
>>> memory.read(4, 3)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.read(0, 6)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

abstract readinto(*address*, *buffer*)

Reads data into a pre-allocated buffer.

Provided a pre-allocated writable buffer (*e.g.* a `bytearray` or a `memoryview` slice of it), this method reads a chunk of data from an address, with the size of the target buffer. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.

- **buffer** (*writable*) – Pre-allocated buffer to fill with data.

Returns
int – Number of bytes read.

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

Examples

```
>>> from bytesparse 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']])
>>> buffer = bytearray(3)
>>> memory.readinto(2, buffer)
3
>>> buffer
bytearray(b'BCD')
>>> view = memoryview(buffer)
>>> memory.readinto(9, view[1:2])
1
>>> buffer
bytearray(b'ByD')
>>> memory.readinto(4, buffer)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.readinto(0, bytearray(6))
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

abstract remove(*item*, *start=None*, *endex=None*)

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 bytesparse 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
```

abstract remove_backup(*item*, *start=None*, *endex=None*)

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()

abstract remove_restore(*backup*)

Restores a *remove()* operation.

Parameters

backup (`Memory`) – Backup memory region.

See also:

remove() remove_backup()

abstract 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 bytesparse 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']]
```

abstract 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()

abstract 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()

abstract reverse()

Reverses the memory in-place.

Data is reversed within the memory *span*.

Examples

```
>>> from bytesparse 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']]
```

abstract rfind(*item*, *start=None*, *endex=None*)

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.

Warning: If the memory allows negative addresses, *rindex()* is more appropriate, because it raises `ValueError` if the item is not found.
~~~

See also:

rindex()

abstract rindex(*item*, *start=None*, *endex=None*)

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.

> **Warning:** If the memory allows negative addresses, *index()* is more appropriate, because it raises `ValueError` if the item is not found.

See also:

rfind()

abstract rvalues(*start=None*, *endex=None*, *pattern=None*)

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 bytesparse 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))
```

(continues on next page)

(continued from previous page)

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

abstract setdefault(*address*, *default=None*)

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 bytesparse 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
```

abstract 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()

abstract 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()

abstract shift(*offset*)

Shifts the items.

Parameters

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

See also:

shift_backup() shift_restore()

Examples

```
>>> from bytesparse 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']]
```

abstract 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()

abstract 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()

abstract property span: Tuple[int, int]

Memory address span.

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

Examples

```
>>> from bytesparse 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

abstract property start: int

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 *bound_start* not `None`, that is returned.

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

Examples

```
>>> from bytesparse 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

abstract to_blocks(*start=None*, *endex=None*)

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 bytesparse 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)]
[]
```

abstract to_bytes(*start=None*, *endex=None*)

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 bytesparse 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'
```

abstract update(*data*, *clear=False*, ***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 bytesparse 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']]
```

abstract update_backup(*data*, *clear=False*, ***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()

abstract update_restore(*backups*)

Restores an *update()* operation.

Parameters

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

See also:

update() update_backup()

abstract 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).

abstract values(*start=None*, *endex=None*, *pattern=None*)

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 bytesparse 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]
```
~~~

abstract view(*start=None*, *endex=None*)

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 bytesparse 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
```

abstract write(*address*, *data*, *clear=False*)

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 bytesparse 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']]
```

abstract write_backup(*address*, *data*, *clear=False*)

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()*

abstract write_restore(*backups*)

Restores a *write()* operation.

Parameters

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

See also:

write() *write_backup()*

3.3 bytesparse.inplace

In-place implementation.

This implementation in pure Python uses the basic `bytearray` data type to hold block data, which allows mutable in-place operations.

Classes

Memory	Virtual memory.
bytesparse	Wrapper for more *bytearray* compatibility.

3.3.1 Memory

class bytesparse.inplace.**Memory**(*start=None*, *endex=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.

See also:

`ImmutableMemory` `MutableMemory`

Variables

- **_blocks** (*list of blocks*) – A sequence of spaced blocks, sorted by address.
- **_bound_start** (*int*) – Memory bounds start address. Any data before this address is automatically discarded; disabled if `None`.
- **_bound_endex** (*int*) – Memory bounds 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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.

Examples

```
>>> from bytesparse import Memory
```

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

```
>>> memory = Memory(start=3, endex=10)
>>> memory.bound_span
(3, 10)
>>> memory.write(0, b'Hello, World!')
>>> memory.to_blocks()
[[3, b'lo, Wor']]
```

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

Attributes

bound_endex	Bounds exclusive end address.
bound_span	Bounds span addresses.
bound_start	Bounds start address.
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.

Methods

__init__	
align	Floods blocks to align their boundaries.
align_backup	Backups an *align()* operation.
align_restore	Restores an *align()* operation.
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.
chop	Iterates over chopped blocks.
clear	Clears an address range.
clear_backup	Backups a *clear()* operation.
clear_restore	Restores a *clear()* operation.
collapse_blocks	Collapses a generic sequence of blocks.
content_blocks	Iterates over blocks.
content_items	Iterates over content address and value pairs.
content_keys	Iterates over content addresses.
content_values	Iterates over content values.
copy	Creates a deep 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.

continues on next page

Table 4 – continued from previous page

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.
hexdump	Textual hex dump.
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.
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.
popitem	Pops the last item.
popitem_backup	Backups a *popitem()* operation.
popitem_restore	Restores a *popitem()* operation.
read	Reads data.
readinto	Reads data into a pre-allocated buffer.
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.
rvalues	Iterates over values, reversed order.
setdefault	Defaults a value.
setdefault_backup	Backups a *setdefault()* operation.

continues on next page

Table 4 – continued from previous page

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.

__add__(*value*)

Concatenates items.

Equivalent to `self.copy() += items` of a `MutableMemory`.

See also:

`MutableMemory.__iadd__()`

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC')
>>> memory2 = memory1 + b'xyz'
>>> memory2.to_blocks()
[[0, b'ABCxyz']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.content_endex
4
>>> memory3 = memory1 + memory2
>>> memory3.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

__bool__()

Has any items.

Returns

bool – Has any items.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> bool(memory)
False
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> bool(memory)
True
```

__bytes__()

Creates a bytes clone.

Returns
bytes – Cloned data.

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

Examples

```
>>> from bytesparse import Memory
```

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

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

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> bytes(memory)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> bytes(memory)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

classmethod __class_getitem__()

Represent a PEP 585 generic type

E.g. for t = list[int], t.__origin__ is list and t.__args__ is (int,).

__contains__(*item*)

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 bytesparse 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.

Returns
ImmutableMemory – Shallow copy.

__deepcopy__()
Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

__delitem__(*key*)
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 bytesparse 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__(*other*)

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 bytesparse 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
```

__getitem__(*key*)

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 bytesparse 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'$'], [8, b'xyz']])
>>> memory[9]  # -> ord('y') = 121
121
>>> memory[:3]._blocks
[[1, b'AB']]
>>> memory[3:10]._blocks
[[3, b'CD'], [6, b'$'], [8, b'xy']]
>>> bytes(memory[3:10:b'.'])
b'CD.$.xy'
>>> memory[memory.endex]
None
>>> bytes(memory[3:10:3])
b'C$y'
>>> memory[3:10:2]._blocks
[[3, b'C'], [6, b'y']]
>>> bytes(memory[3:10:2])
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

__hash__ = None

__iadd__(*value*)

Concatenates items.

Equivalent to `self.extend(value)`.

See also:

extend()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'ABC')
>>> memory += b'xyz'
>>> memory.to_blocks()
[[0, b'ABCxyz']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.content_endex
4
>>> memory1 += memory2
>>> memory1.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

__imul__(*times*)

Concatenates a repeated copy.

Equivalent to `self.extend(items)` repeated *times* times.

See also:

extend()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'ABC')
>>> memory *= 3
>>> memory.to_blocks()
[[0, b'ABCABCABC']]
```

```
>>> memory = Memory.from_blocks([[1, b'ABC']])
>>> memory *= 3
>>> memory.to_blocks()
[[1, b'ABCABCABC']]
```

__init__(*start=None*, *endex=None*)

__ior__(*value*)

Merges memories.

Equivalent to `self.write(0, value)`.

See also:

extend()

See also:

`MutableMemory.__ior__()`

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1 |= memory2
>>> memory1.to_blocks()
[[1, b'ABC'], [5, b'xyz']]
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory1 |= b'xyz'
>>> memory2.to_blocks()
[[0, b'xyzBC']]
```

__iter__()

Iterates over values.

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

Yields

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> list(memory)
[65, 66, 67, None, 120, 121, 122]
```

__len__()

Actual length.

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

Returns

int – Memory length.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> len(memory)
0
```

```
>>> memory = Memory(start=3, endex=10)
>>> len(memory)
7
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [9, b'xyz']])
>>> len(memory)
11
```

```
>>> memory = Memory.from_blocks([[3, b'ABC'], [9, b'xyz']], start=1, endex=15)
>>> len(memory)
14
```

__mul__(*times*)

Concatenates a repeated copy.

Equivalent to `self.copy() *= items` of a `MutableMemory`.

See also:

`MutableMemory.__imul__()`

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory2 = memory1 * 3
>>> memory2.to_blocks()
[[0, b'ABCABCABC']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = memory1 * 3
>>> memory2.to_blocks()
[[1, b'ABCABCABC']]
```

__or__(*value*)

Merges memories.

Equivalent to `self.copy() |= items` of a `MutableMemory`.

See also:

`MutableMemory.__ior__()`

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory3 = memory1 | memory2
>>> memory3.to_blocks()
[[1, b'ABC'], [5, b'xyz']]
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory2 = memory1 | b'xyz'
>>> memory2.to_blocks()
[[0, b'xyzBC']]
```

__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.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> list(memory)
[65, 66, 67, None, 120, 121, 122]
>>> list(reversed(memory))
[122, 121, 120, None, 67, 66, 65]
```

__setitem__(*key*, *value*)

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 bytesparse 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']]
```

__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 bytesparse 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']]>
```

classmethod __subclasshook__(*C*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__

list of weak references to the object (if defined)

_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 bytesparse 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 bytesparse 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 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 bytesparse 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]
```

_erase(*start*, *endex*, *shift_after*)

Erases an address range.

Low-level method to erase data within the underlying data structure.

Parameters

- **start** (*int*) – Start address of the erasure range.
- **endex** (*int*) – Exclusive end address of the erasure range.
- **shift_after** (*bool*) – Shifts addresses of blocks after the end of the range, subtracting the size of the range itself. If data blocks before and after the address range are contiguous after erasure, merge the two blocks together.

_place(*address*, *data*, *shift_after*)

Places data.

Low-level method to place data into the underlying data structure.

Parameters

- **address** (*int*) – Address of the insertion point.
- **data** (bytearray) – Data to insert.
- **shift_after** (*bool*) – Shifts the addresses of blocks after the insertion point, adding the size of the inserted data.

_prebound_endex(*start_min*, *size*)

Bounds final data.

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

Parameters

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

See also:

_prebound_endex_backup()

_prebound_endex_backup(*start_min*, *size*)

Backups a *_prebound_endex()* operation.

Parameters

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

Returns

ImmutableMemory – Backup memory region.

See also:

_prebound_endex()

_prebound_start(*endex_max*, *size*)

Bounds initial data.

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

Parameters

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

See also:

_prebound_start_backup()

_prebound_start_backup(*endex_max*, *size*)

Backups a *_prebound_start()* operation.

Parameters

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

Returns

ImmutableMemory – Backup memory region.

See also:

_prebound_start()

align(*modulo*, *start=None*, *endex=None*, *pattern=0*)

Floods blocks to align their boundaries.

Parameters

- **modulo** (*int*) – Alignment modulo.
- **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:

align_backup() align_restore() flood()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11	12
	[A	B	C]			[x	y	z]				
[0	A	B	C	0	1	x	y	z	1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.align(4, pattern=b'0123')
>>> memory.to_blocks()
[[0, b'0ABC01xyz123']]
```

~~~

0	1	2	3	4	5	6	7	8	9	10	11	12
	[A	B	C]				[0	1	2	3]		
[x	A	B	C]			[x	0	1	2	3	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [7, b'0123']])
>>> memory.align(2, pattern=b'xyz')
>>> memory.to_blocks()
[[0, b'xABC'], [6, b'x0123z']]
```

~~~

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']])
>>> memory.align(2, start=3, endex=7, pattern=b'.@')
>>> memory.to_blocks()
[[1, b'ABC'], [6, b'xyz']]
```

align_backup(*modulo*, *start=None*, *endex=None*)

Backups an *align()* operation.

Parameters

- **modulo** (*int*) – Alignment modulo.
- **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:

align() align_restore()

align_restore(*gaps*)

Restores an *align()* operation.

Parameters

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

See also:

align() align_backup()

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 bytesparse 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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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 bounds limits, *bound_start* or *bound_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 bytesparse 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)
```

(continues on next page)

(continued from previous page)

```
(2, 8)
>>> memory.bound(None, 6)
(1, 6)
```

property bound_endex: int | None

Bounds exclusive end address.

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_endex = 10
>>> memory.to_blocks()
[[5, b'Hello']]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, endex=10)
>>> memory.to_blocks()
[[5, b'Hello']]
```

Type
int

property bound_span: Tuple[int | None, int | None]

Bounds span addresses.

A `tuple` holding *bound_start* and *bound_endex*.

Notes

Assigning `None` to `MutableMemory.bound_span` sets both *bound_start* and *bound_endex* to `None` (equivalent to `(None, None)`).

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_span = (7, 13)
>>> memory.to_blocks()
[[7, b'llo, W']]
>>> memory.bound_span = None
>>> memory.bound_span
(None, None)
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=7, endex=13)
>>> memory.to_blocks()
[[7, b'llo, W']]
```

Type
tuple of int

property bound_start: int | None

Bounds start address.

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_start = 10
>>> memory.to_blocks()
[[10, b', World!']]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=10)
>>> memory.to_blocks()
[[10, b', World!']]
```

Type
int

chop(*width*, *start=None*, *endex=None*, *align=False*)

Iterates over chopped blocks.

The provided range is split into sub-ranges of a fixed width. For each sub-range, it yields views of the contained block chunks.

Parameters

- **width** (*int*) – Sub-range width.
- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.
- **align** (*bool*) – Sub-ranges are aligned to *width*.

Examples

```
>>> from bytesparse import Memory
```

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

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> chopping = memory.chop(2, align=False)
>>> [(address, bytes(view)) for address, view in chopping]
[(1, b'AB'), (3, b'C'), (6, b'xy'), (8, b'z')]
>>> chopping = memory.chop(2, align=True)
>>> [(address, bytes(view)) for address, view in chopping]
[(1, b'A'), (2, b'BC'), (6, b'xy'), (8, b'z')]
```

clear(*start=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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()

classmethod 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

```
>>> from bytesparse import Memory
```

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'$'],
...     [6, b'xyz'],
... ]
>>> Memory.collapse_blocks(blocks)
[[0, b'$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'$'],
... ]
>>> Memory.collapse_blocks(blocks)
[[0, b'0$2'], [4, b'ABxyz']]
```

content_blocks(*block_index_start=None*, *block_index_endex=None*, *block_index_step=None*)

Iterates over blocks.

Iterates over data blocks within a block index range.

Parameters

- **block_index_start** (*int*) – Inclusive block start index. A negative index is referred to *content_parts*. If None, `0` is considered.
- **block_index_endex** (*int*) – Exclusive block end index. A negative index is referred to *content_parts*. If None, *content_parts* is considered.
- **block_index_step** (*int*) – Block index step, which can be negative. If None, `1` is considered.

Yields

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

See also:

content_parts

Examples

```
>>> from bytesparse 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.content_blocks()]
[[1, b'AB'], [5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(1, 2)]
[[5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(3, 5)]
```

(continues on next page)

(continued from previous page)

```
[]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_start=-2)]
[[5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_endex=-1)]
[[1, b'AB'], [5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_step=2)]
[[1, b'AB'], [7, b'123']]
```

property content_endex: int

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 bounds, *start* is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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

property content_endin: int

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 bounds, *start* minus one is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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]
```

(continues on next page)

(continued from previous page)

```
>>> list(memory.content_keys(3, 5))
[]
```

property content_parts: int

Number of blocks.

Returns

int – The number of blocks.

Examples

```
>>> from bytesparse 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
```

property content_size: int

Actual content size.

Returns

int – The sum of all block lengths.

Examples

```
>>> from bytesparse 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
```

property content_span: Tuple[int, int]

Memory content address span.

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

Examples

```
>>> from bytesparse 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

property content_start: int

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 bounds, 0 is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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=None*, *endex=None*)

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 bytesparse 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))
[]
```

property contiguous: bool

Contains contiguous data.

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

If bounds are defined, there must be no empty space also towards them.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory.contiguous
False
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.contiguous
False
```

Type
bool

copy()

Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory()
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(None, None)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory(start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory2 = memory1.copy()
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
>>> memory2.bound_span = (2, 19)
>>> memory1 == memory2
True
```

```
>>> memory1 = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory2 = memory1.copy()
[[5, b'ABC'], [9, b'xyz']]
>>> memory1 == memory2
True
```

count(*item*, *start=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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=None*, *endex=None*, *bound=True*)

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 bounds 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.

Examples

```
>>> from bytesparse import Memory
```

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

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

delete(*start=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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()

property endex: int

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 *bound_endex* not None, that is returned.

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

Examples

```
>>> from bytesparse 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

property endin: int

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 *bound_endex* not None, that minus one is returned.

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

Examples

```
>>> from bytesparse 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 bytesparse 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=0*)

Concatenates items.

Appends *items* after *content_endex*. 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:

__iadd__() *extend_backup()* *extend_restore()*

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.extend(b'123')
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz123']]
```

~~~

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.extend(range(49, 52), offset=4)
```

(continues on next page)

(continued from previous page)

```
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz'], [12, b'123']]
```

~~~

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.extend(memory2)
>>> memory1.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

extend_backup(*offset=0*)

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=None*, *endex=None*, *pattern=None*, *step=None*, *bound=True*)

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

fill(*start=None*, *endex=None*, *pattern=0*)

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 bytesparse 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=None*, *endex=None*)

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=None*, *endex=None*)

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.

> **Warning:** If the memory allows negative addresses, *index()* is more appropriate, because it raises `ValueError` if the item is not found.

See also:

index()
~~~

flood(*start=None*, *endex=None*, *pattern=0*)

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 bytesparse 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(start=3, endex=7, pattern=b'123')
>>> memory.to_blocks()
[[1, b'ABC23xyz']]
```

flood_backup(*start=None*, *endex=None*)

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=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse 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  # noqa
>>> blocks = hr.load_blocks('records.hex')
>>> memory = Memory.from_blocks(Memory.collapse_blocks(blocks))
>>> memory
    ...
```

classmethod from_bytes(*data*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse 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=0*, *start=None*, *endex=None*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 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=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse import Memory
```

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

~~~

```
>>> memory1 = Memory.from_bytes(b'ABC', 10)
>>> memory2 = Memory.from_memory(memory1, -3)
>>> memory2._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._blocks, memory2._blocks))
True
```

classmethod from_values(*values*, *offset=0*, *start=None*, *endex=None*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 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 bytesparse import Memory
```

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

~~~

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

gaps(*start=None*, *endex=None*)

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 bytesparse 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=None*)

Gets the item at an address.
~~~

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

Examples

```
>>> from bytesparse 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 bytesparse 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
```

hexdump(*start=None*, *endex=None*, *columns=16*, *addrfmt='{:08X} '*, *bytefmt=' {:02X}'*, *headfmt=None*, *charmap='..... !"#$%&\\'()*+, -./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\\\]^_`abcdefghijklmnopqrstuvwxyz{|}~.....><'*, *emptystr=' --'*, *beforestr=' >>'*, *afterstr=' <<'*, *charsep=' |'*, *charend='|'*, *stream=Ellipsis*)

Textual hex dump.

This function generates a hex dump of the bytes within the specified range.

If *stream* is not `None`, the hex dump is written on it, otherwise it is returned as a `str`.

The default output is similar to that of `hexdump` or `xxd` commands, with some degree of tweaking. In case more customized formatting is desired, a dedicated custom function can be written by carefully looping over *values()*.

Parameters

- **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.
- **columns** (*int*) – Number of byte columns per row.
- **addrfmt** (*str*) – Address formatting string.
- **bytefmt** (*str*) – Byte formatting string.
- **headfmt** (*str*) – Header offset formatting string. If `Ellipsis`, it applies that of *bytefmt*. If `None`, no header row is generated.
- **charmap** (*mapping*) – Mapping to convert a byte integer into a string character. If `None`, no character data are appended to each row.

  The table is structured this way:

  – The initial 256 bytes map actual byte values.

  – Index `0x100` represents an empty byte (`None`).

  – Index `0x101` represents a byte before *start*.

  – Index `0x102` represents a byte after *endex*.
- **emptystr** (*str*) – Placeholder for an empty byte (`None` value).
- **beforestr** (*str*) – Placeholder for a byte before *bound_start*.
~~~

- **afterstr** (*str*) – Placeholder for a byte after *bound_endex*.
- **charsep** (*str*) – Separator between byte data and character data.
- **charend** (*str*) – Separator after character data.
- **stream** (*IO stream*) – Stream to write text onto. If `Ellipsis`, it uses `sys.stdout`. If not `None`, the function returns `None`.

Returns

str – Textual hex dump, if *stream* is `None`.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']], offset=0xDA7A)
>>> memory.hexdump()
0000DA7B  41 42 43 -- -- 78 79 7A -- -- -- -- -- -- -- --  |ABC  xyz        |
>>> memory.hexdump(stream=None)
'0000DA7B  41 42 43 -- -- 78 79 7A -- -- -- -- -- -- -- --  |ABC  xyz        |'
>>> memory.hexdump(start=0xDA7A, charmap=None)
0000DA7A  -- 41 42 43 -- -- 78 79 7A -- -- -- -- -- -- --
>>> memory.hexdump(start=0xDA7A)
0000DA7A  -- 41 42 43 -- -- 78 79 7A -- -- -- -- -- -- --  | ABC  xyz       |
>>> memory.hexdump(start=0xDA70)
0000DA70  -- -- -- -- -- -- -- -- -- -- -- 41 42 43 -- --  |           ABC  |
0000DA80  78 79 7A -- -- -- -- -- -- -- -- -- -- -- -- --  |xyz             |
>>> memory.bound_span = (0xDA78, 0xDA88)
>>> memory.hexdump(start=0xDA70)
0000DA70  >> >> >> >> >> >> >> >> -- -- -- 41 42 43 -- --  |>>>>>>>>   ABC  |
0000DA80  78 79 7A -- -- -- -- -- << << << << << << << <<  |xyz     <<<<<<<<|
>>> memory.hexdump(start=0xDA70, headfmt=...)
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000DA70  >> >> >> >> >> >> >> >> -- -- -- 41 42 43 -- --  |>>>>>>>>   ABC  |
0000DA80  78 79 7A -- -- -- -- -- << << << << << << << <<  |xyz     <<<<<<<<|
>>> memory.hexdump(start=0xDA78, endex=0xDA84, columns=4)
0000DA78  -- -- -- 41  |   A|
0000DA7C  42 43 -- --  |BC  |
0000DA80  78 79 7A --  |xyz |
```

index(*item*, *start=None*, *endex=None*)

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.

See also:

find()

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*, *pattern=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse import Memory
```

```
>>> from itertools import islice
>>> memory = Memory()
>>> list(memory.keys())
[]
>>> list(memory.keys(endex=8))
```

(continues on next page)
~~~

(continued from previous page)

```
[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]
```

peek(*address*)

Gets the item at an address.

Returns

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

Examples

```
>>> from bytesparse 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 bytesparse 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=None*, *default=None*)

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 bytesparse 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=None*)

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 bytesparse 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 or byte*) – Item to restore.

See also:

popitem() popitem_backup()

read(*address*, *size*)

Reads data.

Reads a chunk of data from an address, with a given size. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.
- **size** (*int*) – Chunk size.

Returns
memoryview – A view over the addressed chunk.

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

Examples

```
>>> from bytesparse 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.read(2, 3))
b'BCD'
>>> bytes(memory.read(9, 1))
b'y'
>>> memory.read(4, 3)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.read(0, 6)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

readinto(*address*, *buffer*)

Reads data into a pre-allocated buffer.

Provided a pre-allocated writable buffer (*e.g.* a bytearray or a memoryview slice of it), this method reads a chunk of data from an address, with the size of the target buffer. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.
- **buffer** (*writable*) – Pre-allocated buffer to fill with data.

Returns
int – Number of bytes read.

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

Examples

```
>>> from bytesparse 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']])
>>> buffer = bytearray(3)
>>> memory.readinto(2, buffer)
3
>>> buffer
bytearray(b'BCD')
>>> view = memoryview(buffer)
>>> memory.readinto(9, view[1:2])
1
>>> buffer
bytearray(b'ByD')
>>> memory.readinto(4, buffer)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.readinto(0, bytearray(6))
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

remove(*item*, *start=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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 bytesparse 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=None*, *endex=None*)

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.

Warning: If the memory allows negative addresses, *rindex()* is more appropriate, because it raises `ValueError` if the item is not found.
~~~

See also:

rindex()

rindex(*item*, *start=None*, *endex=None*)

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.

Warning: If the memory allows negative addresses, *index()* is more appropriate, because it raises `ValueError` if the item is not found.

See also:

rfind()

rvalues(*start=None*, *endex=None*, *pattern=None*)

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 bytesparse 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))
```

(continues on next page)

(continued from previous page)

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

setdefault(*address*, *default=None*)

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 bytesparse 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 bytesparse 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()

property span: Tuple[int, int]

Memory address span.

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

Examples

```
>>> from bytesparse 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

property start: int

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 *bound_start* not `None`, that is returned.

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

Examples

```
>>> from bytesparse 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=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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'
```

update(*data*, *clear=False*, ***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 bytesparse 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=False*, ***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=None*, *endex=None*, *pattern=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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=False*)

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 bytesparse 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=False*)

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()

3.3.2 bytesparse

class bytesparse.inplace.**bytesparse**(**args*, *start=None*, *endex=None*)

Wrapper for more *bytearray* compatibility.

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

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

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

See also:

`ImmutableMemory` `MutableMemory`

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 deleted. 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 deleted.

Examples

```
>>> from bytesparse import bytesparse
```

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

```
>>> memory = bytesparse(start=3, endex=10)
>>> memory.bound_span
(3, 10)
>>> memory.write(0, b'Hello, World!')
>>> memory.to_blocks()
[[3, b'lo, Wor']]
```

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

```
>>> memory = bytesparse(b'Hello, World!')
>>> memory.to_blocks()
[[0, b'Hello, World!']]
```

```
>>> memory = bytesparse(3)
>>> memory.to_blocks()
[[0, b'\x00\x00\x00']]
```

```
>>> memory = bytesparse([65, 66, 67])
>>> memory.to_blocks()
[[0, b'ABC']]
```

```
>>> memory = bytesparse('ASCII string', 'ascii')
>>> memory.to_blocks()
[[0, b'ASCII string']]
```

```
>>> memory = bytesparse('Non-ASCII: \u2204', 'ascii', 'backslashreplace')
>>> memory.to_blocks()
[[0, b'Non-ASCII: \\u2204']]
```

```
>>> memory = bytesparse('Non-ASCII: \u2204', 'ascii', 'xmlcharrefreplace')
>>> memory.to_blocks()
[[0, b'Non-ASCII: &#8708;']]
```

```
>>> memory = bytesparse('Non-ASCII: \u2204', 'ascii', 'replace')
>>> memory.to_blocks()
[[0, b'Non-ASCII: ?']]
```

```
>>> memory = bytesparse('Non-ASCII: \u2204', 'ascii', 'ignore')
>>> memory.to_blocks()
[[0, b'Non-ASCII: ']]
```

```
>>> memory = bytesparse('Non-ASCII: \u2204', 'ascii', 'strict')
Traceback (most recent call last):
    ...
UnicodeEncodeError: 'ascii' codec can't encode character '\u2204' in position 11:␣
↪ordinal not in range(128)
```

```
>>> memory = bytesparse('Missing encoding')
Traceback (most recent call last):
    ...
TypeError: string argument without an encoding
```

Method Groups

- **Creation** – *__init__()*
- **Addressing** – *_rectify_address()* *_rectify_span()*

Attributes

bound_endex	Bounds exclusive end address.
bound_span	Bounds span addresses.
bound_start	Bounds start address.
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.

Methods

__init__	
align	Floods blocks to align their boundaries.
align_backup	Backups an *align()* operation.
align_restore	Restores an *align()* operation.
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.
chop	Iterates over chopped blocks.
clear	Clears an address range.
clear_backup	Backups a *clear()* operation.
clear_restore	Restores a *clear()* operation.
collapse_blocks	Collapses a generic sequence of blocks.
content_blocks	Iterates over blocks.
content_items	Iterates over content address and value pairs.
content_keys	Iterates over content addresses.
content_values	Iterates over content values.
copy	Creates a deep 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.

continues on next page

Table 5 – continued from previous page

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.
hexdump	Textual hex dump.
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.
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.
popitem	Pops the last item.
popitem_backup	Backups a *popitem()* operation.
popitem_restore	Restores a *popitem()* operation.
read	Reads data.
readinto	Reads data into a pre-allocated buffer.
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.
rvalues	Iterates over values, reversed order.
setdefault	Defaults a value.
setdefault_backup	Backups a *setdefault()* operation.

continues on next page

Table 5 – continued from previous page

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.

__add__(*value*)

Concatenates items.

Equivalent to `self.copy() += items` of a `MutableMemory`.

See also:

`MutableMemory.__iadd__()`

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC')
>>> memory2 = memory1 + b'xyz'
>>> memory2.to_blocks()
[[0, b'ABCxyz']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.content_endex
4
>>> memory3 = memory1 + memory2
>>> memory3.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

__bool__()

Has any items.

Returns

bool – Has any items.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> bool(memory)
False
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> bool(memory)
True
```

__bytes__()

Creates a bytes clone.

Returns
bytes – Cloned data.

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

Examples

```
>>> from bytesparse import Memory
```

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

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

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> bytes(memory)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> bytes(memory)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

classmethod __class_getitem__()

Represent a PEP 585 generic type

E.g. for t = list[int], t.__origin__ is list and t.__args__ is (int,).

__contains__(*item*)

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 bytesparse 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.

Returns
ImmutableMemory – Shallow copy.

__deepcopy__()
Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

__delitem__(*key*)
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 bytesparse 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__(*other*)

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 bytesparse 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
```

__getitem__(*key*)

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 bytesparse 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'$'], [8, b'xyz']])
>>> memory[9]  # -> ord('y') = 121
121
>>> memory[:3]._blocks
[[1, b'AB']]
>>> memory[3:10]._blocks
[[3, b'CD'], [6, b'$'], [8, b'xy']]
>>> bytes(memory[3:10:b'.'])
b'CD.$.xy'
>>> memory[memory.endex]
None
>>> bytes(memory[3:10:3])
b'C$y'
>>> memory[3:10:2]._blocks
[[3, b'C'], [6, b'y']]
>>> bytes(memory[3:10:2])
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

__hash__ = None

__iadd__(*value*)

Concatenates items.

Equivalent to `self.extend(value)`.

See also:

extend()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'ABC')
>>> memory += b'xyz'
>>> memory.to_blocks()
[[0, b'ABCxyz']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.content_endex
4
>>> memory1 += memory2
>>> memory1.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

__imul__(*times*)

Concatenates a repeated copy.

Equivalent to `self.extend(items)` repeated *times* times.

See also:

extend()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'ABC')
>>> memory *= 3
>>> memory.to_blocks()
[[0, b'ABCABCABC']]
```

```
>>> memory = Memory.from_blocks([[1, b'ABC']])
>>> memory *= 3
>>> memory.to_blocks()
[[1, b'ABCABCABC']]
```

__init__(**args*, *start=None*, *endex=None*)

__ior__(*value*)

Merges memories.

Equivalent to `self.write(0, value)`.

See also:

extend()

See also:

`MutableMemory.__ior__()`

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1 |= memory2
>>> memory1.to_blocks()
[[1, b'ABC'], [5, b'xyz']]
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory1 |= b'xyz'
>>> memory2.to_blocks()
[[0, b'xyzBC']]
```

__iter__()

Iterates over values.

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

Yields

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> list(memory)
[65, 66, 67, None, 120, 121, 122]
```

__len__()

Actual length.

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

Returns

int – Memory length.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> len(memory)
0
```

```
>>> memory = Memory(start=3, endex=10)
>>> len(memory)
7
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [9, b'xyz']])
>>> len(memory)
11
```

```
>>> memory = Memory.from_blocks([[3, b'ABC'], [9, b'xyz']], start=1, endex=15)
>>> len(memory)
14
```

__mul__(*times*)

Concatenates a repeated copy.

Equivalent to `self.copy() *= items` of a `MutableMemory`.

See also:

`MutableMemory.__imul__()`

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory2 = memory1 * 3
>>> memory2.to_blocks()
[[0, b'ABCABCABC']]
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = memory1 * 3
>>> memory2.to_blocks()
[[1, b'ABCABCABC']]
```

__or__(*value*)

Merges memories.

Equivalent to `self.copy() |= items` of a `MutableMemory`.

See also:

`MutableMemory.__ior__()`

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory3 = memory1 | memory2
>>> memory3.to_blocks()
[[1, b'ABC'], [5, b'xyz']]
```

```
>>> memory1 = Memory.from_bytes(b'ABC', offset=2)
>>> memory2 = memory1 | b'xyz'
>>> memory2.to_blocks()
[[0, b'xyzBC']]
```

__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.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> list(memory)
[65, 66, 67, None, 120, 121, 122]
>>> list(reversed(memory))
[122, 121, 120, None, 67, 66, 65]
```

__setitem__(*key*, *value*)

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 bytesparse 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']]
```

__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 bytesparse 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']]>
```

classmethod __subclasshook__(*C*)

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__

list of weak references to the object (if defined)

_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 bytesparse 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 bytesparse 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 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 bytesparse 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]
```

_erase(*start*, *endex*, *shift_after*)

Erases an address range.

Low-level method to erase data within the underlying data structure.

Parameters

- **start** (*int*) – Start address of the erasure range.
- **endex** (*int*) – Exclusive end address of the erasure range.
- **shift_after** (*bool*) – Shifts addresses of blocks after the end of the range, subtracting the size of the range itself. If data blocks before and after the address range are contiguous after erasure, merge the two blocks together.

_place(*address*, *data*, *shift_after*)

Places data.

Low-level method to place data into the underlying data structure.

Parameters

- **address** (*int*) – Address of the insertion point.
- **data** (bytearray) – Data to insert.
- **shift_after** (*bool*) – Shifts the addresses of blocks after the insertion point, adding the size of the inserted data.

_prebound_endex(*start_min*, *size*)

Bounds final data.

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

Parameters

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

See also:

_prebound_endex_backup()

_prebound_endex_backup(*start_min*, *size*)

Backups a *_prebound_endex()* operation.

Parameters

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

Returns

ImmutableMemory – Backup memory region.

See also:

_prebound_endex()

_prebound_start(*endex_max*, *size*)

Bounds initial data.

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

Parameters

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

See also:

_prebound_start_backup()

_prebound_start_backup(*endex_max*, *size*)

Backups a *_prebound_start()* operation.

Parameters

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

Returns

ImmutableMemory – Backup memory region.

See also:

_prebound_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.

align(*modulo*, *start=None*, *endex=None*, *pattern=0*)

Floods blocks to align their boundaries.

Parameters

- **modulo** (*int*) – Alignment modulo.

- **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:

align_backup() align_restore() flood()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11	12
	[A	B	C]			[x	y	z]				
[0	A	B	C	0	1	x	y	z	1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.align(4, pattern=b'0123')
>>> memory.to_blocks()
[[0, b'0ABC01xyz123']]
```

~~~

0	1	2	3	4	5	6	7	8	9	10	11	12
	[A	B	C]				[0	1	2	3]		
[x	A	B	C]			[x	0	1	2	3	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [7, b'0123']])
>>> memory.align(2, pattern=b'xyz')
>>> memory.to_blocks()
[[0, b'xABC'], [6, b'x0123z']]
```

~~~

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']])
>>> memory.align(2, start=3, endex=7, pattern=b'.@')
>>> memory.to_blocks()
[[1, b'ABC'], [6, b'xyz']]
```

align_backup(*modulo*, *start=None*, *endex=None*)

Backups an *align()* operation.

Parameters

- **modulo** (*int*) – Alignment modulo.

- **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:

align() align_restore()

align_restore(*gaps*)

Restores an *align()* operation.

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

See also:

align() align_backup()

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 bytesparse 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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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 bounds limits, *bound_start* or *bound_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 bytesparse 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)
```

property bound_endex: int | None

Bounds exclusive end address.

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_endex = 10
>>> memory.to_blocks()
[[5, b'Hello']]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, endex=10)
>>> memory.to_blocks()
[[5, b'Hello']]
```

Type
int
~~~

property bound_span: Tuple[int | None, int | None]

Bounds span addresses.

A `tuple` holding *bound_start* and *bound_endex*.

Notes

Assigning `None` to `MutableMemory.bound_span` sets both *bound_start* and *bound_endex* to `None` (equivalent to `(None, None)`).

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_span = (7, 13)
>>> memory.to_blocks()
[[7, b'llo, W']]
>>> memory.bound_span = None
>>> memory.bound_span
(None, None)
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=7, endex=13)
>>> memory.to_blocks()
[[7, b'llo, W']]
```

Type
tuple of int

property bound_start: int | None

Bounds start address.

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

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_start = 10
>>> memory.to_blocks()
[[10, b', World!']]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=10)
>>> memory.to_blocks()
[[10, b', World!']]
```

Type
int

chop(*width*, *start=None*, *endex=None*, *align=False*)

Iterates over chopped blocks.

The provided range is split into sub-ranges of a fixed width. For each sub-range, it yields views of the contained block chunks.

Parameters

- **width** (*int*) – Sub-range width.
- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.
- **align** (*bool*) – Sub-ranges are aligned to *width*.

Examples

```
>>> from bytesparse import Memory
```

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

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> chopping = memory.chop(2, align=False)
>>> [(address, bytes(view)) for address, view in chopping]
[(1, b'AB'), (3, b'C'), (6, b'xy'), (8, b'z')]
>>> chopping = memory.chop(2, align=True)
>>> [(address, bytes(view)) for address, view in chopping]
[(1, b'A'), (2, b'BC'), (6, b'xy'), (8, b'z')]
```

clear(*start=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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()

classmethod 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

```
>>> from bytesparse import Memory
```

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'$'],
...     [6, b'xyz'],
... ]
>>> Memory.collapse_blocks(blocks)
[[0, b'$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'$'],
... ]
>>> Memory.collapse_blocks(blocks)
[[0, b'0$2'], [4, b'ABxyz']]
```

content_blocks(*block_index_start=None*, *block_index_endex=None*, *block_index_step=None*)

Iterates over blocks.

Iterates over data blocks within a block index range.

Parameters

- **block_index_start** (*int*) – Inclusive block start index. A negative index is referred to *content_parts*. If None, `0` is considered.
- **block_index_endex** (*int*) – Exclusive block end index. A negative index is referred to *content_parts*. If None, *content_parts* is considered.
~~~

- **block_index_step** (*int*) – Block index step, which can be negative. If None, 1 is considered.

Yields

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

See also:

content_parts

Examples

```
>>> from bytesparse 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.content_blocks()]
[[1, b'AB'], [5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(1, 2)]
[[5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(3, 5)]
[]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_start=-2)]
[[5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_endex=-1)]
[[1, b'AB'], [5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_step=2)]
[[1, b'AB'], [7, b'123']]
```

property content_endex: int

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 bounds, *start* is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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

property content_endin: int

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 bounds, *start* minus one is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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))
[]
```

property content_parts: int

Number of blocks.

Returns
int – The number of blocks.

Examples

```
>>> from bytesparse 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
```

property content_size: int

Actual content size.

Returns

int – The sum of all block lengths.

Examples

```
>>> from bytesparse 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
```

property content_span: Tuple[int, int]

Memory content address span.

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

Examples

```
>>> from bytesparse 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

property content_start: int

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 bounds, 0 is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse 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=None*, *endex=None*)

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 bytesparse 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))
[]
```

property contiguous: bool

Contains contiguous data.

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

If bounds are defined, there must be no empty space also towards them.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory.contiguous
False
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.contiguous
False
```

Type
bool

copy()

Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory()
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(None, None)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory(start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory2 = memory1.copy()
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
>>> memory2.bound_span = (2, 19)
>>> memory1 == memory2
True
```

```
>>> memory1 = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory2 = memory1.copy()
[[5, b'ABC'], [9, b'xyz']]
>>> memory1 == memory2
True
```

count(*item*, *start=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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=None*, *endex=None*, *bound=True*)

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 bounds 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.

Examples

```
>>> from bytesparse import Memory
```

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

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

delete(*start=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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()

`property endex: int`

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 *bound_endex* not `None`, that is returned.

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

Examples

```
>>> from bytesparse 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

property endin: int

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 *bound_endex* not `None`, that minus one is returned.

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

Examples

```
>>> from bytesparse 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 bytesparse 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=0*)

Concatenates items.

Appends *items* after *content_endex*. 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:

__iadd__() extend_backup() extend_restore()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.extend(b'123')
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz123']]
```

~~~

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.extend(range(49, 52), offset=4)
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz'], [12, b'123']]
```

~~~

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.extend(memory2)
>>> memory1.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

extend_backup(*offset=0*)

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=None*, *endex=None*, *pattern=None*, *step=None*, *bound=True*)

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

fill(*start=None*, *endex=None*, *pattern=0*)

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 bytesparse 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=None*, *endex=None*)

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=None*, *endex=None*)

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.

> **Warning:** If the memory allows negative addresses, *index()* is more appropriate, because it raises `ValueError` if the item is not found.

See also:

index()
~~~

flood(*start=None*, *endex=None*, *pattern=0*)

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 bytesparse 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(start=3, endex=7, pattern=b'123')
>>> memory.to_blocks()
[[1, b'ABC23xyz']]
```

flood_backup(*start=None*, *endex=None*)

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=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse 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  # noqa
>>> blocks = hr.load_blocks('records.hex')
>>> memory = Memory.from_blocks(Memory.collapse_blocks(blocks))
>>> memory
    ...
```

classmethod from_bytes(*data*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse 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=0*, *start=None*, *endex=None*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 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=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 bytesparse import Memory
```

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

~~~

```
>>> memory1 = Memory.from_bytes(b'ABC', 10)
>>> memory2 = Memory.from_memory(memory1, -3)
>>> memory2._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._blocks, memory2._blocks))
True
```

classmethod from_values(*values*, *offset=0*, *start=None*, *endex=None*, *validate=True*)

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 deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **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 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 bytesparse import Memory
```

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

~~~

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

gaps(*start=None*, *endex=None*)

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 bytesparse 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=None*)

Gets the item at an address.
~~~

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

Examples

```
>>> from bytesparse 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 bytesparse 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
```

hexdump(*start=None*, *endex=None*, *columns=16*, *addrfmt='{:08X} '*, *bytefmt=' {:02X}'*, *headfmt=None*, *charmap='..... !"#$%&\\'()*+, -./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\\\]^_`abcdefghijklmnopqrstuvwxyz{|}~.....><'*, *emptystr=' --'*, *beforestr=' >>'*, *afterstr=' <<'*, *charsep=' |'*, *charend='|'*, *stream=Ellipsis*)

Textual hex dump.

This function generates a hex dump of the bytes within the specified range.

If *stream* is not `None`, the hex dump is written on it, otherwise it is returned as a `str`.

The default output is similar to that of `hexdump` or `xxd` commands, with some degree of tweaking. In case more customized formatting is desired, a dedicated custom function can be written by carefully looping over *values()*.

Parameters

- **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.
- **columns** (*int*) – Number of byte columns per row.
- **addrfmt** (*str*) – Address formatting string.
- **bytefmt** (*str*) – Byte formatting string.
- **headfmt** (*str*) – Header offset formatting string. If `Ellipsis`, it applies that of *bytefmt*. If `None`, no header row is generated.
- **charmap** (*mapping*) – Mapping to convert a byte integer into a string character. If `None`, no character data are appended to each row.

  The table is structured this way:

  – The initial 256 bytes map actual byte values.

  – Index `0x100` represents an empty byte (`None`).

  – Index `0x101` represents a byte before *start*.

  – Index `0x102` represents a byte after *endex*.
- **emptystr** (*str*) – Placeholder for an empty byte (`None` value).
- **beforestr** (*str*) – Placeholder for a byte before *bound_start*.
~~~

- **afterstr** (*str*) – Placeholder for a byte after *bound_endex*.
- **charsep** (*str*) – Separator between byte data and character data.
- **charend** (*str*) – Separator after character data.
- **stream** (*IO stream*) – Stream to write text onto. If `Ellipsis`, it uses `sys.stdout`. If not `None`, the function returns `None`.

Returns

str – Textual hex dump, if *stream* is `None`.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']], offset=0xDA7A)
>>> memory.hexdump()
0000DA7B  41 42 43 -- -- 78 79 7A -- -- -- -- -- -- -- --  |ABC  xyz        |
>>> memory.hexdump(stream=None)
'0000DA7B  41 42 43 -- -- 78 79 7A -- -- -- -- -- -- -- --  |ABC  xyz        |'
>>> memory.hexdump(start=0xDA7A, charmap=None)
0000DA7A  -- 41 42 43 -- -- 78 79 7A -- -- -- -- -- -- --
>>> memory.hexdump(start=0xDA7A)
0000DA7A  -- 41 42 43 -- -- 78 79 7A -- -- -- -- -- -- --  | ABC  xyz       |
>>> memory.hexdump(start=0xDA70)
0000DA70  -- -- -- -- -- -- -- -- -- -- -- 41 42 43 -- --  |           ABC  |
0000DA80  78 79 7A -- -- -- -- -- -- -- -- -- -- -- -- --  |xyz             |
>>> memory.bound_span = (0xDA78, 0xDA88)
>>> memory.hexdump(start=0xDA70)
0000DA70  >> >> >> >> >> >> >> >> -- -- -- 41 42 43 -- --  |>>>>>>>>   ABC  |
0000DA80  78 79 7A -- -- -- -- -- << << << << << << << <<  |xyz     <<<<<<<<|
>>> memory.hexdump(start=0xDA70, headfmt=...)
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0000DA70  >> >> >> >> >> >> >> >> -- -- -- 41 42 43 -- --  |>>>>>>>>   ABC  |
0000DA80  78 79 7A -- -- -- -- -- << << << << << << << <<  |xyz     <<<<<<<<|
>>> memory.hexdump(start=0xDA78, endex=0xDA84, columns=4)
0000DA78  -- -- -- 41  |   A|
0000DA7C  42 43 -- --  |BC  |
0000DA80  78 79 7A --  |xyz |
```

index(*item*, *start=None*, *endex=None*)

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.

See also:

find()

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*, *pattern=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse import Memory
```

```
>>> from itertools import islice
>>> memory = Memory()
>>> list(memory.keys())
[]
>>> list(memory.keys(endex=8))
```

(continues on next page)
~~~

(continued from previous page)

```
[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]
```

peek(*address*)

Gets the item at an address.

Returns

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

Examples

```
>>> from bytesparse 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 bytesparse 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=None*, *default=None*)

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 bytesparse 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=None*)

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 bytesparse 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 or byte*) – Item to restore.

See also:

popitem() popitem_backup()

read(*address*, *size*)

Reads data.

Reads a chunk of data from an address, with a given size. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.
- **size** (*int*) – Chunk size.

Returns
memoryview – A view over the addressed chunk.

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

Examples

```
>>> from bytesparse 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.read(2, 3))
b'BCD'
>>> bytes(memory.read(9, 1))
b'y'
>>> memory.read(4, 3)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.read(0, 6)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

readinto(*address*, *buffer*)

Reads data into a pre-allocated buffer.

Provided a pre-allocated writable buffer (*e.g.* a bytearray or a memoryview slice of it), this method reads a chunk of data from an address, with the size of the target buffer. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.
- **buffer** (*writable*) – Pre-allocated buffer to fill with data.

Returns
int – Number of bytes read.

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

Examples

```
>>> from bytesparse 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']])
>>> buffer = bytearray(3)
>>> memory.readinto(2, buffer)
3
>>> buffer
bytearray(b'BCD')
>>> view = memoryview(buffer)
>>> memory.readinto(9, view[1:2])
1
>>> buffer
bytearray(b'ByD')
>>> memory.readinto(4, buffer)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.readinto(0, bytearray(6))
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

remove(*item*, *start=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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 bytesparse 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=None*, *endex=None*)

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.

Warning: If the memory allows negative addresses, *rindex()* is more appropriate, because it raises `ValueError` if the item is not found.
~~~

See also:

rindex()

rindex(*item*, *start=None*, *endex=None*)

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.

Warning: If the memory allows negative addresses, *index()* is more appropriate, because it raises `ValueError` if the item is not found.

See also:

rfind()

rvalues(*start=None*, *endex=None*, *pattern=None*)

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 bytesparse 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))
```

(continues on next page)

(continued from previous page)

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

setdefault(*address*, *default=None*)

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 bytesparse 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 bytesparse 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()

property span: Tuple[int, int]

Memory address span.

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

Examples

```
>>> from bytesparse 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

property start: int

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 *bound_start* not None, that is returned.

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

Examples

```
>>> from bytesparse 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=None*, *endex=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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'
```

update(*data*, *clear=False*, ***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 bytesparse 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=False*, ***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=None*, *endex=None*, *pattern=None*)

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 bytesparse 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=None*, *endex=None*)

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 bytesparse 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=False*)

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 bytesparse 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=False*)

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()*

3.4 bytesparse.io

Streaming utilities.

Attributes

SEEK_SET	Seek from the beginning of the stream (position 0).
SEEK_CUR	Seek from the current stream position (*MemoryIO.tell()*).
SEEK_END	Seek from the end of the underlying memory object (`ImmutableMemory.endex()`).
SEEK_DATA	Seek to the next data block.
SEEK_HOLE	Seek to the next memory hole.

3.4.1 SEEK_SET

bytesparse.io.**SEEK_SET: int = 0**

Seek from the beginning of the stream (position 0).

3.4.2 SEEK_CUR

bytesparse.io.**SEEK_CUR: int = 1**

Seek from the current stream position (*MemoryIO.tell()*).

3.4.3 SEEK_END

bytesparse.io.**SEEK_END: int = 2**

Seek from the end of the underlying memory object (`ImmutableMemory.endex()`).

3.4.4 SEEK_DATA

bytesparse.io.**SEEK_DATA: int = 3**

Seek to the next data block.

3.4.5 SEEK_HOLE

bytesparse.io.**SEEK_HOLE: int = 4**

Seek to the next memory hole.

Classes

MemoryIO	Buffered I/O wrapper.

3.4.6 MemoryIO

class bytesparse.io.**MemoryIO**(*memory=None*, *seek=None*)

Buffered I/O wrapper.

This class wraps an `ImmutableMemory` object so that the latter can be accessed like a typical Python I/O read-only stream.

A memory stream is writable if, on its creation (*__init__()*), its underlying memory object can be written an empty byte string at its start address, without raising any exceptions.

Any operations executed on a closed stream may fail, raising an exception.

The stream position (the result of *tell()*) indicates the address currently pointed by the stream. It is just a number, and as such it is allowed to fall outside the actual memory bounds.

The stream position always refers to absolute address 0; in no way it ever refers to the `ImmutableMemory.start` of the underlying wrapped memory object.

Parameters

- **memory** (`ImmutableMemory`) – The memory object to wrap. If `None`, it assigns a new empty `bytesparse.Memory`.
- **seek** (*int*) – If `Ellipsis`, *seek()* to `memory.start`. If not `None`, *seek()* to the absolute address *seek*.

Variables

- **_memory** (`ImmutableMemory`) – The underlying wrapped memory object. It is set to `None` when *closed*.
- **_position** (*int*) – The current stream position. It always refers to absolute address 0.
- **_writable** (*bool*) – The stream is writable on creation.

See also:

bytesparse.base.ImmutableMemory *bytesparse.base.MutableMemory* *seek()* *writable()*

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> stream = MemoryIO(seek=3)
>>> stream.write(b'Hello')
5
>>> str(stream.memory)
"<[[3, b'Hello']]>"
>>> stream.seek(10)
10
>>> stream.write(b'World!')
6
>>> str(stream.memory)
"<[[3, b'Hello'], [10, b'World!']]>"
```

```
>>> memory = Memory.from_bytes(b'Hello, World!')
>>> stream = MemoryIO(memory, seek=7)
>>> stream.read()
b'World!'
>>> stream.tell()
13
>>> stream.seek(-6, SEEK_END)
7
>>> stream.write(b'Human')
5
>>> bytes(memory)
b'Hello, Human!'
>>> stream.truncate(5)
5
>>> stream.seek(3, SEEK_CUR)
8
>>> stream.write(b'World')
5
>>> memory.to_blocks()
[[0, b'Hello'], [8, b'World']]
>>> stream.close()
```

```
>>> blocks = [[3, b'Hello'], [10, b'World!']]
>>> memory = Memory.from_blocks(blocks)
>>> stream = MemoryIO(memory, seek=...)
>>> stream.tell()
3
>>> stream.read()
b'Hello'
```

```
>>> blocks = [[3, b'Hello\nWorld!'], [20, b'Bye\n'], [28, b'Bye!']]
>>> with MemoryIO(Memory.from_blocks(blocks)) as stream:
...     lines = stream.readlines()
>>> lines
[b'Hello\n', b'World!', b'Bye\n', b'Bye!']
>>> stream.seek(0)
Traceback (most recent call last):
    ...
ValueError: I/O operation on closed stream.
```

Attributes

closed	Closed stream.
memory	Underlying memory object.

Methods

__init__	
close	Closes the stream.
detach	Detaches the underlying raw stream.
fileno	File descriptor identifier.
flush	Flushes buffered data into the underlying raw steam.
getbuffer	Memory view of the underlying memory object.
getvalue	Byte string copy of the underlying memory object.
isatty	Interactive console stream.
peek	Previews the next chunk of bytes.
read	Reads a chunk of bytes.
read1	Reads a chunk of bytes.
readable	Stream is readable.
readinto	Reads data into a byte buffer.
readinto1	Reads data into a byte buffer.
readline	Reads a line.
readlines	Reads a list of lines.
seek	Changes the current stream position.
seekable	Stream is seekable.
skip_data	Skips a data block.
skip_hole	Skips a memory hole.
tell	Current stream position.
truncate	Truncates stream.
writable	Stream is writable.
write	Writes data into the stream.
writelines	Writes lines to the stream.

__del__()

Prepares the object for destruction.

It makes sure the stream is closed upon object destruction.

__enter__()

Context manager enter function.

Returns

MemoryIO – The stream object itself.

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> with MemoryIO(Memory.from_bytes(b'Hello, World!')) as stream:
...     data = stream.read()
>>> data
b'Hello, World!'
```

__exit__(*exc_type*, *exc_val*, *exc_tb*)

Context manager exit function.

It makes sure the stream is closed upon context exit.

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> with MemoryIO(Memory.from_bytes(b'Hello, World!')) as stream:
...     print(stream.closed)
False
>>> print(stream.closed)
True
```

__init__(*memory=None*, *seek=None*)

__iter__()

Iterates over lines.

Repeatedly calls *readline()*, as long as it returns byte strings. Yields the values returned by such calls.

Yields
bytes – Single line; terminator included.

See also:

readline()

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> blocks = [[3, b'Hello\nWorld!'], [20, b'Bye\n'], [28, b'Bye!']]
>>> with MemoryIO(Memory.from_blocks(blocks)) as stream:
...     lines = [line for line in stream]
>>> lines
[b'Hello\n', b'World!', b'Bye\n', b'Bye!']
```

__new__(***kwargs*)

__next__()

Next iterated line.

Calls *readline()* once, returning the value.

Returns
bytes or int – Line read from the stream, or the negative gap size.

See also:

readline()

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> blocks = [[3, b'Hello\nWorld!'], [20, b'Bye\n'], [28, b'Bye!']]
>>> with MemoryIO(Memory.from_blocks(blocks), seek=9) as stream:
...     print(next(stream))
b'World!'
```

_check_closed()

Checks if the stream is closed.

In case the stream is *closed*, it raises `ValueError`.

Raises
ValueError – The stream is closed.

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> with MemoryIO(Memory.from_bytes(b'ABC')) as stream:
...     stream._check_closed()
>>> stream._check_closed()
Traceback (most recent call last):
    ...
ValueError: I/O operation on closed stream.
```

close()

Closes the stream.

Any subsequent operations on the closed stream may fail, and some properties may change state.

The stream no more links to an underlying memory object.

See also:

closed

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> stream = MemoryIO(Memory.from_bytes(b'ABC'))
>>> stream.closed
False
>>> stream.memory is None
False
>>> stream.readable()
True
>>> stream.close()
>>> stream.closed
True
>>> stream.memory is None
True
>>> stream.readable()
Traceback (most recent call last):
    ...
ValueError: I/O operation on closed stream.
```

property closed: bool

Closed stream.

Returns

bool – Closed stream.

See also:

close()

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> stream = MemoryIO(Memory.from_bytes(b'ABC'))
>>> stream.closed
False
>>> stream.close()
>>> stream.closed
True
```

```
>>> with MemoryIO(Memory.from_bytes(b'ABC')) as stream:
...     print(stream.closed)
False
>>> print(stream.closed)
True
```

detach()

Detaches the underlying raw stream.

> **Warning:** It always raises `io.UnsupportedOperation`. This method is present only for API compatibility. No actual underlying stream is present for this object.

Raises
io.UnsupportedOperation – No underlying raw stream.

`fileno()`
File descriptor identifier.

> **Warning:** It always raises `io.UnsupportedOperation`. This method is present only for API compatibility. No actual file descriptor is associated to this object.

Raises
OSError – Not a file stream.

`flush()`
Flushes buffered data into the underlying raw steam.

Notes

Since no underlying stream is associated, this method does nothing.

`getbuffer()`
Memory view of the underlying memory object.

> **Warning:** This method may fail when the underlying memory object has gaps within data.

Returns
memoryview – Memory view over the underlying memory object.

See also:

`ImmutableMemory.view()`

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> with MemoryIO(Memory.from_bytes(b'Hello, World!')) as stream:
...     with stream.getbuffer() as buffer:
...         print(type(buffer), '=', bytes(buffer))
<class 'memoryview'> = b'Hello, World!'
```

```
>>> blocks = [[3, b'Hello'], [10, b'World!']]
>>> with MemoryIO(Memory.from_blocks(blocks)) as stream:
...     stream.getbuffer()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
ValueError: non-contiguous data within range
```

getvalue()

Byte string copy of the underlying memory object.

> **Warning:** This method may fail when the underlying memory object has gaps within data.

Returns

bytes – Byte string copy of the underlying memory object.

See also:

`ImmutableMemory.to_bytes()`

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> with MemoryIO(Memory.from_bytes(b'Hello, World!')) as stream:
...     value = stream.getvalue()
...     print(type(value), '=', bytes(value))
<class 'bytes'> = b'Hello, World!'
```

```
>>> blocks = [[3, b'Hello'], [10, b'World!']]
>>> with MemoryIO(Memory.from_blocks(blocks)) as stream:
...     stream.getvalue()
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

isatty()

Interactive console stream.

Returns

bool – `False`, not an interactive console stream.

property memory: *ImmutableMemory* **| None**

Underlying memory object.

`None` when *closed*.

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> memory = Memory.from_bytes(b'Hello, World!')
>>> with MemoryIO(memory) as stream:
...     print(stream.memory is memory)
True
>>> print(stream.memory is memory)
False
>>> print(stream.memory is None)
True
```

Type
ImmutableMemory

peek(*size=0*, *asmemview=False*)

Previews the next chunk of bytes.

Similar to *read()*, without moving the stream position instead. This method can be used to preview the next chunk of bytes, without affecting the stream itself.

The number of returned bytes may be different from *size*, which acts as a mere hint.

If the current stream position lies within a memory gap, this method returns the negative amount of bytes to reach the next data block.

If the current stream position is after the end of memory data, this method returns an empty byte string.

Parameters

- **size** (*int*) – Number of bytes to read. If negative or `None`, read as many bytes as possible.
- **asmemview** (*bool*) – Return a `memoryview` instead of `bytes`.

Returns
bytes – Chunk of bytes.

See also:

read()

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> blocks = [[3, b'Hello'], [10, b'World!']]
>>> memory = Memory.from_blocks(blocks)
>>> stream = MemoryIO(memory, seek=4)
>>> stream.peek()
b''
>>> stream.peek(1)
b'e'
>>> stream.peek(11)
b'ello'
```

(continues on next page)

(continued from previous page)

```
>>> stream.peek(None)
b'ello'
>>> stream.tell()
4
>>> memview = stream.peek(-1, asmemview=True)
>>> type(memview)
<class 'memoryview'>
>>> bytes(memview)
b'ello'
>>> stream.seek(8)
8
>>> stream.peek()
-2
```

read(*size=-1*, *asmemview=False*)

Reads a chunk of bytes.

Starting from the current stream position, this method tries to read up to *size* bytes (or as much as possible if negative or `None`).

The number of bytes can be less than *size* in the case a memory hole or the end are encountered.

If the current stream position lies within a memory gap, this method returns the negative amount of bytes to reach the next data block.

If the current stream position is after the end of memory data, this method returns an empty byte string.

Parameters

- **size** (*int*) – Number of bytes to read. If negative or `None`, read as many bytes as possible.
- **asmemview** (*bool*) – Return a `memoryview` instead of `bytes`.

Returns

bytes – Chunk of up to *size* bytes.

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> blocks = [[3, b'Hello'], [10, b'World!']]
>>> memory = Memory.from_blocks(blocks)
>>> stream = MemoryIO(memory, seek=4)
>>> stream.read(1)
b'e'
>>> stream.tell()
5
>>> stream.read(99)
b'llo'
>>> stream.tell()
8
>>> stream.read()
-2
>>> stream.tell()
```

(continues on next page)

(continued from previous page)

```
10
>>> memview = stream.read(None, asmemview=True)
>>> type(memview)
<class 'memoryview'>
>>> bytes(memview)
b'World!'
>>> stream.tell()
16
>>> stream.read()
b''
>>> stream.tell()
16
```

read1(*size=-1*, *asmemview=False*)

Reads a chunk of bytes.

Starting from the current stream position, this method tries to read up to *size* bytes (or as much as possible if negative or `None`).

The number of bytes can be less than *size* in the case a memory hole or the end are encountered.

If the current stream position lies within a memory gap, this method returns the negative amount of bytes to reach the next data block.

If the current stream position is after the end of memory data, this method returns an empty byte string.

Parameters

- **size** (*int*) – Number of bytes to read. If negative or `None`, read as many bytes as possible.
- **asmemview** (*bool*) – Return a `memoryview` instead of `bytes`.

Returns

bytes – Chunk of up to *size* bytes.

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> blocks = [[3, b'Hello'], [10, b'World!']]
>>> memory = Memory.from_blocks(blocks)
>>> stream = MemoryIO(memory, seek=4)
>>> stream.read(1)
b'e'
>>> stream.tell()
5
>>> stream.read(99)
b'llo'
>>> stream.tell()
8
>>> stream.read()
-2
>>> stream.tell()
10
```

(continues on next page)

(continued from previous page)

```
>>> memview = stream.read(None, asmemview=True)
>>> type(memview)
<class 'memoryview'>
>>> bytes(memview)
b'World!'
>>> stream.tell()
16
>>> stream.read()
b''
>>> stream.tell()
16
```

readable()

Stream is readable.

Returns

bool – `True`, stream is always readable.

readinto(*buffer*, *skipgaps=True*)

Reads data into a byte buffer.

If the stream is pointing after the memory end, no bytes are read.

If pointing within a memory hole (gap), the negative number of bytes until the next data block is returned. The stream is always positioned after the gap.

If a memory hole (gap) is encountered after reading some bytes, the reading stops there, and the number of bytes read is returned. The stream is always positioned after the gap.

Standard operation reads data until *buffer* is full, or encountering the memory end. It returns the number of bytes read.

Parameters

- **buffer** (*bytearray*) – A pre-allocated byte array to fill with bytes read from the stream.
- **skipgaps** (*bool*) – If false, it stops reading when a memory hole (gap) is encountered.

Returns

int – Number of bytes read, or the negative gap size.

Examples

```
>>> from bytesparse import Memory, MemoryIO
>>> blocks = [[3, b'Hello'], [10, b'World!']]
>>> memory = Memory.from_blocks(blocks)
```

```
>>> stream = MemoryIO(memory, seek=4)
>>> buffer = bytearray(b'.' * 8)
>>> stream.readinto(buffer, skipgaps=True)
8
>>> buffer
bytearray(b'elloWorl')
>>> stream.tell()
14
```

(continues on next page)

(continued from previous page)

```
>>> stream.readinto(buffer, skipgaps=True)
2
>>> buffer
bytearray(b'd!loWorl')
>>> stream.tell()
16
>>> stream.readinto(buffer, skipgaps=True)
0
>>> buffer
bytearray(b'd!loWorl')
>>> stream.tell()
16
```

```
>>> stream = MemoryIO(memory, seek=4)
>>> buffer = bytearray(b'.' * 8)
>>> stream.readinto(buffer, skipgaps=False)
4
>>> buffer
bytearray(b'ello....')
>>> stream.tell()
8
>>> stream.readinto(buffer, skipgaps=False)
-2
>>> stream.tell()
10
>>> stream.readinto(buffer, skipgaps=False)
6
>>> buffer
bytearray(b'World!..')
>>> stream.tell()
16
>>> stream.readinto(buffer, skipgaps=False)
0
>>> buffer
bytearray(b'World!..')
>>> stream.tell()
16
```

readinto1(*buffer*, *skipgaps=True*)

Reads data into a byte buffer.

If the stream is pointing after the memory end, no bytes are read.

If pointing within a memory hole (gap), the negative number of bytes until the next data block is returned. The stream is always positioned after the gap.

If a memory hole (gap) is encountered after reading some bytes, the reading stops there, and the number of bytes read is returned. The stream is always positioned after the gap.

Standard operation reads data until *buffer* is full, or encountering the memory end. It returns the number of bytes read.

Parameters

- **buffer** (*bytearray*) – A pre-allocated byte array to fill with bytes read from the stream.

- **skipgaps** (*bool*) – If false, it stops reading when a memory hole (gap) is encountered.

Returns

int – Number of bytes read, or the negative gap size.

Examples

```
>>> from bytesparse import Memory, MemoryIO
>>> blocks = [[3, b'Hello'], [10, b'World!']]
>>> memory = Memory.from_blocks(blocks)
```

```
>>> stream = MemoryIO(memory, seek=4)
>>> buffer = bytearray(b'.' * 8)
>>> stream.readinto(buffer, skipgaps=True)
8
>>> buffer
bytearray(b'elloWorl')
>>> stream.tell()
14
>>> stream.readinto(buffer, skipgaps=True)
2
>>> buffer
bytearray(b'd!loWorl')
>>> stream.tell()
16
>>> stream.readinto(buffer, skipgaps=True)
0
>>> buffer
bytearray(b'd!loWorl')
>>> stream.tell()
16
```

```
>>> stream = MemoryIO(memory, seek=4)
>>> buffer = bytearray(b'.' * 8)
>>> stream.readinto(buffer, skipgaps=False)
4
>>> buffer
bytearray(b'ello....')
>>> stream.tell()
8
>>> stream.readinto(buffer, skipgaps=False)
-2
>>> stream.tell()
10
>>> stream.readinto(buffer, skipgaps=False)
6
>>> buffer
bytearray(b'World!..')
>>> stream.tell()
16
>>> stream.readinto(buffer, skipgaps=False)
0
```

(continues on next page)

(continued from previous page)

```
>>> buffer
bytearray(b'World!..')
>>> stream.tell()
16
```

readline(*size=-1*, *skipgaps=True*, *asmemview=False*)

Reads a line.

A standard line is a sequence of bytes terminating with a `b'\n'` newline character.

If *size* is provided (not `None` nor negative), the current line ends there, without a trailing newline character.

If the stream is pointing after the memory end, an empty byte string is returned.

If a memory hole (gap) is encountered, the current line ends there without a trailing newline character. The stream is always positioned after the gap.

If the stream points within a memory hole, it returns the negative number of bytes until the next data block. The stream is always positioned after the gap.

Parameters

- **size** (*int*) – Maximum number of bytes for the line to read. If `None` or negative, no limit is set.
- **skipgaps** (*bool*) – If false, the negative size of the pointed memory hole.
- **asmemview** (*bool*) – If true, the returned object is a `memoryview` instead of `bytes`.

Returns

bytes or int – Line read from the stream, or the negative gap size.

See also:

read() readlines()

Examples

```
>>> from bytesparse import Memory, MemoryIO
>>> blocks = [[3, b'Hello\nWorld!'], [20, b'Bye\n'], [28, b'Bye!']]
```

```
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> stream.readline()
b'Hello\n'
>>> stream.tell()
9
>>> stream.readline(None)
b'World!'
>>> stream.tell()
15
>>> stream.readline(99)
b'Bye\n'
>>> stream.tell()
24
>>> stream.readline(99)
b'Bye!'
```

(continues on next page)

(continued from previous page)

```
>>> stream.tell()
32
>>> stream.readline()
b''
>>> stream.tell()
32
```

```
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> stream.readline(4)
b'Hell'
>>> stream.tell()
7
>>> stream.readline(4)
b'o\n'
>>> stream.tell()
9
```

```
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> view = stream.readline(asmemview=True)
>>> type(view) is memoryview
True
>>> bytes(view)
b'Hello\n'
```

```
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> # Emulating stream.readlines(skipgaps=False)
>>> lines = []
>>> line = True
>>> while line:
...     line = stream.readline(skipgaps=False)
...     lines.append(line)
>>> lines
[-3, b'Hello\n', b'World!', -5, b'Bye\n', -4, b'Bye!']
>>> stream.tell()
32
>>> stream.readline(skipgaps=False)
b''
>>> stream.tell()
32
```

readlines(*hint=-1*, *skipgaps=True*, *asmemview=False*)

Reads a list of lines.

It repeatedly calls *readline()*, collecting the returned values into a list, until the total number of bytes read reaches *hint*.

If a memory hole (gap) is encountered, the current line ends there without a trailing newline character, and the stream is positioned after the gap.

If *skipgaps* is false, the list is appended the negative size of each encountered memory hole.

Parameters

- **hint** (*int*) – Number of bytes after which line reading stops. If `None` or negative, no limit

is set.

- **skipgaps** (*bool*) – If false, the list hosts the negative size of each memory hole.
- **asmemview** (*bool*) – If true, the returned objects are memory views instead of byte strings.

Returns

list of bytes or int – List of lines and gaps read from the stream.

See also:

__iter__() read() readline()

Examples

```
>>> from bytesparse import Memory, MemoryIO
>>> blocks = [[3, b'Hello\nWorld!'], [20, b'Bye\n'], [28, b'Bye!']]
```

```
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> stream.readlines()
[b'Hello\n', b'World!', b'Bye\n', b'Bye!']
>>> stream.tell()
32
>>> stream.readlines()
[]
>>> stream.tell()
32
```

```
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> stream.readlines(hint=10)
[b'Hello\n', b'World!']
>>> stream.tell()
15
```

```
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> views = stream.readlines(asmemview=True)
>>> all(type(view) is memoryview for view in views)
True
>>> [bytes(view) for view in views]
[b'Hello\n', b'World!', b'Bye\n', b'Bye!']
```

```
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> stream.readlines(skipgaps=False)
[-3, b'Hello\n', b'World!', -5, b'Bye\n', -4, b'Bye!']
>>> stream.tell()
32
>>> stream.readlines(skipgaps=False)
[]
>>> stream.tell()
32
```

seek(*offset*, *whence=0*)

Changes the current stream position.

It performs the classic `seek()` I/O operation.

The *whence* can be any of:

- ***SEEK_SET*** **(0 or None):**
  referring to the absolute address 0.
- ***SEEK_CUR*** **(1):**
  referring to the current stream position (*tell()*).
- ***SEEK_END*** **(2):**
  referring to the memory end (`ImmutableMemory.endex`).
- ***SEEK_DATA*** **(3):**
  if the current stream position lies within a memory hole, it moves to the beginning of the next data block; no operation is performed otherwise.
- ***SEEK_HOLE*** **(4):**
  if the current stream position lies within a data block, it moves to the beginning of the next memory hole (note: the end of the stream is considered as a memory hole); no operation is performed otherwise.

Parameters

- **offset** (*int*) – Position offset to apply.
- **whence** (*int*) – Where the offset is referred. It can be any of the standard `SEEK_*` values. By default, it refers to the beginning of the stream.

Returns

int – The updated stream position.

Notes

Stream position is just a number, not related to memory ranges.

Examples

```
>>> from bytesparse import *
```

```
>>> blocks = [[3, b'Hello'], [12, b'World!']]
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> stream.seek(5)
5
>>> stream.seek(-3, SEEK_END)
15
>>> stream.seek(2, SEEK_CUR)
17
>>> stream.seek(1, SEEK_SET)
1
>>> stream.seek(stream.tell(), SEEK_HOLE)
1
>>> stream.seek(stream.tell(), SEEK_DATA)
3
>>> stream.seek(stream.tell(), SEEK_HOLE)
8
>>> stream.seek(stream.tell(), SEEK_DATA)
```

(continues on next page)

(continued from previous page)

```
12
>>> stream.seek(stream.tell(), SEEK_HOLE)  # EOF
18
>>> stream.seek(stream.tell(), SEEK_DATA)  # EOF
18
>>> stream.seek(22)  # after
22
>>> stream.seek(0)  # before
0
```

seekable()

Stream is seekable.

Returns

bool – `True`, stream is always seekable.

skip_data()

Skips a data block.

It moves the current stream position after the end of the currently pointed data block.

No action is performed if the current stream position lies within a memory hole (gap).

Returns

int – Updated stream position.

See also:

seek()

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> blocks = [[3, b'Hello'], [12, b'World!']]
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> stream.skip_data()
0
>>> stream.seek(6)
6
>>> stream.skip_data()
8
>>> stream.skip_data()
8
>>> stream.seek(12)
12
>>> stream.skip_data()
18
>>> stream.skip_data()
18
>>> stream.seek(20)
20
>>> stream.skip_data()
20
```

skip_hole()

Skips a memory hole.

It moves the current stream position after the end of the currently pointed memory hole (gap).

No action is performed if the current stream position lies within a data block.

Returns

int – Updated stream position.

See also:

seek()

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> blocks = [[3, b'Hello'], [12, b'World!']]
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> stream.skip_hole()
3
>>> stream.skip_hole()
3
>>> stream.seek(9)
9
>>> stream.skip_hole()
12
>>> stream.skip_hole()
12
>>> stream.seek(20)
20
>>> stream.skip_hole()
20
```

tell()

Current stream position.

Returns

int – Current stream position.

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> blocks = [[3, b'Hello'], [12, b'World!']]
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> stream.tell()
0
>>> stream.skip_hole()
3
>>> stream.tell()
3
```

(continues on next page)

(continued from previous page)

```
>>> stream.read(5)
b'Hello'
>>> stream.tell()
8
>>> stream.skip_hole()
12
>>> stream.read()
b'World!'
>>> stream.tell()
18
>>> stream.seek(20)
20
>>> stream.tell()
20
```

truncate(*size=None*)

Truncates stream.

If *size* is provided, it moves the current stream position to it.

Any data after the updated stream position are deleted from the underlying memory object.

The updated stream position can lie outside the actual memory bounds (i.e. extending after the memory). No filling is performed, only the stream position is moved there.

Parameters
size (*int*) – If not `None`, the stream is positioned there.

Returns
int – Updated stream position.

Raises
io.UnsupportedOperation – Stream not writable.

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> blocks = [[3, b'Hello'], [12, b'World!']]
>>> stream = MemoryIO(Memory.from_blocks(blocks))
>>> stream.seek(7)
7
>>> stream.truncate()
7
>>> stream.tell()
7
>>> stream.memory.to_blocks()
[[3, b'Hell']]
>>> stream.truncate(10)
10
>>> stream.tell()
10
```

```
>>> memory = Memory.from_bytes(b'Hello, World!')
>>> setattr(memory, 'write', None)  # exception on write()
>>> stream = MemoryIO(memory)
>>> stream.seek(7)
7
>>> stream.truncate()
Traceback (most recent call last):
    ...
io.UnsupportedOperation: truncate
```

writable()

Stream is writable.

Returns

bool – Stream is writable.

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> memory = Memory.from_bytes(b'Hello, World!')
>>> with MemoryIO(memory) as stream:
...     print(stream.writable())
True
>>> setattr(memory, 'write', None)  # exception on write()
>>> with MemoryIO(memory) as stream:
...     print(stream.writable())
False
```

write(*buffer*)

Writes data into the stream.

The behaviour depends on the nature of *buffer*: byte-like or integer.

Byte-like data are written into the underlying memory object via its *bytesparse.base.MutableMemory.write()* method, at the current stream position (i.e. *tell()*). The stream position is always incremented by the size of *buffer*, regardless of the actual number of bytes written into the underlying memory object (e.g. when cropped by existing `bytesparse.base.MutableMemory.bounds_span` settings).

If *buffer* is a positive integer, that is the amount of bytes to *bytesparse.base.MutableMemory.clear()* from the current stream position onwards. The stream position is incremented by *buffer* bytes. It returns *buffer* as a positive number.

If *buffer* is a negative integer, that is the amount of bytes to *bytesparse.base.MutableMemory.delete()* from the current stream position onwards. The stream position is not changed. It returns *buffer* as a positive number.

Notes

buffer is considered an integer if the execution of `buffer.__index__()` does not raise an `Exception`.

Parameters
buffer (*bytes*) – Byte data to write at the current stream position.

Returns
int – Size of the written *buffer*.

Raises
io.UnsupportedOperation – Stream not writable.

See also:

bytesparse.base.MutableMemory.clear() *bytesparse.base.MutableMemory.delete()*
bytesparse.base.MutableMemory.write()

Examples

```
>>> from bytesparse import Memory, MemoryIO
```

```
>>> blocks = [[3, b'Hello'], [10, b'World!']]
>>> memory = Memory.from_blocks(blocks)
>>> stream = MemoryIO(memory, seek=10)
>>> stream.write(b'Human')
5
>>> memory.to_blocks()
[[3, b'Hello'], [10, b'Human!']]
>>> stream.tell()
15
>>> stream.seek(7)
7
>>> stream.write(5)  # clear 5 bytes
5
>>> memory.to_blocks()
[[3, b'Hell'], [12, b'man!']]
>>> stream.tell()
12
>>> stream.seek(7)
7
>>> stream.write(-5)  # delete 5 bytes
5
>>> memory.to_blocks()
[[3, b'Hellman!']]
>>> stream.tell()
7
```

```
>>> memory = Memory.from_bytes(b'Hello, World!')
>>> setattr(memory, 'write', None)  # exception on write()
>>> stream = MemoryIO(memory, seek=7)
>>> stream.write(b'Human')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
io.UnsupportedOperation: not writable
```

writelines(*lines*)

Writes lines to the stream.

Line separators are not added, so it is usual for each of the lines provided to have a line separator at the end.

If a *line* is an integer, its behavior is as per *write()* (positive: clear, negative: delete).

Parameters

lines (*list of bytes*) – List of byte strings to write.

See also:

bytesparse.base.MutableMemory.clear() *bytesparse.base.MutableMemory.delete()* *write()*

Examples

```
>>> from bytesparse import Memory, MemoryIO
>>> lines = [3, b'Hello\n', b'World!', 5, b'Bye\n', 4, b'Bye!']
>>> stream = MemoryIO()
>>> stream.writelines(lines)
>>> stream.memory.to_blocks()
[[3, b'Hello\nWorld!'], [20, b'Bye\n'], [28, b'Bye!']]
```

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

bytesparse could always use more documentation, whether as part of the official bytesparse 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/bytesparse/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 *bytesparse* for local development:

1. Fork bytesparse (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/bytesparse.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 1.0.0 (2024-03-07)

- Following the `major.minor.patch` semantic versioning.
- Added `align` methods.

6.2 0.1.0 (2024-02-22)

- Improved documentation.
- Version number deserved something more stable.

6.3 0.0.8 (2024-01-21)

- Added `chop` method.
- Minor fixes.

6.4 0.0.7 (2023-12-10)

- Added support for Python 3.12.
- Added `hexdump` method.
- Added `bytesparse.io` package.
- Added `bytesparse.MemoryIO` as a stream wrapper for `bytesparse.Memory`.

6.5 0.0.6 (2023-02-18)

- Added support to Python 3.11, removed 3.6.
- Added some minor features.
- Improved documentation.
- Improved testing.
- Improved repository layout (`pyproject.toml`).
- Minor fixes.

6.6 0.0.5 (2022-02-22)

- Added `bytesparse` class, closer to `bytearray` than `Memory`.
- Added missing abstract and ported methods.
- Added cut feature.
- Added more helper methods.
- Fixed values iteration.
- Improved extraction performance.
- Improved testing.

6.7 0.0.4 (2022-01-09)

- Refactored current implementation as the `inplace` sub-module.
- Added abstract base classes and base types into the `base` 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.

6.8 0.0.3 (2022-01-03)

- Using explicit factory methods instead of constructor arguments.
- Added block collapsing helper function.
- Minor fixes.
- Improved test suite.

6.9 0.0.2 (2021-12-27)

- Cython implementation moved to its own `cbytesparse` Python package.
- Remote testing moved to GitHub Actions.

6.10 0.0.1 (2021-04-04)

- First release on PyPI.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

M

P

R

S

T

U

V

W