

---

# **bytesparse**

***Release 1.0.0***

**Andrea Zoppi**

**Mar 07, 2024**



# CONTENTS

<b>1</b>	<b>Overview</b>	<b>1</b>
1.1	Objectives	1
1.2	Python implementation	2
1.3	Cython implementation	2
1.4	Examples	2
1.5	Background	6
1.6	Documentation	6
1.7	Installation	6
1.8	Development	7
<b>2</b>	<b>Installation</b>	<b>9</b>
2.1	From <i>PyPI</i>	9
2.2	From source	9
<b>3</b>	<b>Reference</b>	<b>11</b>
3.1	bytesparse	11
3.2	bytesparse.base	12
3.2.1	STR_MAX_CONTENT_SIZE	13
3.2.2	HUMAN_ASCII	13
3.2.3	ImmutableMemory	13
3.2.4	MutableBytesparse	62
3.2.5	MutableMemory	139
3.3	bytesparse.inplace	214
3.3.1	Memory	215
3.3.2	bytesparse	290
3.4	bytesparse.io	367
3.4.1	SEEK_SET	368
3.4.2	SEEK_CUR	368
3.4.3	SEEK_END	368
3.4.4	SEEK_DATA	368
3.4.5	SEEK_HOLE	368
3.4.6	MemoryIO	369
<b>4</b>	<b>Contributing</b>	<b>393</b>
4.1	Bug reports	393
4.2	Documentation improvements	393
4.3	Feature requests and feedback	393
4.4	Development	394
4.4.1	Pull Request Guidelines	394
4.4.2	Tips	394

<b>5</b>	<b>Authors</b>	<b>395</b>
<b>6</b>	<b>Changelog</b>	<b>397</b>
6.1	1.0.0 (2024-03-07) . . . . .	397
6.2	0.1.0 (2024-02-22) . . . . .	397
6.3	0.0.8 (2024-01-21) . . . . .	397
6.4	0.0.7 (2023-12-10) . . . . .	397
6.5	0.0.6 (2023-02-18) . . . . .	398
6.6	0.0.5 (2022-02-22) . . . . .	398
6.7	0.0.4 (2022-01-09) . . . . .	398
6.8	0.0.3 (2022-01-03) . . . . .	398
6.9	0.0.2 (2021-12-27) . . . . .	399
6.10	0.0.1 (2021-04-04) . . . . .	399
<b>7</b>	<b>Indices and tables</b>	<b>401</b>
	<b>Python Module Index</b>	<b>403</b>
	<b>Index</b>	<b>405</b>

## 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 `byparsparse` package provides the following virtual memory types:

- `byparsparse.Memory`, a generic virtual memory with infinite address range.
- `byparsparse.byparsparse`, 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 `cbytparsparse` Python package for more details.

## 1.4 Examples

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

```
>>> from bytparsparse import Memory
>>> from bytparsparse import bytparsparse
```

```
>>> m = bytparsparse(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, endx=(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 = bytparse(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 = byparsparse.from_blocks([[4, b'ABC'], [9, b'xy']], start=2, end=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, `byparsparse` 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://byparsparse.readthedocs.io/>

## 1.7 Installation

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

```
$ pip install byparsparse
```

From the source code root directory:

```
$ pip install .
```

## 1.8 Development

To run the all the tests:

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



## 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 .
```



## REFERENCE

<code>bytesparse</code>	Utilities for sparse blocks of bytes.
<code>bytesparse.base</code>	Common stuff, shared across modules.
<code>bytesparse.inplace</code>	In-place implementation.
<code>bytesparse.io</code>	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 byparsparse.base

Common stuff, shared across modules.

### Attributes

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



### 3.2.1 STR\_MAX\_CONTENT\_SIZE

`bytestparse.base.STR_MAX_CONTENT_SIZE: int = 1000`

Maximum memory content size for string representation.

### 3.2.2 HUMAN\_ASCII

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

Mapping from byte to human-readable ASCII characters.

## Classes

<code>ImmutableMemory</code>	Immutable virtual memory.
<code>MutableBytestparse</code>	Wrapper for more <i>bytearray</i> compatibility.
<code>MutableMemory</code>	Mutable virtual memory.

### 3.2.3 ImmutableMemory

`class bytestparse.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 bytestparse 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_endx()` `_block_index_start()` `block_span()` `bound()` `bound_endx` `bound_span` `bound_start` `content_endx` `content_endin` `content_parts` `content_size` `content_span` `content_start` `count` `endx` `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

<code>bound_endx</code>	Bounds exclusive end address.
<code>bound_span</code>	Bounds span addresses.
<code>bound_start</code>	Bounds start address.
<code>content_endx</code>	Exclusive content end address.
<code>content_endin</code>	Inclusive content end address.
<code>content_parts</code>	Number of blocks.
<code>content_size</code>	Actual content size.
<code>content_span</code>	Memory content address span.
<code>content_start</code>	Inclusive content start address.
<code>contiguous</code>	Contains contiguous data.
<code>endx</code>	Exclusive end address.
<code>endin</code>	Inclusive end address.
<code>span</code>	Memory address span.
<code>start</code>	Inclusive start address.

## Methods

<code>__init__</code>	
<code>block_span</code>	Span of block data.
<code>blocks</code>	Iterates over blocks.
<code>bound</code>	Bounds addresses.
<code>chop</code>	Iterates over chopped blocks.
<code>collapse_blocks</code>	Collapses a generic sequence of blocks.
<code>content_blocks</code>	Iterates over blocks.
<code>content_items</code>	Iterates over content address and value pairs.
<code>content_keys</code>	Iterates over content addresses.
<code>content_values</code>	Iterates over content values.
<code>copy</code>	Creates a deep copy.
<code>count</code>	Counts items.
<code>equal_span</code>	Span of homogeneous data.
<code>extract</code>	Selects items from a range.
<code>find</code>	Index of an item.
<code>from_blocks</code>	Creates a virtual memory from blocks.
<code>from_bytes</code>	Creates a virtual memory from a byte-like chunk.
<code>from_items</code>	Creates a virtual memory from a iterable address/byte mapping.
<code>from_memory</code>	Creates a virtual memory from another one.
<code>from_values</code>	Creates a virtual memory from a byte-like sequence.
<code>fromhex</code>	Creates a virtual memory from an hexadecimal string.
<code>gaps</code>	Iterates over block gaps.
<code>get</code>	Gets the item at an address.
<code>hex</code>	Converts into an hexadecimal string.
<code>hexdump</code>	Textual hex dump.
<code>index</code>	Index of an item.
<code>intervals</code>	Iterates over block intervals.
<code>items</code>	Iterates over address and value pairs.
<code>keys</code>	Iterates over addresses.
<code>peek</code>	Gets the item at an address.
<code>read</code>	Reads data.
<code>readinto</code>	Reads data into a pre-allocated buffer.
<code>rfind</code>	Index of an item, reversed search.
<code>rindex</code>	Index of an item, reversed search.
<code>rvalues</code>	Iterates over values, reversed order.
<code>to_blocks</code>	Exports into blocks.
<code>to_bytes</code>	Exports into bytes.
<code>validate</code>	Validates internal structure.
<code>values</code>	Iterates over values.
<code>view</code>	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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byteparse 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 byteparse 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 bytestparse 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 bytestparse 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	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\_endx(*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_endx(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 bytestparse 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 bytestparse 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, end=None*)

Iterates over blocks.

Iterates over data blocks within an address range.

#### Parameters

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

Bounds addresses.

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

In case of stored data, *content\_start* and *content\_end* are used as bounds.

In case of bounds limits, *bound\_start* or *bound\_end* are used as bounds, when not *None*.

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

#### Returns

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

## Examples

```
>>> from bytestparse 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 bytestparse 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 bytestparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_span = (7, 13)
>>> memory.to_blocks()
[[7, b'ello, 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'ello, 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 bytestparse 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 bytestparse 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 bytestparse 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\_end=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\_end** (*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 bytestparse 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_end=-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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byteparse 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, end=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 bytestparse 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 bytestparse 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 maximum 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 bytestparse 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 maximum 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byparsе 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 byparsе 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 empty 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, end=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 bytestparse 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*. Implements 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byparse import Memory
```

```
>>> Memory().span
(0, 0)
>>> Memory(start=1, end=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 byparse 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 bytestparse 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*. Implements 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 bytestparse 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 bytestparse 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 MutableBytestparse

**class** bytestparse.base.MutableBytestparse(\*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 *MutableBytestparse.\_\_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  $0 \leq x < 256$ , 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 = bytparse('Non-ASCII: \u2204', 'ascii', 'replace')
>>> memory.to_blocks()
[[0, b'Non-ASCII: ?']]
```

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

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

### Method Groups

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

### Attributes

<code>bound_endex</code>	Bounds exclusive end address.
<code>bound_span</code>	Bounds span addresses.
<code>bound_start</code>	Bounds start address.
<code>content_endex</code>	Exclusive content end address.
<code>content_endin</code>	Inclusive content end address.
<code>content_parts</code>	Number of blocks.
<code>content_size</code>	Actual content size.
<code>content_span</code>	Memory content address span.
<code>content_start</code>	Inclusive content start address.
<code>contiguous</code>	Contains contiguous data.
<code>endex</code>	Exclusive end address.
<code>endin</code>	Inclusive end address.
<code>span</code>	Memory address span.
<code>start</code>	Inclusive start address.

## Methods

<code>__init__</code>	
<code>align</code>	Floods blocks to align their boundaries.
<code>align_backup</code>	Backups an <i>align()</i> operation.
<code>align_restore</code>	Restores an <i>align()</i> operation.
<code>append</code>	Appends a single item.
<code>append_backup</code>	Backups an <i>append()</i> operation.
<code>append_restore</code>	Restores an <i>append()</i> operation.
<code>block_span</code>	Span of block data.
<code>blocks</code>	Iterates over blocks.
<code>bound</code>	Bounds addresses.
<code>chop</code>	Iterates over chopped blocks.
<code>clear</code>	Clears an address range.
<code>clear_backup</code>	Backups a <i>clear()</i> operation.
<code>clear_restore</code>	Restores a <i>clear()</i> operation.
<code>collapse_blocks</code>	Collapses a generic sequence of blocks.
<code>content_blocks</code>	Iterates over blocks.
<code>content_items</code>	Iterates over content address and value pairs.
<code>content_keys</code>	Iterates over content addresses.
<code>content_values</code>	Iterates over content values.
<code>copy</code>	Creates a deep copy.
<code>count</code>	Counts items.
<code>crop</code>	Keeps data within an address range.
<code>crop_backup</code>	Backups a <i>crop()</i> operation.
<code>crop_restore</code>	Restores a <i>crop()</i> operation.
<code>cut</code>	Cuts a slice of memory.
<code>delete</code>	Deletes an address range.
<code>delete_backup</code>	Backups a <i>delete()</i> operation.
<code>delete_restore</code>	Restores a <i>delete()</i> operation.
<code>equal_span</code>	Span of homogeneous data.
<code>extend</code>	Concatenates items.
<code>extend_backup</code>	Backups an <i>extend()</i> operation.
<code>extend_restore</code>	Restores an <i>extend()</i> operation.
<code>extract</code>	Selects items from a range.
<code>fill</code>	Overwrites a range with a pattern.
<code>fill_backup</code>	Backups a <i>fill()</i> operation.
<code>fill_restore</code>	Restores a <i>fill()</i> operation.
<code>find</code>	Index of an item.
<code>flood</code>	Fills emptiness between non-touching blocks.
<code>flood_backup</code>	Backups a <i>flood()</i> operation.
<code>flood_restore</code>	Restores a <i>flood()</i> operation.
<code>from_blocks</code>	Creates a virtual memory from blocks.
<code>from_bytes</code>	Creates a virtual memory from a byte-like chunk.
<code>from_items</code>	Creates a virtual memory from a iterable address/byte mapping.
<code>from_memory</code>	Creates a virtual memory from another one.
<code>from_values</code>	Creates a virtual memory from a byte-like sequence.
<code>fromhex</code>	Creates a virtual memory from an hexadecimal string.
<code>gaps</code>	Iterates over block gaps.
<code>get</code>	Gets the item at an address.

continues on next page

Table 2 – continued from previous page

<code>hex</code>	Converts into an hexadecimal string.
<code>hexdump</code>	Textual hex dump.
<code>index</code>	Index of an item.
<code>insert</code>	Inserts data.
<code>insert_backup</code>	Backups an <code>insert()</code> operation.
<code>insert_restore</code>	Restores an <code>insert()</code> operation.
<code>intervals</code>	Iterates over block intervals.
<code>items</code>	Iterates over address and value pairs.
<code>keys</code>	Iterates over addresses.
<code>peek</code>	Gets the item at an address.
<code>poke</code>	Sets the item at an address.
<code>poke_backup</code>	Backups a <code>poke()</code> operation.
<code>poke_restore</code>	Restores a <code>poke()</code> operation.
<code>pop</code>	Takes a value away.
<code>pop_backup</code>	Backups a <code>pop()</code> operation.
<code>pop_restore</code>	Restores a <code>pop()</code> operation.
<code>popitem</code>	Pops the last item.
<code>popitem_backup</code>	Backups a <code>popitem()</code> operation.
<code>popitem_restore</code>	Restores a <code>popitem()</code> operation.
<code>read</code>	Reads data.
<code>readinto</code>	Reads data into a pre-allocated buffer.
<code>remove</code>	Removes an item.
<code>remove_backup</code>	Backups a <code>remove()</code> operation.
<code>remove_restore</code>	Restores a <code>remove()</code> operation.
<code>reserve</code>	Inserts emptiness.
<code>reserve_backup</code>	Backups a <code>reserve()</code> operation.
<code>reserve_restore</code>	Restores a <code>reserve()</code> operation.
<code>reverse</code>	Reverses the memory in-place.
<code>rfind</code>	Index of an item, reversed search.
<code>rindex</code>	Index of an item, reversed search.
<code>rvalues</code>	Iterates over values, reversed order.
<code>setdefault</code>	Defaults a value.
<code>setdefault_backup</code>	Backups a <code>setdefault()</code> operation.
<code>setdefault_restore</code>	Restores a <code>setdefault()</code> operation.
<code>shift</code>	Shifts the items.
<code>shift_backup</code>	Backups a <code>shift()</code> operation.
<code>shift_restore</code>	Restores an <code>shift()</code> operation.
<code>to_blocks</code>	Exports into blocks.
<code>to_bytes</code>	Exports into bytes.
<code>update</code>	Updates data.
<code>update_backup</code>	Backups an <code>update()</code> operation.
<code>update_restore</code>	Restores an <code>update()</code> operation.
<code>validate</code>	Validates internal structure.
<code>values</code>	Iterates over values.
<code>view</code>	Creates a view over a range.
<code>write</code>	Writes data.
<code>write_backup</code>	Backups a <code>write()</code> operation.
<code>write_restore</code>	Restores a <code>write()</code> 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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_endx`(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 bytestparse 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 bytestparse 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, endx)`

Rectifies an address span.

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

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

#### Returns

*pair of int* – Rectified address span.

**abstract** `align(modulo, start=None, endx=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.
- **endx** (*int*) – Exclusive end address for flooding. If *None*, *endx* 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'ABBBB'], [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, end=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'ello, 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'ello, 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 byparse 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\_end=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\_end** (*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 bytestparse 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_end=-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 bytestparse 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 bytestparse 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 bytestparse import Memory
```

```
>>> Memory().content_start
0
>>> Memory(start=1).content_start
1
>>> Memory(start=1, end=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 bytestparse 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 bytestparse 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 byparsparse 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 byparsparse 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 bytestparse 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 bytestparse 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 maximum 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 bytestparse 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 maximum 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byparse 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 bytestparse import Memory
```

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

~~~

```
>>> memory = Memory.fromhex('48656C6C6F2C20576F7226C6421')
>>> 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 bytestparse 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 bytestparse 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 empty 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 bytestparse 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*. Implements 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byteparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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  |
|   |   |   |   |   |   |   |   |    |    |    |

```
>>> 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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** `bytestparse.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 byteparse 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

<code>bound_endex</code>	Bounds exclusive end address.
<code>bound_span</code>	Bounds span addresses.
<code>bound_start</code>	Bounds start address.
<code>content_endex</code>	Exclusive content end address.
<code>content_endin</code>	Inclusive content end address.
<code>content_parts</code>	Number of blocks.
<code>content_size</code>	Actual content size.
<code>content_span</code>	Memory content address span.
<code>content_start</code>	Inclusive content start address.
<code>contiguous</code>	Contains contiguous data.
<code>endex</code>	Exclusive end address.
<code>endin</code>	Inclusive end address.
<code>span</code>	Memory address span.
<code>start</code>	Inclusive start address.

## Methods

<code>__init__</code>	
<code>align</code>	Floods blocks to align their boundaries.
<code>align_backup</code>	Backups an <i>align()</i> operation.
<code>align_restore</code>	Restores an <i>align()</i> operation.
<code>append</code>	Appends a single item.
<code>append_backup</code>	Backups an <i>append()</i> operation.
<code>append_restore</code>	Restores an <i>append()</i> operation.
<code>block_span</code>	Span of block data.
<code>blocks</code>	Iterates over blocks.
<code>bound</code>	Bounds addresses.
<code>chop</code>	Iterates over chopped blocks.
<code>clear</code>	Clears an address range.
<code>clear_backup</code>	Backups a <i>clear()</i> operation.
<code>clear_restore</code>	Restores a <i>clear()</i> operation.
<code>collapse_blocks</code>	Collapses a generic sequence of blocks.
<code>content_blocks</code>	Iterates over blocks.
<code>content_items</code>	Iterates over content address and value pairs.
<code>content_keys</code>	Iterates over content addresses.
<code>content_values</code>	Iterates over content values.
<code>copy</code>	Creates a deep copy.
<code>count</code>	Counts items.
<code>crop</code>	Keeps data within an address range.
<code>crop_backup</code>	Backups a <i>crop()</i> operation.
<code>crop_restore</code>	Restores a <i>crop()</i> operation.
<code>cut</code>	Cuts a slice of memory.
<code>delete</code>	Deletes an address range.
<code>delete_backup</code>	Backups a <i>delete()</i> operation.
<code>delete_restore</code>	Restores a <i>delete()</i> operation.
<code>equal_span</code>	Span of homogeneous data.
<code>extend</code>	Concatenates items.

continues on next page

Table 3 – continued from previous page

<i>extend_backup</i>	Backups an <i>extend()</i> operation.
<i>extend_restore</i>	Restores an <i>extend()</i> operation.
<i>extract</i>	Selects items from a range.
<i>fill</i>	Overwrites a range with a pattern.
<i>fill_backup</i>	Backups a <i>fill()</i> operation.
<i>fill_restore</i>	Restores a <i>fill()</i> operation.
<i>find</i>	Index of an item.
<i>flood</i>	Fills emptiness between non-touching blocks.
<i>flood_backup</i>	Backups a <i>flood()</i> operation.
<i>flood_restore</i>	Restores a <i>flood()</i> operation.
<i>from_blocks</i>	Creates a virtual memory from blocks.
<i>from_bytes</i>	Creates a virtual memory from a byte-like chunk.
<i>from_items</i>	Creates a virtual memory from an iterable address/byte mapping.
<i>from_memory</i>	Creates a virtual memory from another one.
<i>from_values</i>	Creates a virtual memory from a byte-like sequence.
<i>fromhex</i>	Creates a virtual memory from a hexadecimal string.
<i>gaps</i>	Iterates over block gaps.
<i>get</i>	Gets the item at an address.
<i>hex</i>	Converts into a hexadecimal string.
<i>hexdump</i>	Textual hex dump.
<i>index</i>	Index of an item.
<i>insert</i>	Inserts data.
<i>insert_backup</i>	Backups an <i>insert()</i> operation.
<i>insert_restore</i>	Restores an <i>insert()</i> operation.
<i>intervals</i>	Iterates over block intervals.
<i>items</i>	Iterates over address and value pairs.
<i>keys</i>	Iterates over addresses.
<i>peek</i>	Gets the item at an address.
<i>poke</i>	Sets the item at an address.
<i>poke_backup</i>	Backups a <i>poke()</i> operation.
<i>poke_restore</i>	Restores a <i>poke()</i> operation.
<i>pop</i>	Takes a value away.
<i>pop_backup</i>	Backups a <i>pop()</i> operation.
<i>pop_restore</i>	Restores a <i>pop()</i> operation.
<i>popitem</i>	Pops the last item.
<i>popitem_backup</i>	Backups a <i>popitem()</i> operation.
<i>popitem_restore</i>	Restores a <i>popitem()</i> operation.
<i>read</i>	Reads data.
<i>readinto</i>	Reads data into a pre-allocated buffer.
<i>remove</i>	Removes an item.
<i>remove_backup</i>	Backups a <i>remove()</i> operation.
<i>remove_restore</i>	Restores a <i>remove()</i> operation.
<i>reserve</i>	Inserts emptiness.
<i>reserve_backup</i>	Backups a <i>reserve()</i> operation.
<i>reserve_restore</i>	Restores a <i>reserve()</i> operation.
<i>reverse</i>	Reverses the memory in-place.
<i>rfind</i>	Index of an item, reversed search.
<i>rindex</i>	Index of an item, reversed search.
<i>rvalues</i>	Iterates over values, reversed order.
<i>setdefault</i>	Defaults a value.
<i>setdefault_backup</i>	Backups a <i>setdefault()</i> operation.

continues on next page

Table 3 – continued from previous page

<code>setdefault_restore</code>	Restores a <code>setdefault()</code> operation.
<code>shift</code>	Shifts the items.
<code>shift_backup</code>	Backups a <code>shift()</code> operation.
<code>shift_restore</code>	Restores an <code>shift()</code> operation.
<code>to_blocks</code>	Exports into blocks.
<code>to_bytes</code>	Exports into bytes.
<code>update</code>	Updates data.
<code>update_backup</code>	Backups an <code>update()</code> operation.
<code>update_restore</code>	Restores an <code>update()</code> operation.
<code>validate</code>	Validates internal structure.
<code>values</code>	Iterates over values.
<code>view</code>	Creates a view over a range.
<code>write</code>	Writes data.
<code>write_backup</code>	Backups a <code>write()</code> operation.
<code>write_restore</code>	Restores a <code>write()</code> operation.

**abstract** `__add__(value)`

Concatenates items.

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

**See also:**

`MutableMemory.__iadd__()`

## Examples

```
>>> from bytestparse 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 bytestparse 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 bytestparse 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, end=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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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		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\_endx(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_endx(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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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, end=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\_end:** int | None

Bounds exclusive end address.

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

## Examples

```
>>> from byparsparse import Memory
```

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

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, end=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 bytestparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_span = (7, 13)
>>> memory.to_blocks()
[[7, b'ello, 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'ello, 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 bytestparse 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 bytestparse 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 bytestparse 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, endx=None*)

Backups a *clear()* operation.

### Parameters

- **start** (*int*) – Inclusive start address for clearing. If *None*, *start* is considered.
- **endx** (*int*) – Exclusive end address for clearing. If *None*, *endx* 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 bytestparse 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\_end=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\_end** (*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 bytestparse 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_end=-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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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]								)))

1

Actual content size.

*int* – The sum of all block lengths.

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

①

①

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

6

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

3

Memory content address span.

A tuple holding both `content_start` and `content_end`.

## Examples

```
>>> from bytestparse 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 bytestparse 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 bytestparse 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 bytparse 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, end=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 bytparse import Memory
```

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

```
>>> memory1 = Memory(start=1, end=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 bytestparse 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, endx=None)

Deletes an address range.

#### Parameters

- **start** (int) – Inclusive start address for deletion. If None, **start** is considered.
- **endx** (int) – Exclusive end address for deletion. If None, **endx** 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, endx=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 maximum 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 bytestparse 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 maximum 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 empty 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 bytestparse 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*. Implements 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 bytestparse 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 bytestparse 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 bytestparse 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.peak(10) # -> ord('z') = 122
122
>>> memory.peak(0)
None
>>> memory.peak(7)
None
>>> memory.peak(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 bytestparse 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 bytestparse 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 bytestparse 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, end=10)
>>> memory.reverse()
>>> memory.to_blocks()
[[5, b'CBA']]
```

### abstract `rfind(item, start=None, end=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.
- **end** (*int*) – Exclusive end of the searched range. If *None*, *end* 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 bytestparse 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

```
>>> 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 byparse import Memory
```

```
>>> Memory().span
(0, 0)
>>> Memory(start=1, end=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 byparse 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 bytestparse 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 bytestparse 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 byparsparse 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 bytestparse 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 bytestparse.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

|                   |                                                  |
|-------------------|--------------------------------------------------|
| <i>Memory</i>     | Virtual memory.                                  |
| <i>bytesparse</i> | Wrapper for more <i>bytearray</i> 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

|                            |                                  |
|----------------------------|----------------------------------|
| <code>bound_endex</code>   | Bounds exclusive end address.    |
| <code>bound_span</code>    | Bounds span addresses.           |
| <code>bound_start</code>   | Bounds start address.            |
| <code>content_endex</code> | Exclusive content end address.   |
| <code>content_endin</code> | Inclusive content end address.   |
| <code>content_parts</code> | Number of blocks.                |
| <code>content_size</code>  | Actual content size.             |
| <code>content_span</code>  | Memory content address span.     |
| <code>content_start</code> | Inclusive content start address. |
| <code>contiguous</code>    | Contains contiguous data.        |
| <code>endex</code>         | Exclusive end address.           |
| <code>endin</code>         | Inclusive end address.           |
| <code>span</code>          | Memory address span.             |
| <code>start</code>         | Inclusive start address.         |

## Methods

|                              |                                                |
|------------------------------|------------------------------------------------|
| <code>__init__</code>        |                                                |
| <code>align</code>           | Floods blocks to align their boundaries.       |
| <code>align_backup</code>    | Backups an <code>align()</code> operation.     |
| <code>align_restore</code>   | Restores an <code>align()</code> operation.    |
| <code>append</code>          | Appends a single item.                         |
| <code>append_backup</code>   | Backups an <code>append()</code> operation.    |
| <code>append_restore</code>  | Restores an <code>append()</code> operation.   |
| <code>block_span</code>      | Span of block data.                            |
| <code>blocks</code>          | Iterates over blocks.                          |
| <code>bound</code>           | Bounds addresses.                              |
| <code>chop</code>            | Iterates over chopped blocks.                  |
| <code>clear</code>           | Clears an address range.                       |
| <code>clear_backup</code>    | Backups a <code>clear()</code> operation.      |
| <code>clear_restore</code>   | Restores a <code>clear()</code> operation.     |
| <code>collapse_blocks</code> | Collapses a generic sequence of blocks.        |
| <code>content_blocks</code>  | Iterates over blocks.                          |
| <code>content_items</code>   | Iterates over content address and value pairs. |
| <code>content_keys</code>    | Iterates over content addresses.               |
| <code>content_values</code>  | Iterates over content values.                  |
| <code>copy</code>            | Creates a deep copy.                           |
| <code>count</code>           | Counts items.                                  |
| <code>crop</code>            | Keeps data within an address range.            |
| <code>crop_backup</code>     | Backups a <code>crop()</code> operation.       |
| <code>crop_restore</code>    | Restores a <code>crop()</code> operation.      |
| <code>cut</code>             | Cuts a slice of memory.                        |
| <code>delete</code>          | Deletes an address range.                      |
| <code>delete_backup</code>   | Backups a <code>delete()</code> operation.     |
| <code>delete_restore</code>  | Restores a <code>delete()</code> operation.    |
| <code>equal_span</code>      | Span of homogeneous data.                      |
| <code>extend</code>          | Concatenates items.                            |

continues on next page

Table 4 – continued from previous page

|                          |                                                                 |
|--------------------------|-----------------------------------------------------------------|
| <i>extend_backup</i>     | Backups an <i>extend()</i> operation.                           |
| <i>extend_restore</i>    | Restores an <i>extend()</i> operation.                          |
| <i>extract</i>           | Selects items from a range.                                     |
| <i>fill</i>              | Overwrites a range with a pattern.                              |
| <i>fill_backup</i>       | Backups a <i>fill()</i> operation.                              |
| <i>fill_restore</i>      | Restores a <i>fill()</i> operation.                             |
| <i>find</i>              | Index of an item.                                               |
| <i>flood</i>             | Fills emptiness between non-touching blocks.                    |
| <i>flood_backup</i>      | Backups a <i>flood()</i> operation.                             |
| <i>flood_restore</i>     | Restores a <i>flood()</i> operation.                            |
| <i>from_blocks</i>       | Creates a virtual memory from blocks.                           |
| <i>from_bytes</i>        | Creates a virtual memory from a byte-like chunk.                |
| <i>from_items</i>        | Creates a virtual memory from an iterable address/byte mapping. |
| <i>from_memory</i>       | Creates a virtual memory from another one.                      |
| <i>from_values</i>       | Creates a virtual memory from a byte-like sequence.             |
| <i>fromhex</i>           | Creates a virtual memory from a hexadecimal string.             |
| <i>gaps</i>              | Iterates over block gaps.                                       |
| <i>get</i>               | Gets the item at an address.                                    |
| <i>hex</i>               | Converts into a hexadecimal string.                             |
| <i>hexdump</i>           | Textual hex dump.                                               |
| <i>index</i>             | Index of an item.                                               |
| <i>insert</i>            | Inserts data.                                                   |
| <i>insert_backup</i>     | Backups an <i>insert()</i> operation.                           |
| <i>insert_restore</i>    | Restores an <i>insert()</i> operation.                          |
| <i>intervals</i>         | Iterates over block intervals.                                  |
| <i>items</i>             | Iterates over address and value pairs.                          |
| <i>keys</i>              | Iterates over addresses.                                        |
| <i>peek</i>              | Gets the item at an address.                                    |
| <i>poke</i>              | Sets the item at an address.                                    |
| <i>poke_backup</i>       | Backups a <i>poke()</i> operation.                              |
| <i>poke_restore</i>      | Restores a <i>poke()</i> operation.                             |
| <i>pop</i>               | Takes a value away.                                             |
| <i>pop_backup</i>        | Backups a <i>pop()</i> operation.                               |
| <i>pop_restore</i>       | Restores a <i>pop()</i> operation.                              |
| <i>popitem</i>           | Pops the last item.                                             |
| <i>popitem_backup</i>    | Backups a <i>popitem()</i> operation.                           |
| <i>popitem_restore</i>   | Restores a <i>popitem()</i> operation.                          |
| <i>read</i>              | Reads data.                                                     |
| <i>readinto</i>          | Reads data into a pre-allocated buffer.                         |
| <i>remove</i>            | Removes an item.                                                |
| <i>remove_backup</i>     | Backups a <i>remove()</i> operation.                            |
| <i>remove_restore</i>    | Restores a <i>remove()</i> operation.                           |
| <i>reserve</i>           | Inserts emptiness.                                              |
| <i>reserve_backup</i>    | Backups a <i>reserve()</i> operation.                           |
| <i>reserve_restore</i>   | Restores a <i>reserve()</i> operation.                          |
| <i>reverse</i>           | Reverses the memory in-place.                                   |
| <i>rfind</i>             | Index of an item, reversed search.                              |
| <i>rindex</i>            | Index of an item, reversed search.                              |
| <i>rvalues</i>           | Iterates over values, reversed order.                           |
| <i>setdefault</i>        | Defaults a value.                                               |
| <i>setdefault_backup</i> | Backups a <i>setdefault()</i> operation.                        |

continues on next page

Table 4 – continued from previous page

|                                 |                                                 |
|---------------------------------|-------------------------------------------------|
| <code>setdefault_restore</code> | Restores a <code>setdefault()</code> operation. |
| <code>shift</code>              | Shifts the items.                               |
| <code>shift_backup</code>       | Backups a <code>shift()</code> operation.       |
| <code>shift_restore</code>      | Restores an <code>shift()</code> operation.     |
| <code>to_blocks</code>          | Exports into blocks.                            |
| <code>to_bytes</code>           | Exports into bytes.                             |
| <code>update</code>             | Updates data.                                   |
| <code>update_backup</code>      | Backups an <code>update()</code> operation.     |
| <code>update_restore</code>     | Restores an <code>update()</code> operation.    |
| <code>validate</code>           | Validates internal structure.                   |
| <code>values</code>             | Iterates over values.                           |
| <code>view</code>               | Creates a view over a range.                    |
| <code>write</code>              | Writes data.                                    |
| <code>write_backup</code>       | Backups a <code>write()</code> operation.       |
| <code>write_restore</code>      | Restores a <code>write()</code> operation.      |

### `__add__(value)`

Concatenates items.

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

**See also:**

`MutableMemory.__iadd__()`

### Examples

```
>>> from byparsparse 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, end=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 byparsparse 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 byparsparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byparsparse 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 byparsparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byparse 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 byparse 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  |   | 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\_endx**(*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_endx(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 bytestparse 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 bytestparse 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'ABBBB'], [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, end=None*)

Iterates over blocks.

Iterates over data blocks within an address range.

#### Parameters

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

#### Yields

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

**See also:**

[intervals\(\)](#) [to\\_blocks\(\)](#)

## Examples

```
>>> from bytestparse 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, end*)

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 bytestparse 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 bytestparse 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 bytestparse 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'ello, 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 bytestparse 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 bytearray 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, end=None*)

Clears an address range.

### Parameters

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

See also:

[clear\\_backup\(\)](#) [clear\\_restore\(\)](#)

## Examples

```
>>> from bytearray 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, end=None*)

Backups a *clear()* operation.

### Parameters

- **start** (*int*) – Inclusive start address for clearing. If None, **start** is considered.
- **end** (*int*) – Exclusive end address for clearing. If None, **end** 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 bytestparse 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\_end=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\_end** (*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 bytestparse 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_end=-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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byparse import Memory
```

```
>>> Memory().content_start
0
>>> Memory(start=1).content_start
1
>>> Memory(start=1, end=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 bytestparse 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 bytestparse 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, end=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 byparsе import Memory
```

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

```
>>> memory1 = Memory(start=1, end=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, end=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 bytestparse 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 bytestparse 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 maximum 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 maximum 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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, endx=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.
- **endx** (*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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 empty 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 bytestparse 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 bytestparse 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 bytestparse 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*. Implements 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 bytestparse 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*. Implements 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 bytestparse 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(endx=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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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

```
>>> 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 bytestparse 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 bytestparse 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 bytestparse 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 byparsparse 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 bytestparse 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 bytestparse

**class** `bytestparse.inplace.bytestparse(*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 `MutableBytestparse.__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  $0 \leq x < 256$ , 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 = bytparse([65, 66, 67])
>>> memory.to_blocks()
[[0, b'ABC']]
```

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

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

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

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

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

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

### Method Groups

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



## Attributes

|                            |                                  |
|----------------------------|----------------------------------|
| <code>bound_endex</code>   | Bounds exclusive end address.    |
| <code>bound_span</code>    | Bounds span addresses.           |
| <code>bound_start</code>   | Bounds start address.            |
| <code>content_endex</code> | Exclusive content end address.   |
| <code>content_endin</code> | Inclusive content end address.   |
| <code>content_parts</code> | Number of blocks.                |
| <code>content_size</code>  | Actual content size.             |
| <code>content_span</code>  | Memory content address span.     |
| <code>content_start</code> | Inclusive content start address. |
| <code>contiguous</code>    | Contains contiguous data.        |
| <code>endex</code>         | Exclusive end address.           |
| <code>endin</code>         | Inclusive end address.           |
| <code>span</code>          | Memory address span.             |
| <code>start</code>         | Inclusive start address.         |

## Methods

|                              |                                                |
|------------------------------|------------------------------------------------|
| <code>__init__</code>        |                                                |
| <code>align</code>           | Floods blocks to align their boundaries.       |
| <code>align_backup</code>    | Backups an <i>align()</i> operation.           |
| <code>align_restore</code>   | Restores an <i>align()</i> operation.          |
| <code>append</code>          | Appends a single item.                         |
| <code>append_backup</code>   | Backups an <i>append()</i> operation.          |
| <code>append_restore</code>  | Restores an <i>append()</i> operation.         |
| <code>block_span</code>      | Span of block data.                            |
| <code>blocks</code>          | Iterates over blocks.                          |
| <code>bound</code>           | Bounds addresses.                              |
| <code>chop</code>            | Iterates over chopped blocks.                  |
| <code>clear</code>           | Clears an address range.                       |
| <code>clear_backup</code>    | Backups a <i>clear()</i> operation.            |
| <code>clear_restore</code>   | Restores a <i>clear()</i> operation.           |
| <code>collapse_blocks</code> | Collapses a generic sequence of blocks.        |
| <code>content_blocks</code>  | Iterates over blocks.                          |
| <code>content_items</code>   | Iterates over content address and value pairs. |
| <code>content_keys</code>    | Iterates over content addresses.               |
| <code>content_values</code>  | Iterates over content values.                  |
| <code>copy</code>            | Creates a deep copy.                           |
| <code>count</code>           | Counts items.                                  |
| <code>crop</code>            | Keeps data within an address range.            |
| <code>crop_backup</code>     | Backups a <i>crop()</i> operation.             |
| <code>crop_restore</code>    | Restores a <i>crop()</i> operation.            |
| <code>cut</code>             | Cuts a slice of memory.                        |
| <code>delete</code>          | Deletes an address range.                      |
| <code>delete_backup</code>   | Backups a <i>delete()</i> operation.           |
| <code>delete_restore</code>  | Restores a <i>delete()</i> operation.          |
| <code>equal_span</code>      | Span of homogeneous data.                      |
| <code>extend</code>          | Concatenates items.                            |

continues on next page

Table 5 – continued from previous page

|                          |                                                                 |
|--------------------------|-----------------------------------------------------------------|
| <i>extend_backup</i>     | Backups an <i>extend()</i> operation.                           |
| <i>extend_restore</i>    | Restores an <i>extend()</i> operation.                          |
| <i>extract</i>           | Selects items from a range.                                     |
| <i>fill</i>              | Overwrites a range with a pattern.                              |
| <i>fill_backup</i>       | Backups a <i>fill()</i> operation.                              |
| <i>fill_restore</i>      | Restores a <i>fill()</i> operation.                             |
| <i>find</i>              | Index of an item.                                               |
| <i>flood</i>             | Fills emptiness between non-touching blocks.                    |
| <i>flood_backup</i>      | Backups a <i>flood()</i> operation.                             |
| <i>flood_restore</i>     | Restores a <i>flood()</i> operation.                            |
| <i>from_blocks</i>       | Creates a virtual memory from blocks.                           |
| <i>from_bytes</i>        | Creates a virtual memory from a byte-like chunk.                |
| <i>from_items</i>        | Creates a virtual memory from an iterable address/byte mapping. |
| <i>from_memory</i>       | Creates a virtual memory from another one.                      |
| <i>from_values</i>       | Creates a virtual memory from a byte-like sequence.             |
| <i>fromhex</i>           | Creates a virtual memory from a hexadecimal string.             |
| <i>gaps</i>              | Iterates over block gaps.                                       |
| <i>get</i>               | Gets the item at an address.                                    |
| <i>hex</i>               | Converts into an hexadecimal string.                            |
| <i>hexdump</i>           | Textual hex dump.                                               |
| <i>index</i>             | Index of an item.                                               |
| <i>insert</i>            | Inserts data.                                                   |
| <i>insert_backup</i>     | Backups an <i>insert()</i> operation.                           |
| <i>insert_restore</i>    | Restores an <i>insert()</i> operation.                          |
| <i>intervals</i>         | Iterates over block intervals.                                  |
| <i>items</i>             | Iterates over address and value pairs.                          |
| <i>keys</i>              | Iterates over addresses.                                        |
| <i>peek</i>              | Gets the item at an address.                                    |
| <i>poke</i>              | Sets the item at an address.                                    |
| <i>poke_backup</i>       | Backups a <i>poke()</i> operation.                              |
| <i>poke_restore</i>      | Restores a <i>poke()</i> operation.                             |
| <i>pop</i>               | Takes a value away.                                             |
| <i>pop_backup</i>        | Backups a <i>pop()</i> operation.                               |
| <i>pop_restore</i>       | Restores a <i>pop()</i> operation.                              |
| <i>popitem</i>           | Pops the last item.                                             |
| <i>popitem_backup</i>    | Backups a <i>popitem()</i> operation.                           |
| <i>popitem_restore</i>   | Restores a <i>popitem()</i> operation.                          |
| <i>read</i>              | Reads data.                                                     |
| <i>readinto</i>          | Reads data into a pre-allocated buffer.                         |
| <i>remove</i>            | Removes an item.                                                |
| <i>remove_backup</i>     | Backups a <i>remove()</i> operation.                            |
| <i>remove_restore</i>    | Restores a <i>remove()</i> operation.                           |
| <i>reserve</i>           | Inserts emptiness.                                              |
| <i>reserve_backup</i>    | Backups a <i>reserve()</i> operation.                           |
| <i>reserve_restore</i>   | Restores a <i>reserve()</i> operation.                          |
| <i>reverse</i>           | Reverses the memory in-place.                                   |
| <i>rfind</i>             | Index of an item, reversed search.                              |
| <i>rindex</i>            | Index of an item, reversed search.                              |
| <i>rvalues</i>           | Iterates over values, reversed order.                           |
| <i>setdefault</i>        | Defaults a value.                                               |
| <i>setdefault_backup</i> | Backups a <i>setdefault()</i> operation.                        |

continues on next page

Table 5 – continued from previous page

|                                 |                                                 |
|---------------------------------|-------------------------------------------------|
| <code>setdefault_restore</code> | Restores a <code>setdefault()</code> operation. |
| <code>shift</code>              | Shifts the items.                               |
| <code>shift_backup</code>       | Backups a <code>shift()</code> operation.       |
| <code>shift_restore</code>      | Restores an <code>shift()</code> operation.     |
| <code>to_blocks</code>          | Exports into blocks.                            |
| <code>to_bytes</code>           | Exports into bytes.                             |
| <code>update</code>             | Updates data.                                   |
| <code>update_backup</code>      | Backups an <code>update()</code> operation.     |
| <code>update_restore</code>     | Restores an <code>update()</code> operation.    |
| <code>validate</code>           | Validates internal structure.                   |
| <code>values</code>             | Iterates over values.                           |
| <code>view</code>               | Creates a view over a range.                    |
| <code>write</code>              | Writes data.                                    |
| <code>write_backup</code>       | Backups a <code>write()</code> operation.       |
| <code>write_restore</code>      | Restores a <code>write()</code> operation.      |

### `__add__(value)`

Concatenates items.

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

**See also:**

`MutableMemory.__iadd__()`

### Examples

```
>>> from bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byparsparse 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 byparsparse 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 bytestparse 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 bytestparse 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 bytestparse 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 | 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_endx(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_endx(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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byparsparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_span = (7, 13)
>>> memory.to_blocks()
[[7, b'ello, 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'ello, 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 byparsparse 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 bytestparse 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 bytestparse 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\_end=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\_end** (*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 bytestparse 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_end=-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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 maximum 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 bytestparse 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 maximum 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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, end=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.
- **end** (*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 byparsparse 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 bytestparse 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 bytestparse 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 bytestparse 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 empty 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 bytestparse 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 bytestparse 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*. Implements 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*. Implements 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byparse 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, end=10)
>>> memory.reverse()
>>> memory.to_blocks()
[[5, b'CBA']]
```

## rfind(item, start=None, end=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.
- **end** (*int*) – Exclusive end of the searched range. If *None*, [end](#) 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 bytestparse 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  |
|   |   |   |   |   |   |   |   |    |    |    |

```
>>> 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 bytestparse 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 bytestparse 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 bytestparse 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(end=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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse.io

Streaming utilities.

## Attributes

|                  |                                                                                       |
|------------------|---------------------------------------------------------------------------------------|
| <i>SEEK_SET</i>  | Seek from the beginning of the stream (position 0).                                   |
| <i>SEEK_CUR</i>  | Seek from the current stream position ( <i>MemoryIO.tell()</i> ).                     |
| <i>SEEK_END</i>  | Seek from the end of the underlying memory object ( <i>ImmutableMemory.endex()</i> ). |
| <i>SEEK_DATA</i> | Seek to the next data block.                                                          |
| <i>SEEK_HOLE</i> | Seek to the next memory hole.                                                         |

### 3.4.1 SEEK\_SET

`byparsparse.io.SEEK_SET: int = 0`  
 Seek from the beginning of the stream (position 0).

### 3.4.2 SEEK\_CUR

`byparsparse.io.SEEK_CUR: int = 1`  
 Seek from the current stream position (*MemoryIO.tell()*).

### 3.4.3 SEEK\_END

`byparsparse.io.SEEK_END: int = 2`  
 Seek from the end of the underlying memory object (*ImmutableMemory.endex()*).

### 3.4.4 SEEK\_DATA

`byparsparse.io.SEEK_DATA: int = 3`  
 Seek to the next data block.

### 3.4.5 SEEK\_HOLE

`byparsparse.io.SEEK_HOLE: int = 4`  
 Seek to the next memory hole.

## Classes

|                 |                       |
|-----------------|-----------------------|
| <i>MemoryIO</i> | Buffered I/O wrapper. |
|-----------------|-----------------------|

### 3.4.6 MemoryIO

**class** `bytestparse.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 `bytestparse.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:

`bytestparse.base.ImmutableMemory` `bytestparse.base.MutableMemory` `seek()` `writable()`

#### Examples

```
>>> from bytestparse 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

|                     |                           |
|---------------------|---------------------------|
| <code>closed</code> | Closed stream.            |
| <code>memory</code> | Underlying memory object. |

## Methods

|                         |                                                      |
|-------------------------|------------------------------------------------------|
| <code>__init__</code>   |                                                      |
| <code>close</code>      | Closes the stream.                                   |
| <code>detach</code>     | Detaches the underlying raw stream.                  |
| <code>fileno</code>     | File descriptor identifier.                          |
| <code>flush</code>      | Flushes buffered data into the underlying raw steam. |
| <code>getbuffer</code>  | Memory view of the underlying memory object.         |
| <code>getvalue</code>   | Byte string copy of the underlying memory object.    |
| <code>isatty</code>     | Interactive console stream.                          |
| <code>peek</code>       | Previews the next chunk of bytes.                    |
| <code>read</code>       | Reads a chunk of bytes.                              |
| <code>read1</code>      | Reads a chunk of bytes.                              |
| <code>readable</code>   | Stream is readable.                                  |
| <code>readinto</code>   | Reads data into a byte buffer.                       |
| <code>readinto1</code>  | Reads data into a byte buffer.                       |
| <code>readline</code>   | Reads a line.                                        |
| <code>readlines</code>  | Reads a list of lines.                               |
| <code>seek</code>       | Changes the current stream position.                 |
| <code>seekable</code>   | Stream is seekable.                                  |
| <code>skip_data</code>  | Skips a data block.                                  |
| <code>skip_hole</code>  | Skips a memory hole.                                 |
| <code>tell</code>       | Current stream position.                             |
| <code>truncate</code>   | Truncates stream.                                    |
| <code>writable</code>   | Stream is writable.                                  |
| <code>write</code>      | Writes data into the stream.                         |
| <code>writelines</code> | 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 bytestparse 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 byparse 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 bytestparse 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 bytestparse 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 byparse 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 bytestparse 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 byparsparse 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 bytestparse 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 byparsе 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 byparsе 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 bytestparse 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 bytestparse 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 `bytestparse.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 `bytestparse.base.MutableMemory.bounds_span` settings).

If *buffer* is a positive integer, that is the amount of bytes to `bytestparse.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 `bytestparse.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:

`bytestparse.base.MutableMemory.clear()`      `bytestparse.base.MutableMemory.delete()`  
`bytestparse.base.MutableMemory.write()`

## Examples

```
>>> from bytestparse 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:**

[bytestparse.base.MutableMemory.clear\(\)](#)      [bytestparse.base.MutableMemory.delete\(\)](#)  
[write\(\)](#)

### **Examples**

```
>>> from bytestparse 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!']]
```

## 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
```

## AUTHORS

- Andrea Zoppi - main developer - <https://github.com/TexZK>





## 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.



## INDICES AND TABLES

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



## PYTHON MODULE INDEX

### b

- `bytesparse`, 11
- `bytesparse.base`, 12
- `bytesparse.inplace`, 214
- `bytesparse.io`, 367





## Symbols

- `__add__()` (*bytesparse.base.ImmutableMemory method*), 15
- `__add__()` (*bytesparse.base.MutableBytesparse method*), 66
- `__add__()` (*bytesparse.base.MutableMemory method*), 143
- `__add__()` (*bytesparse.inplace.Memory method*), 218
- `__add__()` (*bytesparse.inplace.bytesparse method*), 295
- `__bool__()` (*bytesparse.base.ImmutableMemory method*), 16
- `__bool__()` (*bytesparse.base.MutableBytesparse method*), 67
- `__bool__()` (*bytesparse.base.MutableMemory method*), 143
- `__bool__()` (*bytesparse.inplace.Memory method*), 218
- `__bool__()` (*bytesparse.inplace.bytesparse method*), 295
- `__bytes__()` (*bytesparse.base.ImmutableMemory method*), 16
- `__bytes__()` (*bytesparse.base.MutableBytesparse method*), 67
- `__bytes__()` (*bytesparse.base.MutableMemory method*), 144
- `__bytes__()` (*bytesparse.inplace.Memory method*), 219
- `__bytes__()` (*bytesparse.inplace.bytesparse method*), 296
- `__class_getitem__()` (*bytesparse.base.ImmutableMemory class method*), 17
- `__class_getitem__()` (*bytesparse.base.MutableBytesparse class method*), 68
- `__class_getitem__()` (*bytesparse.base.MutableMemory class method*), 144
- `__class_getitem__()` (*bytesparse.inplace.Memory class method*), 219
- `__class_getitem__()` (*bytesparse.inplace.bytesparse class method*), 296
- `__contains__()` (*bytesparse.base.ImmutableMemory method*), 17
- `__contains__()` (*bytesparse.base.MutableBytesparse method*), 68
- `__contains__()` (*bytesparse.base.MutableMemory method*), 144
- `__contains__()` (*bytesparse.inplace.Memory method*), 219
- `__contains__()` (*bytesparse.inplace.bytesparse method*), 296
- `__copy__()` (*bytesparse.base.ImmutableMemory method*), 18
- `__copy__()` (*bytesparse.base.MutableBytesparse method*), 69
- `__copy__()` (*bytesparse.base.MutableMemory method*), 145
- `__copy__()` (*bytesparse.inplace.Memory method*), 220
- `__copy__()` (*bytesparse.inplace.bytesparse method*), 297
- `__deepcopy__()` (*bytesparse.base.ImmutableMemory method*), 18
- `__deepcopy__()` (*bytesparse.base.MutableBytesparse method*), 69
- `__deepcopy__()` (*bytesparse.base.MutableMemory method*), 145
- `__deepcopy__()` (*bytesparse.inplace.Memory method*), 220
- `__deepcopy__()` (*bytesparse.inplace.bytesparse method*), 297
- `__del__()` (*bytesparse.io.MemoryIO method*), 371
- `__delitem__()` (*bytesparse.base.MutableBytesparse method*), 69
- `__delitem__()` (*bytesparse.base.MutableMemory method*), 145
- `__delitem__()` (*bytesparse.inplace.Memory method*), 220
- `__delitem__()` (*bytesparse.inplace.bytesparse method*), 297
- `__enter__()` (*bytesparse.io.MemoryIO method*), 371
- `__eq__()` (*bytesparse.base.ImmutableMemory method*), 18
- `__eq__()` (*bytesparse.base.MutableBytesparse method*), 70
- `__eq__()` (*bytesparse.base.MutableMemory method*),

[146](#)  
[\\_\\_eq\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [221](#)  
[\\_\\_eq\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [298](#)  
[\\_\\_exit\\_\\_\(\)](#) (*bytesparse.io.MemoryIO* method), [372](#)  
[\\_\\_getitem\\_\\_\(\)](#) (*bytesparse.base.ImmutableMemory* method), [18](#)  
[\\_\\_getitem\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse* method), [70](#)  
[\\_\\_getitem\\_\\_\(\)](#) (*bytesparse.base.MutableMemory* method), [147](#)  
[\\_\\_getitem\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [222](#)  
[\\_\\_getitem\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [299](#)  
[\\_\\_hash\\_\\_](#) (*bytesparse.base.ImmutableMemory* attribute), [19](#)  
[\\_\\_hash\\_\\_](#) (*bytesparse.base.MutableBytesparse* attribute), [71](#)  
[\\_\\_hash\\_\\_](#) (*bytesparse.base.MutableMemory* attribute), [147](#)  
[\\_\\_hash\\_\\_](#) (*bytesparse.inplace.Memory* attribute), [222](#)  
[\\_\\_hash\\_\\_](#) (*bytesparse.inplace.bytesparse* attribute), [299](#)  
[\\_\\_iadd\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse* method), [71](#)  
[\\_\\_iadd\\_\\_\(\)](#) (*bytesparse.base.MutableMemory* method), [147](#)  
[\\_\\_iadd\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [222](#)  
[\\_\\_iadd\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [299](#)  
[\\_\\_imul\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse* method), [72](#)  
[\\_\\_imul\\_\\_\(\)](#) (*bytesparse.base.MutableMemory* method), [148](#)  
[\\_\\_imul\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [223](#)  
[\\_\\_imul\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [300](#)  
[\\_\\_init\\_\\_\(\)](#) (*bytesparse.base.ImmutableMemory* method), [19](#)  
[\\_\\_init\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse* method), [72](#)  
[\\_\\_init\\_\\_\(\)](#) (*bytesparse.base.MutableMemory* method), [148](#)  
[\\_\\_init\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [223](#)  
[\\_\\_init\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [300](#)  
[\\_\\_init\\_\\_\(\)](#) (*bytesparse.io.MemoryIO* method), [372](#)  
[\\_\\_ior\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse* method), [72](#)  
[\\_\\_ior\\_\\_\(\)](#) (*bytesparse.base.MutableMemory* method), [148](#)  
[\\_\\_ior\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [223](#)  
[\\_\\_ior\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [300](#)  
[\\_\\_iter\\_\\_\(\)](#) (*bytesparse.base.ImmutableMemory* method), [19](#)  
[\\_\\_iter\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse* method), [73](#)  
[\\_\\_iter\\_\\_\(\)](#) (*bytesparse.base.MutableMemory* method), [149](#)  
[\\_\\_iter\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [224](#)  
[\\_\\_iter\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [301](#)  
[\\_\\_iter\\_\\_\(\)](#) (*bytesparse.io.MemoryIO* method), [372](#)  
[\\_\\_len\\_\\_\(\)](#) (*bytesparse.base.ImmutableMemory* method), [20](#)  
[\\_\\_len\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse* method), [73](#)  
[\\_\\_len\\_\\_\(\)](#) (*bytesparse.base.MutableMemory* method), [149](#)  
[\\_\\_len\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [224](#)  
[\\_\\_len\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [301](#)  
[\\_\\_mul\\_\\_\(\)](#) (*bytesparse.base.ImmutableMemory* method), [20](#)  
[\\_\\_mul\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse* method), [74](#)  
[\\_\\_mul\\_\\_\(\)](#) (*bytesparse.base.MutableMemory* method), [150](#)  
[\\_\\_mul\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [225](#)  
[\\_\\_mul\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [302](#)  
[\\_\\_new\\_\\_\(\)](#) (*bytesparse.io.MemoryIO* method), [372](#)  
[\\_\\_next\\_\\_\(\)](#) (*bytesparse.io.MemoryIO* method), [372](#)  
[\\_\\_or\\_\\_\(\)](#) (*bytesparse.base.ImmutableMemory* method), [21](#)  
[\\_\\_or\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse* method), [74](#)  
[\\_\\_or\\_\\_\(\)](#) (*bytesparse.base.MutableMemory* method), [150](#)  
[\\_\\_or\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [225](#)  
[\\_\\_or\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [302](#)  
[\\_\\_repr\\_\\_\(\)](#) (*bytesparse.base.ImmutableMemory* method), [21](#)  
[\\_\\_repr\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse* method), [75](#)  
[\\_\\_repr\\_\\_\(\)](#) (*bytesparse.base.MutableMemory* method), [151](#)  
[\\_\\_repr\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [226](#)  
[\\_\\_repr\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [303](#)  
[\\_\\_reversed\\_\\_\(\)](#) (*bytesparse.base.ImmutableMemory* method), [21](#)  
[\\_\\_reversed\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse* method), [75](#)  
[\\_\\_reversed\\_\\_\(\)](#) (*bytesparse.base.MutableMemory* method), [151](#)  
[\\_\\_reversed\\_\\_\(\)](#) (*bytesparse.inplace.Memory* method), [226](#)  
[\\_\\_reversed\\_\\_\(\)](#) (*bytesparse.inplace.bytesparse* method), [303](#)  
[\\_\\_setitem\\_\\_\(\)](#) (*bytesparse.base.MutableBytesparse*

method), 75  
 \_\_setitem\_\_() (bytestparse.base.MutableMemory  
 method), 151  
 \_\_setitem\_\_() (bytestparse.inplace.Memory method),  
 226  
 \_\_setitem\_\_() (bytestparse.inplace.bytestparse method),  
 303  
 \_\_str\_\_() (bytestparse.base.ImmutableMemory  
 method), 22  
 \_\_str\_\_() (bytestparse.base.MutableBytestparse  
 method), 76  
 \_\_str\_\_() (bytestparse.base.MutableMemory method),  
 152  
 \_\_str\_\_() (bytestparse.inplace.Memory method), 227  
 \_\_str\_\_() (bytestparse.inplace.bytestparse method), 304  
 \_\_subclasshook\_\_() (bytes-  
 parse.base.ImmutableMemory class method),  
 22  
 \_\_subclasshook\_\_() (bytes-  
 parse.base.MutableBytestparse class method),  
 77  
 \_\_subclasshook\_\_() (bytes-  
 parse.base.MutableMemory class method),  
 153  
 \_\_subclasshook\_\_() (bytestparse.inplace.Memory  
 class method), 228  
 \_\_subclasshook\_\_() (bytestparse.inplace.bytestparse  
 class method), 305  
 \_\_weakref\_\_ (bytestparse.base.ImmutableMemory at-  
 tribute), 22  
 \_\_weakref\_\_ (bytestparse.base.MutableBytestparse at-  
 tribute), 77  
 \_\_weakref\_\_ (bytestparse.base.MutableMemory at-  
 tribute), 153  
 \_\_weakref\_\_ (bytestparse.inplace.Memory attribute),  
 228  
 \_\_weakref\_\_ (bytestparse.inplace.bytestparse attribute),  
 305  
 \_block\_index\_at() (bytes-  
 parse.base.ImmutableMemory method),  
 22  
 \_block\_index\_at() (bytes-  
 parse.base.MutableBytestparse method),  
 77  
 \_block\_index\_at() (bytestparse.base.MutableMemory  
 method), 153  
 \_block\_index\_at() (bytestparse.inplace.Memory  
 method), 228  
 \_block\_index\_at() (bytestparse.inplace.bytestparse  
 method), 305  
 \_block\_index\_end() (bytes-  
 parse.base.ImmutableMemory method),  
 23  
 \_block\_index\_end() (bytes-  
 parse.base.MutableBytestparse method),  
 77  
 \_block\_index\_end() (bytestparse.inplace.Memory  
 method), 153  
 \_block\_index\_end() (bytestparse.inplace.bytestparse  
 method), 228  
 \_block\_index\_end() (bytestparse.inplace.bytestparse  
 method), 305  
 \_block\_index\_start() (bytes-  
 parse.base.ImmutableMemory method),  
 23  
 \_block\_index\_start() (bytes-  
 parse.base.MutableBytestparse method),  
 78  
 \_block\_index\_start() (bytes-  
 parse.base.MutableMemory method), 154  
 \_block\_index\_start() (bytestparse.inplace.Memory  
 method), 229  
 \_block\_index\_start() (bytestparse.inplace.bytestparse  
 method), 306  
 \_check\_closed() (bytestparse.io.MemoryIO method),  
 373  
 \_erase() (bytestparse.inplace.Memory method), 229  
 \_erase() (bytestparse.inplace.bytestparse method), 306  
 \_place() (bytestparse.inplace.Memory method), 230  
 \_place() (bytestparse.inplace.bytestparse method), 307  
 \_prebound\_end() (bytes-  
 parse.base.MutableBytestparse method),  
 78  
 \_prebound\_end() (bytestparse.base.MutableMemory  
 method), 154  
 \_prebound\_end() (bytestparse.inplace.Memory  
 method), 230  
 \_prebound\_end() (bytestparse.inplace.bytestparse  
 method), 307  
 \_prebound\_end\_backup() (bytes-  
 parse.base.MutableBytestparse method),  
 79  
 \_prebound\_end\_backup() (bytes-  
 parse.base.MutableMemory method), 155  
 \_prebound\_end\_backup() (bytes-  
 parse.inplace.Memory method), 230  
 \_prebound\_end\_backup() (bytes-  
 parse.inplace.bytestparse method), 307  
 \_prebound\_start() (bytes-  
 parse.base.MutableBytestparse method),  
 79  
 \_prebound\_start() (bytestparse.base.MutableMemory  
 method), 155  
 \_prebound\_start() (bytestparse.inplace.Memory  
 method), 230  
 \_prebound\_start() (bytestparse.inplace.bytestparse  
 method), 307  
 \_prebound\_start\_backup() (bytes-

*parse.base.MutableBytesparse* method), 79  
*\_prebound\_start\_backup()* (*bytesparse.base.MutableMemory* method), 155  
*\_prebound\_start\_backup()* (*bytesparse.inplace.Memory* method), 231  
*\_prebound\_start\_backup()* (*bytesparse.inplace.bytesparse* method), 308  
*\_rectify\_address()* (*bytesparse.base.MutableBytesparse* method), 79  
*\_rectify\_address()* (*bytesparse.inplace.bytesparse* method), 308  
*\_rectify\_span()* (*bytesparse.base.MutableBytesparse* method), 80  
*\_rectify\_span()* (*bytesparse.inplace.bytesparse* method), 308

## A

*align()* (*bytesparse.base.MutableBytesparse* method), 80  
*align()* (*bytesparse.base.MutableMemory* method), 155  
*align()* (*bytesparse.inplace.bytesparse* method), 308  
*align()* (*bytesparse.inplace.Memory* method), 231  
*align\_backup()* (*bytesparse.base.MutableBytesparse* method), 81  
*align\_backup()* (*bytesparse.base.MutableMemory* method), 156  
*align\_backup()* (*bytesparse.inplace.bytesparse* method), 309  
*align\_backup()* (*bytesparse.inplace.Memory* method), 232  
*align\_restore()* (*bytesparse.base.MutableBytesparse* method), 81  
*align\_restore()* (*bytesparse.base.MutableMemory* method), 157  
*align\_restore()* (*bytesparse.inplace.bytesparse* method), 310  
*align\_restore()* (*bytesparse.inplace.Memory* method), 232  
*append()* (*bytesparse.base.MutableBytesparse* method), 81  
*append()* (*bytesparse.base.MutableMemory* method), 157  
*append()* (*bytesparse.inplace.bytesparse* method), 310  
*append()* (*bytesparse.inplace.Memory* method), 232  
*append\_backup()* (*bytesparse.base.MutableBytesparse* method), 82  
*append\_backup()* (*bytesparse.base.MutableMemory* method), 157  
*append\_backup()* (*bytesparse.inplace.bytesparse* method), 310  
*append\_backup()* (*bytesparse.inplace.Memory* method), 233

*append\_restore()* (*bytesparse.base.MutableBytesparse* method), 82  
*append\_restore()* (*bytesparse.base.MutableMemory* method), 158  
*append\_restore()* (*bytesparse.inplace.bytesparse* method), 310  
*append\_restore()* (*bytesparse.inplace.Memory* method), 233

## B

*block\_span()* (*bytesparse.base.ImmutableMemory* method), 24  
*block\_span()* (*bytesparse.base.MutableBytesparse* method), 82  
*block\_span()* (*bytesparse.base.MutableMemory* method), 158  
*block\_span()* (*bytesparse.inplace.bytesparse* method), 311  
*block\_span()* (*bytesparse.inplace.Memory* method), 233  
*blocks()* (*bytesparse.base.ImmutableMemory* method), 24  
*blocks()* (*bytesparse.base.MutableBytesparse* method), 83  
*blocks()* (*bytesparse.base.MutableMemory* method), 158  
*blocks()* (*bytesparse.inplace.bytesparse* method), 311  
*blocks()* (*bytesparse.inplace.Memory* method), 234  
*bound()* (*bytesparse.base.ImmutableMemory* method), 25  
*bound()* (*bytesparse.base.MutableBytesparse* method), 83  
*bound()* (*bytesparse.base.MutableMemory* method), 159  
*bound()* (*bytesparse.inplace.bytesparse* method), 312  
*bound()* (*bytesparse.inplace.Memory* method), 234  
*bound\_endex* (*bytesparse.base.ImmutableMemory* property), 26  
*bound\_endex* (*bytesparse.base.MutableBytesparse* property), 85  
*bound\_endex* (*bytesparse.base.MutableMemory* property), 160  
*bound\_endex* (*bytesparse.inplace.bytesparse* property), 313  
*bound\_endex* (*bytesparse.inplace.Memory* property), 236  
*bound\_span* (*bytesparse.base.ImmutableMemory* property), 27  
*bound\_span* (*bytesparse.base.MutableBytesparse* property), 85  
*bound\_span* (*bytesparse.base.MutableMemory* property), 161  
*bound\_span* (*bytesparse.inplace.bytesparse* property), 313

- bound\_span (*bytesparse.inplace.Memory* property), 236  
 bound\_start (*bytesparse.base.ImmutableMemory* property), 27  
 bound\_start (*bytesparse.base.MutableBytesparse* property), 86  
 bound\_start (*bytesparse.base.MutableMemory* property), 161  
 bound\_start (*bytesparse.inplace.bytesparse* property), 314  
 bound\_start (*bytesparse.inplace.Memory* property), 237  
 bytesparse  
     module, 11  
 bytesparse (class in *bytesparse.inplace*), 290  
 bytesparse.base  
     module, 12  
 bytesparse.inplace  
     module, 214  
 bytesparse.io  
     module, 367
- ## C
- chop() (*bytesparse.base.ImmutableMemory* method), 28  
 chop() (*bytesparse.base.MutableBytesparse* method), 86  
 chop() (*bytesparse.base.MutableMemory* method), 162  
 chop() (*bytesparse.inplace.bytesparse* method), 314  
 chop() (*bytesparse.inplace.Memory* method), 237  
 clear() (*bytesparse.base.MutableBytesparse* method), 87  
 clear() (*bytesparse.base.MutableMemory* method), 162  
 clear() (*bytesparse.inplace.bytesparse* method), 315  
 clear() (*bytesparse.inplace.Memory* method), 238  
 clear\_backup() (*bytesparse.base.MutableBytesparse* method), 87  
 clear\_backup() (*bytesparse.base.MutableMemory* method), 163  
 clear\_backup() (*bytesparse.inplace.bytesparse* method), 316  
 clear\_backup() (*bytesparse.inplace.Memory* method), 238  
 clear\_restore() (*bytesparse.base.MutableBytesparse* method), 88  
 clear\_restore() (*bytesparse.base.MutableMemory* method), 163  
 clear\_restore() (*bytesparse.inplace.bytesparse* method), 316  
 clear\_restore() (*bytesparse.inplace.Memory* method), 239  
 close() (*bytesparse.io.MemoryIO* method), 373  
 closed (*bytesparse.io.MemoryIO* property), 374  
 collapse\_blocks() (*bytesparse.base.ImmutableMemory* class method), 28  
 collapse\_blocks() (*bytesparse.base.MutableBytesparse* class method), 88  
 collapse\_blocks() (*bytesparse.base.MutableMemory* class method), 163  
 collapse\_blocks() (*bytesparse.inplace.bytesparse* class method), 316  
 collapse\_blocks() (*bytesparse.inplace.Memory* class method), 239  
 content\_blocks() (*bytesparse.base.ImmutableMemory* method), 30  
 content\_blocks() (*bytesparse.base.MutableBytesparse* method), 89  
 content\_blocks() (*bytesparse.base.MutableMemory* method), 164  
 content\_blocks() (*bytesparse.inplace.bytesparse* method), 317  
 content\_blocks() (*bytesparse.inplace.Memory* method), 240  
 content\_index (*bytesparse.base.ImmutableMemory* property), 30  
 content\_index (*bytesparse.base.MutableBytesparse* property), 90  
 content\_index (*bytesparse.base.MutableMemory* property), 165  
 content\_index (*bytesparse.inplace.bytesparse* property), 318  
 content\_index (*bytesparse.inplace.Memory* property), 241  
 content\_endin (*bytesparse.base.ImmutableMemory* property), 31  
 content\_endin (*bytesparse.base.MutableBytesparse* property), 91  
 content\_endin (*bytesparse.base.MutableMemory* property), 166  
 content\_endin (*bytesparse.inplace.bytesparse* property), 319  
 content\_endin (*bytesparse.inplace.Memory* property), 242  
 content\_items() (*bytesparse.base.ImmutableMemory* method), 32  
 content\_items() (*bytesparse.base.MutableBytesparse* method), 91  
 content\_items() (*bytesparse.base.MutableMemory* method), 167  
 content\_items() (*bytesparse.inplace.bytesparse* method), 320  
 content\_items() (*bytesparse.inplace.Memory* method), 242  
 content\_keys() (*bytesparse.base.ImmutableMemory* method), 33  
 content\_keys() (*bytesparse.base.MutableBytesparse*



`method`), 92  
`content_keys()` (`byparse.base.MutableMemory` `method`), 167  
`content_keys()` (`byparse.inplace.byparse` `method`), 320  
`content_keys()` (`byparse.inplace.Memory` `method`), 243  
`content_parts` (`byparse.base.ImmutableMemory` `property`), 33  
`content_parts` (`byparse.base.MutableByparse` `property`), 93  
`content_parts` (`byparse.base.MutableMemory` `property`), 168  
`content_parts` (`byparse.inplace.byparse` `property`), 321  
`content_parts` (`byparse.inplace.Memory` `property`), 244  
`content_size` (`byparse.base.ImmutableMemory` `property`), 34  
`content_size` (`byparse.base.MutableByparse` `property`), 93  
`content_size` (`byparse.base.MutableMemory` `property`), 169  
`content_size` (`byparse.inplace.byparse` `property`), 322  
`content_size` (`byparse.inplace.Memory` `property`), 244  
`content_span` (`byparse.base.ImmutableMemory` `property`), 35  
`content_span` (`byparse.base.MutableByparse` `property`), 94  
`content_span` (`byparse.base.MutableMemory` `property`), 169  
`content_span` (`byparse.inplace.byparse` `property`), 322  
`content_span` (`byparse.inplace.Memory` `property`), 245  
`content_start` (`byparse.base.ImmutableMemory` `property`), 35  
`content_start` (`byparse.base.MutableByparse` `property`), 95  
`content_start` (`byparse.base.MutableMemory` `property`), 170  
`content_start` (`byparse.inplace.byparse` `property`), 323  
`content_start` (`byparse.inplace.Memory` `property`), 246  
`content_values()` (`byparse.base.ImmutableMemory` `method`), 36  
`content_values()` (`byparse.base.MutableByparse` `method`), 95  
`content_values()` (`byparse.base.MutableMemory` `method`), 171  
`content_values()` (`byparse.inplace.byparse` `method`), 324  
`content_values()` (`byparse.inplace.Memory` `method`), 246  
`contiguous` (`byparse.base.ImmutableMemory` `property`), 37  
`contiguous` (`byparse.base.MutableByparse` `property`), 96  
`contiguous` (`byparse.base.MutableMemory` `property`), 171  
`contiguous` (`byparse.inplace.byparse` `property`), 324  
`contiguous` (`byparse.inplace.Memory` `property`), 247  
`copy()` (`byparse.base.ImmutableMemory` `method`), 37  
`copy()` (`byparse.base.MutableByparse` `method`), 97  
`copy()` (`byparse.base.MutableMemory` `method`), 172  
`copy()` (`byparse.inplace.byparse` `method`), 325  
`copy()` (`byparse.inplace.Memory` `method`), 248  
`count()` (`byparse.base.ImmutableMemory` `method`), 38  
`count()` (`byparse.base.MutableByparse` `method`), 98  
`count()` (`byparse.base.MutableMemory` `method`), 173  
`count()` (`byparse.inplace.byparse` `method`), 326  
`count()` (`byparse.inplace.Memory` `method`), 249  
`crop()` (`byparse.base.MutableByparse` `method`), 98  
`crop()` (`byparse.base.MutableMemory` `method`), 173  
`crop()` (`byparse.inplace.byparse` `method`), 326  
`crop()` (`byparse.inplace.Memory` `method`), 249  
`crop_backup()` (`byparse.base.MutableByparse` `method`), 99  
`crop_backup()` (`byparse.base.MutableMemory` `method`), 174  
`crop_backup()` (`byparse.inplace.byparse` `method`), 327  
`crop_backup()` (`byparse.inplace.Memory` `method`), 250  
`crop_restore()` (`byparse.base.MutableByparse` `method`), 99  
`crop_restore()` (`byparse.base.MutableMemory` `method`), 174  
`crop_restore()` (`byparse.inplace.byparse` `method`), 327  
`crop_restore()` (`byparse.inplace.Memory` `method`), 250  
`cut()` (`byparse.base.MutableByparse` `method`), 99  
`cut()` (`byparse.base.MutableMemory` `method`), 174  
`cut()` (`byparse.inplace.byparse` `method`), 327  
`cut()` (`byparse.inplace.Memory` `method`), 250

## D

`delete()` (`byparse.base.MutableByparse` `method`), 100

- delete() (*bytestparse.base.MutableMemory* method), 175  
 delete() (*bytestparse.inplace.bytestparse* method), 328  
 delete() (*bytestparse.inplace.Memory* method), 251  
 delete\_backup() (*bytestparse.base.MutableBytesparse* method), 100  
 delete\_backup() (*bytestparse.base.MutableMemory* method), 175  
 delete\_backup() (*bytestparse.inplace.bytestparse* method), 328  
 delete\_backup() (*bytestparse.inplace.Memory* method), 251  
 delete\_restore() (*bytestparse.base.MutableBytesparse* method), 100  
 delete\_restore() (*bytestparse.base.MutableMemory* method), 176  
 delete\_restore() (*bytestparse.inplace.bytestparse* method), 329  
 delete\_restore() (*bytestparse.inplace.Memory* method), 251  
 detach() (*bytestparse.io.MemoryIO* method), 374
- ## E
- endex (*bytestparse.base.ImmutableMemory* property), 39  
 endex (*bytestparse.base.MutableBytesparse* property), 101  
 endex (*bytestparse.base.MutableMemory* property), 176  
 endex (*bytestparse.inplace.bytestparse* property), 329  
 endex (*bytestparse.inplace.Memory* property), 252  
 endin (*bytestparse.base.ImmutableMemory* property), 40  
 endin (*bytestparse.base.MutableBytesparse* property), 101  
 endin (*bytestparse.base.MutableMemory* property), 177  
 endin (*bytestparse.inplace.bytestparse* property), 330  
 endin (*bytestparse.inplace.Memory* property), 252  
 equal\_span() (*bytestparse.base.ImmutableMemory* method), 40  
 equal\_span() (*bytestparse.base.MutableBytesparse* method), 102  
 equal\_span() (*bytestparse.base.MutableMemory* method), 177  
 equal\_span() (*bytestparse.inplace.bytestparse* method), 330  
 equal\_span() (*bytestparse.inplace.Memory* method), 253  
 extend() (*bytestparse.base.MutableBytesparse* method), 103  
 extend() (*bytestparse.base.MutableMemory* method), 178  
 extend() (*bytestparse.inplace.bytestparse* method), 331  
 extend() (*bytestparse.inplace.Memory* method), 254  
 extend\_backup() (*bytestparse.base.MutableBytesparse* method), 104  
 extend\_backup() (*bytestparse.base.MutableMemory* method), 179  
 extend\_backup() (*bytestparse.inplace.bytestparse* method), 332  
 extend\_backup() (*bytestparse.inplace.Memory* method), 255  
 extend\_restore() (*bytestparse.base.MutableBytesparse* method), 104  
 extend\_restore() (*bytestparse.base.MutableMemory* method), 179  
 extend\_restore() (*bytestparse.inplace.bytestparse* method), 332  
 extend\_restore() (*bytestparse.inplace.Memory* method), 255  
 extract() (*bytestparse.base.ImmutableMemory* method), 41  
 extract() (*bytestparse.base.MutableBytesparse* method), 104  
 extract() (*bytestparse.base.MutableMemory* method), 179  
 extract() (*bytestparse.inplace.bytestparse* method), 332  
 extract() (*bytestparse.inplace.Memory* method), 255
- ## F
- fileno() (*bytestparse.io.MemoryIO* method), 375  
 fill() (*bytestparse.base.MutableBytesparse* method), 105  
 fill() (*bytestparse.base.MutableMemory* method), 180  
 fill() (*bytestparse.inplace.bytestparse* method), 333  
 fill() (*bytestparse.inplace.Memory* method), 256  
 fill\_backup() (*bytestparse.base.MutableBytesparse* method), 106  
 fill\_backup() (*bytestparse.base.MutableMemory* method), 181  
 fill\_backup() (*bytestparse.inplace.bytestparse* method), 334  
 fill\_backup() (*bytestparse.inplace.Memory* method), 257  
 fill\_restore() (*bytestparse.base.MutableBytesparse* method), 106  
 fill\_restore() (*bytestparse.base.MutableMemory* method), 181  
 fill\_restore() (*bytestparse.inplace.bytestparse* method), 334  
 fill\_restore() (*bytestparse.inplace.Memory* method), 257  
 find() (*bytestparse.base.ImmutableMemory* method), 42  
 find() (*bytestparse.base.MutableBytesparse* method), 106  
 find() (*bytestparse.base.MutableMemory* method), 181  
 find() (*bytestparse.inplace.bytestparse* method), 334  
 find() (*bytestparse.inplace.Memory* method), 257

- `flood()` (*byparse.base.MutableByparse method*), 106
  - `flood()` (*byparse.base.MutableMemory method*), 181
  - `flood()` (*byparse.inplace.byparse method*), 334
  - `flood()` (*byparse.inplace.Memory method*), 257
  - `flood_backup()` (*byparse.base.MutableByparse method*), 107
  - `flood_backup()` (*byparse.base.MutableMemory method*), 182
  - `flood_backup()` (*byparse.inplace.byparse method*), 335
  - `flood_backup()` (*byparse.inplace.Memory method*), 258
  - `flood_restore()` (*byparse.base.MutableByparse method*), 107
  - `flood_restore()` (*byparse.base.MutableMemory method*), 182
  - `flood_restore()` (*byparse.inplace.byparse method*), 335
  - `flood_restore()` (*byparse.inplace.Memory method*), 258
  - `flush()` (*byparse.io.MemoryIO method*), 375
  - `from_blocks()` (*byparse.base.ImmutableMemory class method*), 42
  - `from_blocks()` (*byparse.base.MutableByparse class method*), 108
  - `from_blocks()` (*byparse.base.MutableMemory class method*), 183
  - `from_blocks()` (*byparse.inplace.byparse class method*), 336
  - `from_blocks()` (*byparse.inplace.Memory class method*), 259
  - `from_bytes()` (*byparse.base.ImmutableMemory class method*), 43
  - `from_bytes()` (*byparse.base.MutableByparse class method*), 109
  - `from_bytes()` (*byparse.base.MutableMemory class method*), 184
  - `from_bytes()` (*byparse.inplace.byparse class method*), 337
  - `from_bytes()` (*byparse.inplace.Memory class method*), 260
  - `from_items()` (*byparse.base.ImmutableMemory class method*), 44
  - `from_items()` (*byparse.base.MutableByparse class method*), 109
  - `from_items()` (*byparse.base.MutableMemory class method*), 184
  - `from_items()` (*byparse.inplace.byparse class method*), 337
  - `from_items()` (*byparse.inplace.Memory class method*), 260
  - `from_memory()` (*byparse.base.ImmutableMemory class method*), 45
  - `from_memory()` (*byparse.base.MutableByparse class method*), 110
  - `from_memory()` (*byparse.base.MutableMemory class method*), 185
  - `from_memory()` (*byparse.inplace.byparse class method*), 338
  - `from_memory()` (*byparse.inplace.Memory class method*), 261
  - `from_values()` (*byparse.base.ImmutableMemory class method*), 46
  - `from_values()` (*byparse.base.MutableByparse class method*), 111
  - `from_values()` (*byparse.base.MutableMemory class method*), 186
  - `from_values()` (*byparse.inplace.byparse class method*), 339
  - `from_values()` (*byparse.inplace.Memory class method*), 262
  - `fromhex()` (*byparse.base.ImmutableMemory class method*), 47
  - `fromhex()` (*byparse.base.MutableByparse class method*), 112
  - `fromhex()` (*byparse.base.MutableMemory class method*), 187
  - `fromhex()` (*byparse.inplace.byparse class method*), 340
  - `fromhex()` (*byparse.inplace.Memory class method*), 263
- ## G
- `gaps()` (*byparse.base.ImmutableMemory method*), 47
  - `gaps()` (*byparse.base.MutableByparse method*), 113
  - `gaps()` (*byparse.base.MutableMemory method*), 188
  - `gaps()` (*byparse.inplace.byparse method*), 341
  - `gaps()` (*byparse.inplace.Memory method*), 264
  - `get()` (*byparse.base.ImmutableMemory method*), 48
  - `get()` (*byparse.base.MutableByparse method*), 113
  - `get()` (*byparse.base.MutableMemory method*), 188
  - `get()` (*byparse.inplace.byparse method*), 341
  - `get()` (*byparse.inplace.Memory method*), 264
  - `getbuffer()` (*byparse.io.MemoryIO method*), 375
  - `getvalue()` (*byparse.io.MemoryIO method*), 376
- ## H
- `hex()` (*byparse.base.ImmutableMemory method*), 49
  - `hex()` (*byparse.base.MutableByparse method*), 114
  - `hex()` (*byparse.base.MutableMemory method*), 189
  - `hex()` (*byparse.inplace.byparse method*), 342
  - `hex()` (*byparse.inplace.Memory method*), 265
  - `hexdump()` (*byparse.base.ImmutableMemory method*), 49
  - `hexdump()` (*byparse.base.MutableByparse method*), 115



hexdump() (*bytestparse.base.MutableMemory method*), 190  
 hexdump() (*bytestparse.inplace.bytestparse method*), 343  
 hexdump() (*bytestparse.inplace.Memory method*), 266  
 HUMAN\_ASCII (in module *bytestparse.base*), 13  
**I**  
 ImmutableMemory (class in *bytestparse.base*), 13  
 index() (*bytestparse.base.ImmutableMemory method*), 51  
 index() (*bytestparse.base.MutableBytesparse method*), 116  
 index() (*bytestparse.base.MutableMemory method*), 191  
 index() (*bytestparse.inplace.bytestparse method*), 344  
 index() (*bytestparse.inplace.Memory method*), 267  
 insert() (*bytestparse.base.MutableBytesparse method*), 117  
 insert() (*bytestparse.base.MutableMemory method*), 192  
 insert() (*bytestparse.inplace.bytestparse method*), 345  
 insert() (*bytestparse.inplace.Memory method*), 268  
 insert\_backup() (*bytestparse.base.MutableBytesparse method*), 117  
 insert\_backup() (*bytestparse.base.MutableMemory method*), 192  
 insert\_backup() (*bytestparse.inplace.bytestparse method*), 345  
 insert\_backup() (*bytestparse.inplace.Memory method*), 268  
 insert\_restore() (*bytestparse.base.MutableBytesparse method*), 117  
 insert\_restore() (*bytestparse.base.MutableMemory method*), 192  
 insert\_restore() (*bytestparse.inplace.bytestparse method*), 345  
 insert\_restore() (*bytestparse.inplace.Memory method*), 268  
 intervals() (*bytestparse.base.ImmutableMemory method*), 51  
 intervals() (*bytestparse.base.MutableBytesparse method*), 118  
 intervals() (*bytestparse.base.MutableMemory method*), 193  
 intervals() (*bytestparse.inplace.bytestparse method*), 346  
 intervals() (*bytestparse.inplace.Memory method*), 269  
 isatty() (*bytestparse.io.MemoryIO method*), 376  
 items() (*bytestparse.base.ImmutableMemory method*), 52  
 items() (*bytestparse.base.MutableBytesparse method*), 118  
 items() (*bytestparse.base.MutableMemory method*), 193  
 items() (*bytestparse.inplace.bytestparse method*), 346

items() (*bytestparse.inplace.Memory method*), 269

## K

keys() (*bytestparse.base.ImmutableMemory method*), 52  
 keys() (*bytestparse.base.MutableBytesparse method*), 119  
 keys() (*bytestparse.base.MutableMemory method*), 194  
 keys() (*bytestparse.inplace.bytestparse method*), 347  
 keys() (*bytestparse.inplace.Memory method*), 270

## M

memory (*bytestparse.io.MemoryIO property*), 376  
 Memory (class in *bytestparse.inplace*), 215  
 MemoryIO (class in *bytestparse.io*), 369  
 module  
     *bytestparse*, 11  
     *bytestparse.base*, 12  
     *bytestparse.inplace*, 214  
     *bytestparse.io*, 367  
 MutableBytesparse (class in *bytestparse.base*), 62  
 MutableMemory (class in *bytestparse.base*), 139

## P

peek() (*bytestparse.base.ImmutableMemory method*), 53  
 peek() (*bytestparse.base.MutableBytesparse method*), 120  
 peek() (*bytestparse.base.MutableMemory method*), 195  
 peek() (*bytestparse.inplace.bytestparse method*), 348  
 peek() (*bytestparse.inplace.Memory method*), 271  
 peek() (*bytestparse.io.MemoryIO method*), 377  
 poke() (*bytestparse.base.MutableBytesparse method*), 121  
 poke() (*bytestparse.base.MutableMemory method*), 196  
 poke() (*bytestparse.inplace.bytestparse method*), 348  
 poke() (*bytestparse.inplace.Memory method*), 271  
 poke\_backup() (*bytestparse.base.MutableBytesparse method*), 121  
 poke\_backup() (*bytestparse.base.MutableMemory method*), 196  
 poke\_backup() (*bytestparse.inplace.bytestparse method*), 349  
 poke\_backup() (*bytestparse.inplace.Memory method*), 272  
 poke\_restore() (*bytestparse.base.MutableBytesparse method*), 121  
 poke\_restore() (*bytestparse.base.MutableMemory method*), 196  
 poke\_restore() (*bytestparse.inplace.bytestparse method*), 349  
 poke\_restore() (*bytestparse.inplace.Memory method*), 272  
 pop() (*bytestparse.base.MutableBytesparse method*), 122  
 pop() (*bytestparse.base.MutableMemory method*), 197

pop() (*bytesparse.inplace.bytesparse method*), 349  
 pop() (*bytesparse.inplace.Memory method*), 272  
 pop\_backup() (*bytesparse.base.MutableBytesparse method*), 122  
 pop\_backup() (*bytesparse.base.MutableMemory method*), 197  
 pop\_backup() (*bytesparse.inplace.bytesparse method*), 350  
 pop\_backup() (*bytesparse.inplace.Memory method*), 273  
 pop\_restore() (*bytesparse.base.MutableBytesparse method*), 122  
 pop\_restore() (*bytesparse.base.MutableMemory method*), 197  
 pop\_restore() (*bytesparse.inplace.bytesparse method*), 350  
 pop\_restore() (*bytesparse.inplace.Memory method*), 273  
 popitem() (*bytesparse.base.MutableBytesparse method*), 123  
 popitem() (*bytesparse.base.MutableMemory method*), 198  
 popitem() (*bytesparse.inplace.bytesparse method*), 350  
 popitem() (*bytesparse.inplace.Memory method*), 273  
 popitem\_backup() (*bytesparse.base.MutableBytesparse method*), 123  
 popitem\_backup() (*bytesparse.base.MutableMemory method*), 198  
 popitem\_backup() (*bytesparse.inplace.bytesparse method*), 351  
 popitem\_backup() (*bytesparse.inplace.Memory method*), 274  
 popitem\_restore() (*bytesparse.base.MutableBytesparse method*), 123  
 popitem\_restore() (*bytesparse.base.MutableMemory method*), 198  
 popitem\_restore() (*bytesparse.inplace.bytesparse method*), 351  
 popitem\_restore() (*bytesparse.inplace.Memory method*), 274

## R

read() (*bytesparse.base.ImmutableMemory method*), 54  
 read() (*bytesparse.base.MutableBytesparse method*), 124  
 read() (*bytesparse.base.MutableMemory method*), 199  
 read() (*bytesparse.inplace.bytesparse method*), 351  
 read() (*bytesparse.inplace.Memory method*), 274  
 read() (*bytesparse.io.MemoryIO method*), 378  
 read1() (*bytesparse.io.MemoryIO method*), 379  
 readable() (*bytesparse.io.MemoryIO method*), 380  
 readinto() (*bytesparse.base.ImmutableMemory method*), 55  
 readinto() (*bytesparse.base.MutableBytesparse method*), 124  
 readinto() (*bytesparse.base.MutableMemory method*), 199  
 readinto() (*bytesparse.inplace.bytesparse method*), 352  
 readinto() (*bytesparse.inplace.Memory method*), 275  
 readinto() (*bytesparse.io.MemoryIO method*), 380  
 readinto1() (*bytesparse.io.MemoryIO method*), 381  
 readline() (*bytesparse.io.MemoryIO method*), 383  
 readlines() (*bytesparse.io.MemoryIO method*), 384  
 remove() (*bytesparse.base.MutableBytesparse method*), 125  
 remove() (*bytesparse.base.MutableMemory method*), 200  
 remove() (*bytesparse.inplace.bytesparse method*), 353  
 remove() (*bytesparse.inplace.Memory method*), 276  
 remove\_backup() (*bytesparse.base.MutableBytesparse method*), 126  
 remove\_backup() (*bytesparse.base.MutableMemory method*), 201  
 remove\_backup() (*bytesparse.inplace.bytesparse method*), 354  
 remove\_backup() (*bytesparse.inplace.Memory method*), 277  
 remove\_restore() (*bytesparse.base.MutableBytesparse method*), 126  
 remove\_restore() (*bytesparse.base.MutableMemory method*), 201  
 remove\_restore() (*bytesparse.inplace.bytesparse method*), 354  
 remove\_restore() (*bytesparse.inplace.Memory method*), 277  
 reserve() (*bytesparse.base.MutableBytesparse method*), 126  
 reserve() (*bytesparse.base.MutableMemory method*), 201  
 reserve() (*bytesparse.inplace.bytesparse method*), 354  
 reserve() (*bytesparse.inplace.Memory method*), 277  
 reserve\_backup() (*bytesparse.base.MutableBytesparse method*), 127  
 reserve\_backup() (*bytesparse.base.MutableMemory method*), 202  
 reserve\_backup() (*bytesparse.inplace.bytesparse method*), 355  
 reserve\_backup() (*bytesparse.inplace.Memory method*), 278  
 reserve\_restore() (*bytesparse.base.MutableBytesparse method*), 127

reserve\_restore() (*bytestparse.base.MutableMemory method*), 202  
 reserve\_restore() (*bytestparse.inplace.bytestparse method*), 355  
 reserve\_restore() (*bytestparse.inplace.Memory method*), 278  
 reverse() (*bytestparse.base.MutableBytesparse method*), 128  
 reverse() (*bytestparse.base.MutableMemory method*), 203  
 reverse() (*bytestparse.inplace.bytestparse method*), 356  
 reverse() (*bytestparse.inplace.Memory method*), 279  
 rfind() (*bytestparse.base.ImmutableMemory method*), 56  
 rfind() (*bytestparse.base.MutableBytesparse method*), 128  
 rfind() (*bytestparse.base.MutableMemory method*), 203  
 rfind() (*bytestparse.inplace.bytestparse method*), 356  
 rfind() (*bytestparse.inplace.Memory method*), 279  
 rindex() (*bytestparse.base.ImmutableMemory method*), 56  
 rindex() (*bytestparse.base.MutableBytesparse method*), 129  
 rindex() (*bytestparse.base.MutableMemory method*), 204  
 rindex() (*bytestparse.inplace.bytestparse method*), 357  
 rindex() (*bytestparse.inplace.Memory method*), 280  
 rvalues() (*bytestparse.base.ImmutableMemory method*), 56  
 rvalues() (*bytestparse.base.MutableBytesparse method*), 129  
 rvalues() (*bytestparse.base.MutableMemory method*), 204  
 rvalues() (*bytestparse.inplace.bytestparse method*), 357  
 rvalues() (*bytestparse.inplace.Memory method*), 280

## S

seek() (*bytestparse.io.MemoryIO method*), 385  
 SEEK\_CUR (*in module bytestparse.io*), 368  
 SEEK\_DATA (*in module bytestparse.io*), 368  
 SEEK\_END (*in module bytestparse.io*), 368  
 SEEK\_HOLE (*in module bytestparse.io*), 368  
 SEEK\_SET (*in module bytestparse.io*), 368  
 seekable() (*bytestparse.io.MemoryIO method*), 387  
 setdefault() (*bytestparse.base.MutableBytesparse method*), 130  
 setdefault() (*bytestparse.base.MutableMemory method*), 205  
 setdefault() (*bytestparse.inplace.bytestparse method*), 358  
 setdefault() (*bytestparse.inplace.Memory method*), 281  
 setdefault\_backup() (*bytestparse.base.MutableBytesparse method*), 131  
 setdefault\_backup() (*bytestparse.inplace.bytestparse method*), 359  
 setdefault\_backup() (*bytestparse.inplace.Memory method*), 282  
 setdefault\_restore() (*bytestparse.base.MutableBytesparse method*), 131  
 setdefault\_restore() (*bytestparse.inplace.bytestparse method*), 359  
 setdefault\_restore() (*bytestparse.inplace.Memory method*), 282  
 shift() (*bytestparse.base.MutableBytesparse method*), 131  
 shift() (*bytestparse.base.MutableMemory method*), 206  
 shift() (*bytestparse.inplace.bytestparse method*), 359  
 shift() (*bytestparse.inplace.Memory method*), 282  
 shift\_backup() (*bytestparse.base.MutableBytesparse method*), 132  
 shift\_backup() (*bytestparse.base.MutableMemory method*), 207  
 shift\_backup() (*bytestparse.inplace.bytestparse method*), 360  
 shift\_backup() (*bytestparse.inplace.Memory method*), 283  
 shift\_restore() (*bytestparse.base.MutableBytesparse method*), 132  
 shift\_restore() (*bytestparse.base.MutableMemory method*), 207  
 shift\_restore() (*bytestparse.inplace.bytestparse method*), 360  
 shift\_restore() (*bytestparse.inplace.Memory method*), 283  
 skip\_data() (*bytestparse.io.MemoryIO method*), 387  
 skip\_hole() (*bytestparse.io.MemoryIO method*), 387  
 span (*bytestparse.base.ImmutableMemory property*), 57  
 span (*bytestparse.base.MutableBytesparse property*), 132  
 span (*bytestparse.base.MutableMemory property*), 207  
 span (*bytestparse.inplace.bytestparse property*), 360  
 span (*bytestparse.inplace.Memory property*), 283  
 start (*bytestparse.base.ImmutableMemory property*), 58  
 start (*bytestparse.base.MutableBytesparse property*), 133  
 start (*bytestparse.base.MutableMemory property*), 208  
 start (*bytestparse.inplace.bytestparse property*), 361  
 start (*bytestparse.inplace.Memory property*), 284  
 STR\_MAX\_CONTENT\_SIZE (*in module bytestparse.base*), 13

## T

[tell\(\)](#) (*bytesparse.io.MemoryIO method*), 388  
[to\\_blocks\(\)](#) (*bytesparse.base.ImmutableMemory method*), 59  
[to\\_blocks\(\)](#) (*bytesparse.base.MutableBytesparse method*), 134  
[to\\_blocks\(\)](#) (*bytesparse.base.MutableMemory method*), 209  
[to\\_blocks\(\)](#) (*bytesparse.inplace.bytesparse method*), 362  
[to\\_blocks\(\)](#) (*bytesparse.inplace.Memory method*), 285  
[to\\_bytes\(\)](#) (*bytesparse.base.ImmutableMemory method*), 59  
[to\\_bytes\(\)](#) (*bytesparse.base.MutableBytesparse method*), 134  
[to\\_bytes\(\)](#) (*bytesparse.base.MutableMemory method*), 209  
[to\\_bytes\(\)](#) (*bytesparse.inplace.bytesparse method*), 362  
[to\\_bytes\(\)](#) (*bytesparse.inplace.Memory method*), 285  
[truncate\(\)](#) (*bytesparse.io.MemoryIO method*), 389

## U

[update\(\)](#) (*bytesparse.base.MutableBytesparse method*), 135  
[update\(\)](#) (*bytesparse.base.MutableMemory method*), 210  
[update\(\)](#) (*bytesparse.inplace.bytesparse method*), 363  
[update\(\)](#) (*bytesparse.inplace.Memory method*), 286  
[update\\_backup\(\)](#) (*bytesparse.base.MutableBytesparse method*), 136  
[update\\_backup\(\)](#) (*bytesparse.base.MutableMemory method*), 211  
[update\\_backup\(\)](#) (*bytesparse.inplace.bytesparse method*), 364  
[update\\_backup\(\)](#) (*bytesparse.inplace.Memory method*), 287  
[update\\_restore\(\)](#) (*bytesparse.base.MutableBytesparse method*), 136  
[update\\_restore\(\)](#) (*bytesparse.base.MutableMemory method*), 211  
[update\\_restore\(\)](#) (*bytesparse.inplace.bytesparse method*), 364  
[update\\_restore\(\)](#) (*bytesparse.inplace.Memory method*), 287

## V

[validate\(\)](#) (*bytesparse.base.ImmutableMemory method*), 60  
[validate\(\)](#) (*bytesparse.base.MutableBytesparse method*), 136  
[validate\(\)](#) (*bytesparse.base.MutableMemory method*), 211

[validate\(\)](#) (*bytesparse.inplace.bytesparse method*), 364  
[validate\(\)](#) (*bytesparse.inplace.Memory method*), 287  
[values\(\)](#) (*bytesparse.base.ImmutableMemory method*), 60  
[values\(\)](#) (*bytesparse.base.MutableBytesparse method*), 136  
[values\(\)](#) (*bytesparse.base.MutableMemory method*), 211  
[values\(\)](#) (*bytesparse.inplace.bytesparse method*), 364  
[values\(\)](#) (*bytesparse.inplace.Memory method*), 287  
[view\(\)](#) (*bytesparse.base.ImmutableMemory method*), 61  
[view\(\)](#) (*bytesparse.base.MutableBytesparse method*), 137  
[view\(\)](#) (*bytesparse.base.MutableMemory method*), 212  
[view\(\)](#) (*bytesparse.inplace.bytesparse method*), 365  
[view\(\)](#) (*bytesparse.inplace.Memory method*), 288

## W

[writable\(\)](#) (*bytesparse.io.MemoryIO method*), 390  
[write\(\)](#) (*bytesparse.base.MutableBytesparse method*), 138  
[write\(\)](#) (*bytesparse.base.MutableMemory method*), 213  
[write\(\)](#) (*bytesparse.inplace.bytesparse method*), 366  
[write\(\)](#) (*bytesparse.inplace.Memory method*), 289  
[write\(\)](#) (*bytesparse.io.MemoryIO method*), 390  
[write\\_backup\(\)](#) (*bytesparse.base.MutableBytesparse method*), 139  
[write\\_backup\(\)](#) (*bytesparse.base.MutableMemory method*), 214  
[write\\_backup\(\)](#) (*bytesparse.inplace.bytesparse method*), 367  
[write\\_backup\(\)](#) (*bytesparse.inplace.Memory method*), 290  
[write\\_restore\(\)](#) (*bytesparse.base.MutableBytesparse method*), 139  
[write\\_restore\(\)](#) (*bytesparse.base.MutableMemory method*), 214  
[write\\_restore\(\)](#) (*bytesparse.inplace.bytesparse method*), 367  
[write\\_restore\(\)](#) (*bytesparse.inplace.Memory method*), 290  
[writelines\(\)](#) (*bytesparse.io.MemoryIO method*), 392