cbytesparse

Release 0.0.6

Andrea Zoppi

Feb 18, 2023

CONTENTS

CHAPTER

ONE

OVERVIEW

docs
tests
package

Library to handle sparse bytes within a virtual memory space.

- Free software: BSD 2-Clause License

1.1 Objectives

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

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

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

1.2 Python implementation

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

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

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

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

1.3 Cython implementation

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

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

The addressing space is limited to that of an `uint_fast64_t`, so it is not possible to have an infinite addressing space, nor negative addresses. To keep the implementation code simple enough, the highest address (i.e. `0xFFFFFFFFFFFFFFFF`) is reserved.

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

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

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

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

1.4 Examples

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

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

(continues on next page)

(continued from previous page)

```
>>> m.extend(b'!!')  # more emphasis!!!
>>> bytes(m)
b'Hello, World!!!'
>>> # ------------------------------------------------------------------
>>> i = m.index(b',')  # gets the address of the comma
>>> m[:i] = b'Ciao'  # replaces 'Hello' with 'Ciao'
>>> bytes(m)
b'Ciao, World!!!'
>>> # ------------------------------------------------------------------
>>> i = m.index(b',')  # gets the address of the comma
>>> m.insert(i, b'ne')  # inserts 'ne' to make 'Ciaone' ("big ciao")
>>> bytes(m)
b'Ciaone, World!!!'
>>> # ------------------------------------------------------------------
>>> i = m.index(b',')  # gets the address of the comma
>>> m[(i - 2):i] = b' ciao'  # makes 'Ciaone' --> 'Ciao ciao'
>>> bytes(m)
b'Ciao ciao, World!!!'
>>> # ------------------------------------------------------------------
>>> m.pop()  # less emphasis --> 33 == ord('!')
33
>>> bytes(m)
b'Ciao ciao, World!!'
>>> # ------------------------------------------------------------------
>>> del m[m.index(b'l')]  # makes 'World' --> 'Word'
>>> bytes(m)
b'Ciao ciao, Word!!'
>>> # ------------------------------------------------------------------
>>> m.popitem()  # less emphasis --> pops 33 (== '!') at address 16
(16, 33)
>>> bytes(m)
b'Ciao ciao, Word!'
>>> # ------------------------------------------------------------------
>>> m.remove(b' ciao')  # self-explanatory
>>> bytes(m)
b'Ciao, Word!'
>>> # ------------------------------------------------------------------
>>> i = m.index(b',')  # gets the address of the comma
>>> m.clear(start=i, endex=(i + 2))  # makes empty space between the words
>>> m.to_blocks()  # exports as data block list
[[0, b'Ciao'], [6, b'Word!']]
>>> m.contiguous  # multiple data blocks (emptiness inbetween)
False
>>> m.content_parts  # two data blocks
2
>>> m.content_size  # excluding emptiness
9
>>> len(m)  # including emptiness
11
>>> # ------------------------------------------------------------------
>>> m.flood(pattern=b'.')  # replaces emptiness with dots
>>> bytes(m)
```

(continues on next page)

(continued from previous page)

```
b'Ciao..Word!'
>>> m[-2]  # 100 == ord('d')
100
>>> # ------------------------------------------------------------------
>>> m.peek(-2)  # 100 == ord('d')
100
>>> m.poke(-2, b'k')  # makes 'Word' --> 'Work'
>>> bytes(m)
b'Ciao..Work!'
>>> # ------------------------------------------------------------------
>>> m.crop(start=m.index(b'W'))  # keeps 'Work!'
>>> m.to_blocks()
[[6, b'Work!']]
>>> m.span  # address range of the whole memory
(6, 11)
>>> m.start, m.endex  # same as above
(6, 11)
>>> # ------------------------------------------------------------------
>>> m.bound_span = (2, 10)  # sets memory address bounds
>>> str(m)
"<2, [[6, b'Work']], 10>"
>>> m.to_blocks()
[[6, b'Work']]
>>> # ------------------------------------------------------------------
>>> m.shift(-6)  # shifts to the left; NOTE: address bounds will cut 2 bytes!
>>> m.to_blocks()
[[2, b'rk']]
>>> str(m)
"<2, [[2, b'rk']], 10>"
>>> # ------------------------------------------------------------------
>>> a = bytesparse(b'Ma')
>>> a.write(0, m)  # writes [2, b'rk'] --> 'Mark'
>>> a.to_blocks()
[[0, b'Mark']]
>>> # ------------------------------------------------------------------
>>> b = Memory.from_bytes(b'ing', offset=4)
>>> b.to_blocks()
[[4, b'ing']]
>>> # ------------------------------------------------------------------
>>> a.write(0, b)  # writes [4, b'ing'] --> 'Marking'
>>> a.to_blocks()
[[0, b'Marking']]
>>> # ------------------------------------------------------------------
>>> a.reserve(4, 2)  # inserts 2 empty bytes after 'Mark'
>>> a.to_blocks()
[[0, b'Mark'], [6, b'ing']]
>>> # ------------------------------------------------------------------
>>> a.write(4, b'et')  # --> 'Marketing'
>>> a.to_blocks()
[[0, b'Marketing']]
>>> # ------------------------------------------------------------------
>>> a.fill(1, -1, b'*')  # fills asterisks between the first and last letters
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

(continues on next page)

(continued from previous page)

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

1.5 Documentation

For the full documentation, please refer to:

https://cbytesparse.readthedocs.io/

1.6 Installation

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

```
$ pip install cbytesparse
```

From the source code root directory:

```
$ pip install .
```

1.7 Development

To run the all the tests:

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

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

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

or alternatively:

```
$ tox -e cythonize
```

CHAPTER

TWO

INSTALLATION

2.1 From *PyPI*

At the command line:

```
$ pip install cbytesparse
```

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

2.2 From source

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

```
$ pip install .
```

CHAPTER

THREE

REFERENCE

cbytesparse	Utilities for sparse blocks of bytes.
cbytesparse.c	Cython implementation.
cbytesparse.py	Python wrappers.

3.1 cbytesparse

Utilities for sparse blocks of bytes.

Blocks are a useful way to describe sparse linear data.

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

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

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

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

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

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

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

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

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

Contiguous blocks are *non-overlapping*.

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

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

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

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

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

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

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

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

cbytesparse.base	Common stuff, shared across modules.
cbytesparse.c	Cython implementation.
cbytesparse.py	Python wrappers.

3.1.1 cbytesparse.base

Common stuff, shared across modules.

Classes

BaseBytesMethods	Provides useful methods to a byte buffer.
BaseInplaceView	Provides inplace methods to a byte buffer.

cbytesparse.base.BaseBytesMethods

class cbytesparse.base.**BaseBytesMethods**(*wrapped*)

Provides useful methods to a byte buffer.

Python's `memoryview` and most *byte-like* objects do not provide many useful methods found instead within the `bytes` or `str` APIs.

This wrapper class adds a low-level implementation of those methods to anything supporting the *buffer protocol*.

Parameters

wrapped (*byte-like*) – The target object supporting the *buffer protocol*.

Examples

```
>>> from cbytesparse import BytesMethods
>>> import numpy
>>> numbers = list(b'ABC')
>>> numbers
[65, 66, 67]
>>> data = numpy.array(numbers, dtype=numpy.ubyte)
>>> data
array([65, 66, 67], dtype=uint8)
>>> data.lower()  # noqa
Traceback (most recent call last):
    ...
AttributeError: 'numpy.ndarray' object has no attribute 'lower'
>>> wrapped = BytesMethods(data)  # noqa
>>> bytes(wrapped.lower())
b'abc'
>>> wrapped = BytesMethods(memoryview(data))
>>> bytes(wrapped.lower())
b'abc'
```

Methods

__init__	
capitalize	byte-like: First character capitalized, the rest lowercase.
center	Return a centered string of length width.
contains	Contains a substring.
count	Counts token occurrences.

continues on next page

Table 1 – continued from previous page

decode	Decode the bytes using the codec registered for encoding.
endswith	Return True if B ends with the specified suffix, False otherwise.
find	Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end].
index	Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end].
isalnum	Return True if all characters in B are alphanumeric and there is at least one character in B, False otherwise.
isalpha	Return True if all characters in B are alphabetic and there is at least one character in B, False otherwise.
isascii	Return True if B is empty or all characters in B are ASCII, False otherwise.
isdecimal	Return True if the string is a decimal string, False otherwise.
isdigit	Return True if all characters in B are digits and there is at least one character in B, False otherwise.
isidentifier	Return True if the string is a valid Python identifier, False otherwise.
islower	Return True if all cased characters in B are lowercase and there is at least one cased character in B, False otherwise.
isnumeric	Return True if the string is a numeric string, False otherwise.
isprintable	Return True if the string is printable, False otherwise.
isspace	Return True if all characters in B are whitespace and there is at least one character in B, False otherwise.
istitle	Return True if B is a titlecased string and there is at least one character in B, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones.
isupper	Return True if all cased characters in B are uppercase and there is at least one cased character in B, False otherwise.
ljust	Return a left-justified string of length width.
lower	Return a copy of B with all ASCII characters converted to lowercase.
lstrip	Strip leading bytes contained in the argument.
maketrans	Return a translation table useable for the bytes or bytearray translate method.
partition	Partition the bytes into three parts using the given separator.
release	Release the underlying buffer exposed by the memoryview object.
removeprefix	Return a bytes object with the given prefix string removed if present.
removesuffix	Return a bytes object with the given suffix string removed if present.
replace	Return a copy with all occurrences of substring old replaced by new.

continues on next page

Table 1 – continued from previous page

rfind	Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end].
rindex	Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end].
rjust	Return a right-justified string of length width.
rpartition	Partition the bytes into three parts using the given separator.
rstrip	Strip trailing bytes contained in the argument.
startswith	Return True if B starts with the specified prefix, False otherwise.
strip	Strip leading and trailing bytes contained in the argument.
swapcase	Return a copy of B with uppercase ASCII characters converted to lowercase ASCII and vice versa.
title	Return a titlecased version of B, i.e. ASCII words start with uppercase characters, all remaining cased characters have lowercase.
tobytes	Return the data in the buffer as a byte string.
tolist	Return the data in the buffer as a list of elements.
translate	Return a copy with each character mapped by the given translation table.
upper	Return a copy of B with all ASCII characters converted to uppercase.
zfill	Pad a numeric string with zeros on the left, to fill a field of the given width.

Attributes

c_contiguous	Contiguous array, *C* language style.
contiguous	Contiguous array.
f_contiguous	Contiguous array, *Fortran* language style.
format	A string containing the format (in struct module style)
itemsize	The size in bytes of each element of the memoryview.
nbytes	The amount of space in bytes that the array would use in
ndim	An integer indicating how many dimensions of a multi-dimensional
obj	The underlying object of the memoryview.
readonly	A bool indicating whether the memory is read only.
shape	A tuple of ndim integers giving the shape of the memory
strides	A tuple of ndim integers giving the size in bytes to access
suboffsets	A tuple of integers used internally for PIL-style arrays.

abstract __bool__()

Has any items.

Returns

bool – Has any items.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'')
>>> bool(instance)
False
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> bool(instance)
True
```

abstract __bytes__()

Creates a bytes clone.

Returns
bytes – Cloned data.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(bytearray(b''))
>>> bytes(instance)
b''
```

```
>>> instance = BytesMethods(bytearray(b'Hello, World!'))
>>> bytes(instance)
b'Hello, World!'
```

classmethod __class_getitem__(*params*)

Parameterizes a generic class.

At least, parameterizing a generic class is the *main* thing this method does. For example, for some generic class *Foo*, this is called when we do *Foo[int]* - there, with *cls=Foo* and *params=int*.

However, note that this method is also called when defining generic classes in the first place with *class Foo(Generic[T]): ...*.

abstract __contains__(*token*)

Checks if some items are contained.

Parameters
token (*byte-like*) – Token to find.

Returns
bool – Token is contained.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> b'World' in instance
True
>>> b'$' in instance
False
>>> ord('o') in instance
True
>>> ord('$') in instance
False
```

abstract __eq__(*other*)

Equality comparison.

Parameters
other (*byte-like*) – Data to compare with *self*.

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

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> data = bytearray(b'Hello, World!')
>>> instance = BytesMethods(data)
>>> instance == data
True
>>> instance == memoryview(data)
True
>>> instance == b'Something else'
False
```

abstract __ge__(*other*)

Return self>=value.

Warning: This method documentation is just a stub, copied directly from `bytes.__ge__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

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.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance[7]  # -> ord('W') = 87
121
>>> bytes(instance[:3])
b'Hel'
>>> bytes(instance[3:10])
b'lo, Wor'
>>> bytes(instance[-1:])
b'!'
>>> bytes(instance[2:10:3])
b'l,o'
>>> bytes(instance[3:10:2])
b'l,Wr'
```

abstract __gt__(*other*)

Return self>value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__gt__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

__hash__ = None

abstract __init__(*wrapped*)

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

abstract __iter__()

Iterates over values.

Yields

int – Value as byte integer.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> list(instance)
[72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]
```

abstract __le__(*other*)

Return self<=value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__le__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract __len__()

Actual length.

Computes the actual length of the wrapped data object.

Returns

int – Data length.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'')
>>> len(instance)
0
```

```
>>> instance = BytesMethods(bytes(7))
>>> len(instance)
7
```

```
>>> instance = BytesMethods(memoryview(b'Hello, World!'))
>>> len(instance)
13
```

abstract __lt__(*other*)

Return self<value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__lt__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract __ne__(*other*)

Inquality comparison.

Parameters

other (*byte-like*) – Data to compare with *self*.

Returns

bool – *self* is not equal to *other*.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> data = bytearray(b'Hello, World!')
>>> instance = BytesMethods(data)
>>> instance != data
False
>>> instance != memoryview(data)
False
>>> instance != b'Something else'
True
```

abstract __reversed__()

Iterates over values, reversed order.

Yields
int – Value as byte integer.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> list(instance)
[72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]
>>> list(reversed(instance))
[33, 100, 108, 114, 111, 87, 32, 44, 111, 108, 108, 101, 72]
```

abstract __sizeof__()

int: Allocated byte size.

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 property c_contiguous: bool

Contiguous array, *C* language style.

Type
bool

abstract capitalize()

byte-like: First character capitalized, the rest lowercase.

abstract center(*width*, *fillchar=b' '*, *factory=<class 'bytes'>*)

Return a centered string of length width.

Padding is done using the specified fill character.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.center()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract contains(*token*, *start=None*, *endex=None*)

Contains a substring.

Parameters

- **token** (*byte-like*) – Token to search.
- **start** (*int*) – Inclusive start of the searched range, `None` to ignore.
- **endex** (*int*) – Exclusive end of the searched range, `None` to ignore.

Returns

bool – Token contained within the wrapped object.

See also:

__contains__()

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance.contains(b'World')
True
>>> instance.contains(b'$')
False
>>> instance.contains(ord('o'))
True
>>> instance.contains(ord('$'))
False
>>> instance.contains(b'Hello', endex=10)
True
>>> instance.contains(b'Hello', endex=3)
False
>>> instance.contains(b'World', start=3)
True
>>> instance.contains(b'World', start=10)
False
>>> instance.contains(b',', start=3, endex=10)
True
>>> instance.contains(b',', start=8, endex=10)
False
```

abstract property contiguous: bool

Contiguous array.

Type

bool

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

Counts token occurrences.

Parameters

- **token** (*byte-like*) – Token to count.
- **start** (*int*) – Inclusive start of the searched range, None to ignore.
- **endex** (*int*) – Exclusive end of the searched range, None to ignore.

Returns

int – The number of items equal to *token*.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance.count(b'o')
2
>>> instance.count(b'l')
3
>>> instance.count(b'll')
3
>>> instance.count(b'World')
1
>>> instance.count(b'$')
0
>>> instance.count(ord('o'))
1
>>> instance.count(ord('$'))
0
>>> instance.count(b'Hello', endex=10)
1
>>> instance.count(b'Hello', endex=3)
0
>>> instance.count(b'World', start=3)
1
>>> instance.count(b'World', start=10)
0
>>> instance.count(b',', start=3, endex=10)
1
>>> instance.count(b',', start=8, endex=10)
0
```

abstract decode(*encoding='utf-8'*, *errors='strict'*)

Decode the bytes using the codec registered for encoding.

encoding

The encoding with which to decode the bytes.

errors

The error handling scheme to use for the handling of decoding errors. The default is 'strict' meaning that decoding errors raise a UnicodeDecodeError. Other possible values are 'ignore' and 'replace' as well as any other name registered with codecs.register_error that can handle UnicodeDecodeErrors.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.decode()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract endswith(*token*)

Return True if B ends with the specified suffix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. suffix can also be a tuple of bytes to try.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.endswith()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property f_contiguous: bool

Contiguous array, *Fortran* language style.

Type
bool

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

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.find()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property format: str

A string containing the format (in struct module style)
for each element in the view.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.format()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

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

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the subsection is not found.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.index()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isalnum()

Return True if all characters in B are alphanumeric and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isalnum()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isalpha()

Return True if all characters in B are alphabetic and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isalpha()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isascii()

Return True if B is empty or all characters in B are ASCII, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isascii()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

> **Warning:** This method documentation is just a stub, copied directly from `str.isdecimal()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isdigit()

Return True if all characters in B are digits and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isdigit()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

> **Warning:** This method documentation is just a stub, copied directly from `str.isidentifier()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract islower()

Return True if all cased characters in B are lowercase and there is at least one cased character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.islower()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

> **Warning:** This method documentation is just a stub, copied directly from `str.isnumeric()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

> **Warning:** This method documentation is just a stub, copied directly from `str.isprintable()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isspace()

Return True if all characters in B are whitespace and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isspace()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract istitle()

Return True if B is a titlecased string and there is at least one character in B, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.istitle()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isupper()

Return True if all cased characters in B are uppercase and there is at least one cased character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isupper()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property itemsize: int

The size in bytes of each element of the memoryview.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.itemsize()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract ljust(*width*, *fillchar=b' '*, *factory=<class 'bytes'>*)

Return a left-justified string of length width.

Padding is done using the specified fill character.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.ljust()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract lower()

Return a copy of B with all ASCII characters converted to lowercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.lower()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract lstrip(*chars=None*, *factory=<class 'bytes'>*)

Strip leading bytes contained in the argument.

If the argument is omitted or None, strip leading ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.lstrip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract static maketrans(*chars_from*, *chars_to*)

Return a translation table useable for the bytes or bytearray translate method.

The returned table will be one where each byte in frm is mapped to the byte at the same position in to.

The bytes objects frm and to must be of the same length.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.maketrans()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property nbytes: int

The amount of space in bytes that the array would use in
a contiguous representation.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.nbytes()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property ndim: int

An integer indicating how many dimensions of a multi-dimensional
array the memory represents.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.ndim()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property obj: ByteString | memoryview | None

The underlying object of the memoryview.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.obj()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract partition(*sep*, *factory*=<*class 'bytes'*>)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original bytes object and two empty bytes objects.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.partition()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property readonly: bool

A bool indicating whether the memory is read only.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.readonly()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract release()

Release the underlying buffer exposed by the memoryview object.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.release()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract removeprefix(*prefix*, *factory*=<*class 'bytes'*>)

Return a bytes object with the given prefix string removed if present.

If the bytes starts with the prefix string, return bytes[len(prefix):]. Otherwise, return a copy of the original bytes.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.removeprefix()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract removesuffix(*suffix*, *factory*=<*class 'bytes'*>)

Return a bytes object with the given suffix string removed if present.

If the bytes ends with the suffix string and that suffix is not empty, return bytes[:-len(prefix)]. Otherwise, return a copy of the original bytes.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.removesuffix()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract replace(*old*, *new*, *count=None*, *start=None*, *endex=None*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

Warning: This method documentation is just a stub, copied directly from `bytes.replace()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

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

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

Warning: This method documentation is just a stub, copied directly from `bytes.rfind()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

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

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raise ValueError when the subsection is not found.

Warning: This method documentation is just a stub, copied directly from `bytes.rindex()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract rjust(*width*, *fillchar=b' '*, *factory=<class 'bytes'>*)

Return a right-justified string of length width.

Padding is done using the specified fill character.

Warning: This method documentation is just a stub, copied directly from `bytes.rjust()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract rpartition(*sep*, *factory=<class 'bytes'>*)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty bytes objects and the original bytes object.

Warning: This method documentation is just a stub, copied directly from `bytes.rpartition()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract rstrip(*chars=None*, *factory=<class 'bytes'>*)

Strip trailing bytes contained in the argument.

If the argument is omitted or None, strip trailing ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rstrip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property shape: Tuple[int]

A tuple of ndim integers giving the shape of the memory
as an N-dimensional array.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.shape()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract startswith(*token*)

Return True if B starts with the specified prefix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. prefix can also be a tuple of bytes to try.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.startswith()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property strides: Tuple[int]

A tuple of ndim integers giving the size in bytes to access
each element for each dimension of the array.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.strides()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract strip(*chars=None*, *factory=<class 'bytes'>*)

Strip leading and trailing bytes contained in the argument.

If the argument is omitted or None, strip leading and trailing ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.strip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property suboffsets: Tuple

A tuple of integers used internally for PIL-style arrays.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.suboffsets()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract swapcase()

Return a copy of B with uppercase ASCII characters converted to lowercase ASCII and vice versa.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.swapcase()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract title()

Return a titlecased version of B, i.e. ASCII words start with uppercase characters, all remaining cased characters have lowercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.title()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract tobytes()

Return the data in the buffer as a byte string.

Order can be {'C', 'F', 'A'}. When order is 'C' or 'F', the data of the original array is converted to C or Fortran order. For contiguous views, 'A' returns an exact copy of the physical memory. In particular, in-memory Fortran order is preserved. For non-contiguous views, the data is converted to C first. order=None is the same as order='C'.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.tobytes()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract tolist()

Return the data in the buffer as a list of elements.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.tolist()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract translate(*table*)

Return a copy with each character mapped by the given translation table.

table
Translation table, which must be a bytes object of length 256.

All characters occurring in the optional argument delete are removed. The remaining characters are mapped through the given translation table.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.translate()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract upper()

Return a copy of B with all ASCII characters converted to uppercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.upper()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract zfill(*width*, *factory*=<*class 'bytes'*>)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The original string is never truncated.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.zfill()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

cbytesparse.base.BaseInplaceView

class cbytesparse.base.**BaseInplaceView**(*wrapped*)

Provides inplace methods to a byte buffer.

Standard Python *byte-like* objects (e.g. `memoryview` and `bytearray`) only provide a very small set of methods for *inplace* editing of their underlying bytes buffer.

This wrapper class adds a low-level implementation of those methods to anything supporting the *buffer protocol* with a mutable buffer.

Note: Editing support is only limited to the existing buffer items, i.e. the wrapper and its underlying buffer cannot be resized via the methods provided by this class, just like with standard memory views.

Parameters

wrapped (*byte-like*) – The target object supporting the *buffer protocol*.

Examples

```
>>> from cbytesparse import InplaceView
>>> buffer = bytearray(b'Hello, World!')
>>> wrapped = InplaceView(buffer)
>>> wrapped.lower()
>>> buffer
bytearray(b'hello, world!')
>>> wrapped.replace(b'l', b'p')
>>> buffer
bytearray(b'heppo, worpd!')
>>> wrapped.find(b'w')
7
>>> wrapped.rfind(b'o')
8
>>> wrapped.count(b'o')
2
```

Methods

__init__	
capitalize	byte-like: First character capitalized, the rest lowercase.
center	Return a centered string of length width.
contains	Contains a substring.
count	Counts token occurrences.
decode	Decode the bytes using the codec registered for encoding.
endswith	Return True if B ends with the specified suffix, False otherwise.
find	Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end].
index	Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end].
isalnum	Return True if all characters in B are alphanumeric and there is at least one character in B, False otherwise.
isalpha	Return True if all characters in B are alphabetic and there is at least one character in B, False otherwise.
isascii	Return True if B is empty or all characters in B are ASCII, False otherwise.
isdecimal	Return True if the string is a decimal string, False otherwise.
isdigit	Return True if all characters in B are digits and there is at least one character in B, False otherwise.
isidentifier	Return True if the string is a valid Python identifier, False otherwise.
islower	Return True if all cased characters in B are lowercase and there is at least one cased character in B, False otherwise.
isnumeric	Return True if the string is a numeric string, False otherwise.
isprintable	Return True if the string is printable, False otherwise.
isspace	Return True if all characters in B are whitespace and there is at least one character in B, False otherwise.
istitle	Return True if B is a titlecased string and there is at least one character in B, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones.
isupper	Return True if all cased characters in B are uppercase and there is at least one cased character in B, False otherwise.
ljust	Return a left-justified string of length width.
lower	Return a copy of B with all ASCII characters converted to lowercase.
lstrip	Strip leading bytes contained in the argument.
maketrans	Return a translation table useable for the bytes or bytearray translate method.

continues on next page

Table 2 – continued from previous page

partition	Partition the bytes into three parts using the given separator.
release	Release the underlying buffer exposed by the memoryview object.
removeprefix	Return a bytes object with the given prefix string removed if present.
removesuffix	Return a bytes object with the given suffix string removed if present.
replace	Return a copy with all occurrences of substring old replaced by new.
rfind	Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end].
rindex	Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end].
rjust	Return a right-justified string of length width.
rpartition	Partition the bytes into three parts using the given separator.
rstrip	Strip trailing bytes contained in the argument.
startswith	Return True if B starts with the specified prefix, False otherwise.
strip	Strip leading and trailing bytes contained in the argument.
swapcase	Return a copy of B with uppercase ASCII characters converted to lowercase ASCII and vice versa.
title	Return a titlecased version of B, i.e. ASCII words start with uppercase characters, all remaining cased characters have lowercase.
tobytes	Return the data in the buffer as a byte string.
tolist	Return the data in the buffer as a list of elements.
toreadonly	Return a readonly version of the memoryview.
translate	Return a copy with each character mapped by the given translation table.
upper	Return a copy of B with all ASCII characters converted to uppercase.
zfill	Pad a numeric string with zeros on the left, to fill a field of the given width.

Attributes

c_contiguous	Contiguous array, *C* language style.
contiguous	Contiguous array.
f_contiguous	Contiguous array, *Fortran* language style.
format	A string containing the format (in struct module style)
itemsize	The size in bytes of each element of the memoryview.
nbytes	The amount of space in bytes that the array would use in
ndim	An integer indicating how many dimensions of a multi-dimensional
obj	The underlying object of the memoryview.
readonly	A bool indicating whether the memory is read only.
shape	A tuple of ndim integers giving the shape of the memory
strides	A tuple of ndim integers giving the size in bytes to access
suboffsets	A tuple of integers used internally for PIL-style arrays.

abstract __bool__()

Has any items.

Returns

bool – Has any items.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'')
>>> bool(instance)
False
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> bool(instance)
True
```

abstract __bytes__()

Creates a bytes clone.

Returns

`bytes` – Cloned data.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(bytearray(b''))
>>> bytes(instance)
b''
```

```
>>> instance = BytesMethods(bytearray(b'Hello, World!'))
>>> bytes(instance)
b'Hello, World!'
```

classmethod **__class_getitem__**(*params*)

Parameterizes a generic class.

At least, parameterizing a generic class is the *main* thing this method does. For example, for some generic class *Foo*, this is called when we do *Foo[int]* - there, with *cls=Foo* and *params=int*.

However, note that this method is also called when defining generic classes in the first place with *class Foo(Generic[T]): ...*.

abstract **__contains__**(*token*)

Checks if some items are contained.

Parameters
token (*byte-like*) – Token to find.

Returns
bool – Token is contained.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> b'World' in instance
True
>>> b'$' in instance
False
>>> ord('o') in instance
True
>>> ord('$') in instance
False
```

abstract **__eq__**(*other*)

Equality comparison.

Parameters
other (*byte-like*) – Data to compare with *self*.

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

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> data = bytearray(b'Hello, World!')
>>> instance = BytesMethods(data)
>>> instance == data
True
>>> instance == memoryview(data)
True
>>> instance == b'Something else'
False
```

abstract __ge__(*other*)

Return self>=value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__ge__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

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.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance[7]  # -> ord('W') = 87
121
>>> bytes(instance[:3])
b'Hel'
>>> bytes(instance[3:10])
b'lo, Wor'
>>> bytes(instance[-1:])
b'!'
>>> bytes(instance[2:10:3])
b'l,o'
>>> bytes(instance[3:10:2])
b'l,Wr'
```

abstract __gt__(*other*)

Return self>value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__gt__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

__hash__ = None

abstract __init__(*wrapped*)

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

abstract __iter__()

Iterates over values.

Yields

int – Value as byte integer.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> list(instance)
[72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]
```

abstract __le__(*other*)

Return self<=value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__le__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract __len__()

Actual length.

Computes the actual length of the wrapped data object.

Returns

int – Data length.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'')
>>> len(instance)
0
```

```
>>> instance = BytesMethods(bytes(7))
>>> len(instance)
7
```

```
>>> instance = BytesMethods(memoryview(b'Hello, World!'))
>>> len(instance)
13
```

abstract __lt__(*other*)

Return self<value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__lt__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract __ne__(*other*)

Inquality comparison.

Parameters
other (*byte-like*) – Data to compare with *self*.

Returns
bool – *self* is not equal to *other*.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> data = bytearray(b'Hello, World!')
>>> instance = BytesMethods(data)
>>> instance != data
False
>>> instance != memoryview(data)
False
>>> instance != b'Something else'
True
```

abstract __reversed__()

Iterates over values, reversed order.

Yields
int – Value as byte integer.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> list(instance)
[72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]
>>> list(reversed(instance))
[33, 100, 108, 114, 111, 87, 32, 44, 111, 108, 108, 101, 72]
```

abstract __sizeof__()

int: Allocated byte size.

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 property c_contiguous: bool

Contiguous array, *C* language style.

Type
bool

abstract capitalize()

byte-like: First character capitalized, the rest lowercase.

abstract center(*width*, *fillchar=b' '*, *factory=<class 'bytes'>*)

Return a centered string of length width.

Padding is done using the specified fill character.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.center()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract contains(*token*, *start=None*, *endex=None*)

Contains a substring.

Parameters

- **token** (*byte-like*) – Token to search.
- **start** (*int*) – Inclusive start of the searched range, `None` to ignore.
- **endex** (*int*) – Exclusive end of the searched range, `None` to ignore.

Returns
bool – Token contained within the wrapped object.

See also:

__contains__()

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance.contains(b'World')
True
>>> instance.contains(b'$')
False
>>> instance.contains(ord('o'))
True
>>> instance.contains(ord('$'))
False
>>> instance.contains(b'Hello', endex=10)
True
>>> instance.contains(b'Hello', endex=3)
False
>>> instance.contains(b'World', start=3)
True
>>> instance.contains(b'World', start=10)
False
>>> instance.contains(b',', start=3, endex=10)
True
>>> instance.contains(b',', start=8, endex=10)
False
```

abstract property contiguous: bool

Contiguous array.

Type
bool

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

Counts token occurrences.

Parameters

- **token** (*byte-like*) – Token to count.
- **start** (*int*) – Inclusive start of the searched range, None to ignore.
- **endex** (*int*) – Exclusive end of the searched range, None to ignore.

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

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance.count(b'o')
2
>>> instance.count(b'l')
3
>>> instance.count(b'll')
3
>>> instance.count(b'World')
1
>>> instance.count(b'$')
0
>>> instance.count(ord('o'))
1
>>> instance.count(ord('$'))
0
>>> instance.count(b'Hello', endex=10)
1
>>> instance.count(b'Hello', endex=3)
0
>>> instance.count(b'World', start=3)
1
>>> instance.count(b'World', start=10)
0
>>> instance.count(b',', start=3, endex=10)
1
>>> instance.count(b',', start=8, endex=10)
0
```

abstract decode(*encoding='utf-8'*, *errors='strict'*)

Decode the bytes using the codec registered for encoding.

encoding
The encoding with which to decode the bytes.

errors
The error handling scheme to use for the handling of decoding errors. The default is 'strict' meaning that decoding errors raise a UnicodeDecodeError. Other possible values are 'ignore' and 'replace' as well as any other name registered with codecs.register_error that can handle UnicodeDecodeErrors.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.decode()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract endswith(*token*)

Return True if B ends with the specified suffix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. suffix can also be a tuple of bytes to try.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.endswith()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property f_contiguous: bool

Contiguous array, *Fortran* language style.

Type
bool

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

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.find()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property format: str

A string containing the format (in struct module style)
for each element in the view.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.format()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

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

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the subsection is not found.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.index()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isalnum()

Return True if all characters in B are alphanumeric and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isalnum()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isalpha()

Return True if all characters in B are alphabetic and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isalpha()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isascii()

Return True if B is empty or all characters in B are ASCII, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isascii()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

> **Warning:** This method documentation is just a stub, copied directly from `str.isdecimal()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isdigit()

Return True if all characters in B are digits and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isdigit()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

> **Warning:** This method documentation is just a stub, copied directly from `str.isidentifier()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract islower()

Return True if all cased characters in B are lowercase and there is at least one cased character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.islower()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

> **Warning:** This method documentation is just a stub, copied directly from `str.isnumeric()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

> **Warning:** This method documentation is just a stub, copied directly from `str.isprintable()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isspace()

Return True if all characters in B are whitespace and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isspace()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract istitle()

Return True if B is a titlecased string and there is at least one character in B, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.istitle()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract isupper()

Return True if all cased characters in B are uppercase and there is at least one cased character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isupper()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property itemsize: int

The size in bytes of each element of the memoryview.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.itemsize()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract ljust(*width*, *fillchar=b' '*, *factory=<class 'bytes'>*)

Return a left-justified string of length width.

Padding is done using the specified fill character.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.ljust()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract lower()

Return a copy of B with all ASCII characters converted to lowercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.lower()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract lstrip(*chars=None*, *factory=<class 'bytes'>*)

Strip leading bytes contained in the argument.

If the argument is omitted or None, strip leading ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.lstrip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract static maketrans(*chars_from*, *chars_to*)

Return a translation table useable for the bytes or bytearray translate method.

The returned table will be one where each byte in frm is mapped to the byte at the same position in to.

The bytes objects frm and to must be of the same length.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.maketrans()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property nbytes: int

The amount of space in bytes that the array would use in
a contiguous representation.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.nbytes()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property ndim: int

An integer indicating how many dimensions of a multi-dimensional
array the memory represents.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.ndim()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property obj: ByteString | memoryview | None

The underlying object of the memoryview.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.obj()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract partition(*sep*, *factory=<class 'bytes'>*)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original bytes object and two empty bytes objects.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.partition()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property readonly: bool

A bool indicating whether the memory is read only.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.readonly()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract release()

Release the underlying buffer exposed by the memoryview object.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.release()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract removeprefix(*prefix*, *factory*=<*class 'bytes'*>)

Return a bytes object with the given prefix string removed if present.

If the bytes starts with the prefix string, return bytes[len(prefix):]. Otherwise, return a copy of the original bytes.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.removeprefix()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract removesuffix(*suffix*, *factory*=<*class 'bytes'*>)

Return a bytes object with the given suffix string removed if present.

If the bytes ends with the suffix string and that suffix is not empty, return bytes[:-len(prefix)]. Otherwise, return a copy of the original bytes.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.removesuffix()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract replace(*old*, *new*, *count=None*, *start=None*, *endex=None*)

Return a copy with all occurrences of substring old replaced by new.

count
: Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.replace()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

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

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rfind()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

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

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raise ValueError when the subsection is not found.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rindex()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract rjust(*width*, *fillchar=b' '*, *factory=<class 'bytes'>*)

Return a right-justified string of length width.

Padding is done using the specified fill character.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rjust()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract rpartition(*sep*, *factory=<class 'bytes'>*)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty bytes objects and the original bytes object.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rpartition()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract rstrip(*chars=None*, *factory=<class 'bytes'>*)

Strip trailing bytes contained in the argument.

If the argument is omitted or None, strip trailing ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rstrip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract property shape: Tuple[int]

A tuple of ndim integers giving the shape of the memory
as an N-dimensional array.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.shape()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`abstract startswith`(*token*)

Return True if B starts with the specified prefix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. prefix can also be a tuple of bytes to try.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.startswith()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`abstract property strides: Tuple[int]`

A tuple of ndim integers giving the size in bytes to access
each element for each dimension of the array.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.strides()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`abstract strip`(*chars=None*, *factory=<class 'bytes'>*)

Strip leading and trailing bytes contained in the argument.

If the argument is omitted or None, strip leading and trailing ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.strip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`abstract property suboffsets: Tuple`

A tuple of integers used internally for PIL-style arrays.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.suboffsets()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`abstract swapcase`()

Return a copy of B with uppercase ASCII characters converted to lowercase ASCII and vice versa.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.swapcase()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`abstract title`()

Return a titlecased version of B, i.e. ASCII words start with uppercase characters, all remaining cased characters have lowercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.title()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract tobytes()

Return the data in the buffer as a byte string.

Order can be {'C', 'F', 'A'}. When order is 'C' or 'F', the data of the original array is converted to C or Fortran order. For contiguous views, 'A' returns an exact copy of the physical memory. In particular, in-memory Fortran order is preserved. For non-contiguous views, the data is converted to C first. order=None is the same as order='C'.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.tobytes()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract tolist()

Return the data in the buffer as a list of elements.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.tolist()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract toreadonly()

Return a readonly version of the memoryview.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.toreadonly()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract translate(*table*)

Return a copy with each character mapped by the given translation table.

table
Translation table, which must be a bytes object of length 256.

All characters occurring in the optional argument delete are removed. The remaining characters are mapped through the given translation table.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.translate()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract upper()

Return a copy of B with all ASCII characters converted to uppercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.upper()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

abstract zfill(*width*, *factory*=<*class 'bytes'*>)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The original string is never truncated.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.zfill()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

3.1.2 cbytesparse.c

Cython implementation.

Note: Wherever the Python implementation (i.e. `bytesparse` package) provides docstrings, no such docstrings are provided by the Cython implementation, which would result in duplicated docstrings, with maintenance efforts required.

Cython compiles docstrings at compile time, making impossible to patch docstrings dynamically with those from the parent Python implementation. This is also one of the reasons why the `cbytesparse.py` module exists.

Should you need actual documentation, instead of the stubs automatically generated by Sphinx, please refer to *the ``bytesparse`* documentation <https://bytesparse.readthedocs.io/>`_.

Classes

BlockView	Block viewer.
BytesMethods	Provides useful methods to a byte buffer.
InplaceView	Provides inplace methods to a byte buffer.
Memory	Virtual memory.
bytesparse	Wrapper for more *bytearray* compatibility.

cbytesparse.c.BlockView

class cbytesparse.c.**BlockView**

Block viewer.

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

Methods

__init__	
capitalize	
center	
check	Checks for data consistency.

continues on next page

Table 3 – continued from previous page

contains	
count	
decode	
endswith	
find	
index	
isalnum	
isalpha	
isascii	
isdecimal	
isdigit	
isidentifier	
islower	
isnumeric	
isprintable	
isspace	
istitle	
isupper	
ljust	
lower	
lstrip	
maketrans	
partition	
release	Forces object disposal.
removeprefix	
removesuffix	

continues on next page

Table 3 – continued from previous page

replace
rfind
rindex
rjust
rpartition
rstrip
startswith
strip
swapcase
title
tobytes
tolist
toreadonly
translate
upper
zfill

Attributes

acquired	Underlying block currently acquired.
c_contiguous	
contiguous	
endex	Slice exclusive end address.
endin	Slice inclusive end address.
f_contiguous	
format	
itemsize	
nbytes	
ndim	
obj	
readonly	
shape	
start	Slice inclusive start address.
strides	
suboffsets	

__bool__()

True if self else False

__bytes__()

Converts into bytes.

Returns

bytes – `bytes` clone of the viewed slice.

__contains__(*key*, /)

Return key in self.

__delitem__(*key*, /)

Delete self[key].

__eq__(*value*, /)

Return self==value.

__ge__(*value*, /)

Return self>=value.

__getattribute__(*name*, /)

Return getattr(self, name).

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

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

__hash__ = None

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

__iter__()
Implement iter(self).

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

__len__()
int: Slice length.

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

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

__new__(***kwargs*)

__reduce__()
BlockView.__reduce_cython__(self)

__reduce_cython__(*self*)

__repr__()
Return repr(self).

__reversed__()

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

__setstate__()
BlockView.__setstate_cython__(self, __pyx_state)

__setstate_cython__(*self*, *__pyx_state*)

__sizeof__()
int: Allocated byte size.

__str__()
Return str(self).

_memoryview
Python `memoryview` wrapper.

Type
memoryview

acquired

Underlying block currently acquired.

Type
bool

capitalize()

center(*width*, *fillchar*, *factory*)

check()

Checks for data consistency.

contains(*token*, *start*, *endex*)

count(*token*, *start*, *endex*)

decode(*encoding*, *errors*)

endex

Slice exclusive end address.

Type
int

endin

Slice inclusive end address.

Type
int

endswith(*token*)

find(*token*, *start*, *endex*)

index(*token*, *start*, *endex*)

isalnum()

isalpha()

isascii()

isdecimal()

isdigit()

isidentifier()

islower()

isnumeric()

isprintable()

isspace()

istitle()

isupper()

ljust(*width*, *fillchar*, *factory*)

lower()

lstrip(*chars*, *factory*)

static maketrans(*chars_from*, *chars_to*)

partition(*sep*, *factory*)

release()

Forces object disposal.

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

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

removeprefix(*prefix*, *factory*)

removesuffix(*suffix*, *factory*)

replace(*old*, *new*, *count*, *start*, *endex*)

rfind(*token*, *start*, *endex*)

rindex(*token*, *start*, *endex*)

rjust(*width*, *fillchar*, *factory*)

rpartition(*sep*, *factory*)

rstrip(*chars*, *factory*)

start

Slice inclusive start address.

Type
int

startswith(*token*)

strip(*chars*, *factory*)

swapcase()

title()

tobytes()

tolist()

toreadonly()

translate(*table*)

upper()

zfill(*width*, *factory*)

cbytesparse.c.BytesMethods

class cbytesparse.c.**BytesMethods**(*wrapped: ByteString*)

Provides useful methods to a byte buffer.

Please refer to *cbytesparse.py.BytesMethods* for more information.

See also:

cbytesparse.py.BytesMethods

Methods

__init__
capitalize
center
contains
count
decode
endswith
find
index
isalnum
isalpha
isascii
isdecimal
isdigit
isidentifier
islower
isnumeric
isprintable
isspace
istitle

continues on next page

Table 4 – continued from previous page

isupper
ljust
lower
lstrip
maketrans
partition
release
removeprefix
removesuffix
replace
rfind
rindex
rjust
rpartition
rstrip
startswith
strip
swapcase
title
tobytes
tolist
translate
upper
zfill

Attributes

c_contiguous
contiguous
f_contiguous
format
itemsize
nbytes
ndim
obj
readonly
shape
strides
suboffsets

__bool__()
: True if self else False

__bytes__()

__contains__(*key*, /)
: Return key in self.

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

__eq__(*value*, /)
: Return self==value.

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

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

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

__hash__ = None

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

__iter__()
Implement iter(self).

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

__len__()
Return len(self).

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

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

__new__(**kwargs*)

__reduce__()
BytesMethods.__reduce_cython__(self)

__reduce_cython__(*self*)

__reversed__()

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

__setstate__()
BytesMethods.__setstate_cython__(self, __pyx_state)

__setstate_cython__(*self*, *__pyx_state*)

__sizeof__()

capitalize()

center(*width*, *fillchar*, *factory*)

contains(*token*, *start*, *endex*)

count(*token*, *start*, *endex*)

decode(*encoding*, *errors*)

endswith(*token*)

find(*token*, *start*, *endex*)

index(*token*, *start*, *endex*)

isalnum()

isalpha()

isascii()

isdecimal()

isdigit()

isidentifier()

islower()

isnumeric()

isprintable()

isspace()

istitle()

isupper()

ljust(*width*, *fillchar*, *factory*)

lower()

lstrip(*chars*, *factory*)

static maketrans(*chars_from*, *chars_to*)

partition(*sep*, *factory*)

release()

removeprefix(*prefix*, *factory*)

removesuffix(*suffix*, *factory*)

replace(*old*, *new*, *count*, *start*, *endex*)

rfind(*token*, *start*, *endex*)

rindex(*token*, *start*, *endex*)

rjust(*width*, *fillchar*, *factory*)

rpartition(*sep*, *factory*)

rstrip(*chars*, *factory*)

startswith(*token*)

strip(*chars*, *factory*)

swapcase()

title()

tobytes()

tolist()

translate(*table*)

upper()

zfill(*width*, *factory*)

cbytesparse.c.InplaceView

class cbytesparse.c.**InplaceView**

Provides inplace methods to a byte buffer.

Please refer to *cbytesparse.py.InplaceView* for more information.

See also:

cbytesparse.py.InplaceView

Methods

__init__
capitalize
center
contains
count
decode
endswith
find
index
isalnum
isalpha
isascii
isdecimal
isdigit
isidentifier
islower
isnumeric
isprintable
isspace
istitle

continues on next page

Table 5 – continued from previous page

isupper
ljust
lower
lstrip
maketrans
partition
release
removeprefix
removesuffix
replace
rfind
rindex
rjust
rpartition
rstrip
startswith
strip
swapcase
title
tobytes
tolist
toreadonly
translate
upper
zfill

Attributes

c_contiguous
contiguous
f_contiguous
format
itemsize
nbytes
ndim
obj
readonly
shape
strides
suboffsets

__bool__()
: True if self else False

__bytes__()

__contains__(*key*, /)
: Return key in self.

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

__eq__(*value*, /)
: Return self==value.

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

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

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

__hash__ = None

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

__iter__()
Implement iter(self).

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

__len__()
Return len(self).

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

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

__new__(**kwargs)

__reduce__()
InplaceView.__reduce_cython__(self)

__reduce_cython__(*self*)

__reversed__()

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

__setstate__()
InplaceView.__setstate_cython__(self, __pyx_state)

__setstate_cython__(*self*, *__pyx_state*)

__sizeof__()

capitalize()

center(*width*, *fillchar*, *factory*)

contains(*token*, *start*, *endex*)

count(*token*, *start*, *endex*)

decode(*encoding*, *errors*)

endswith(*token*)

find(*token*, *start*, *endex*)

index(*token*, *start*, *endex*)

isalnum()

isalpha()

isascii()

isdecimal()

isdigit()

isidentifier()

islower()

isnumeric()

isprintable()

isspace()

istitle()

isupper()

ljust(*width*, *fillchar*, *factory*)

lower()

lstrip(*chars*, *factory*)

static maketrans(*chars_from*, *chars_to*)

partition(*sep*, *factory*)

release()

removeprefix(*prefix*, *factory*)

removesuffix(*suffix*, *factory*)

replace(*old*, *new*, *count*, *start*, *endex*)

rfind(*token*, *start*, *endex*)

rindex(*token*, *start*, *endex*)

rjust(*width*, *fillchar*, *factory*)

rpartition(*sep*, *factory*)

rstrip(*chars*, *factory*)

startswith(*token*)

strip(*chars*, *factory*)

swapcase()

title()

tobytes()

tolist()

toreadonly()

translate(*table*)

upper()

zfill(*width*, *factory*)

cbytesparse.c.Memory

class cbytesparse.c.**Memory**(*start: Address | None = None*, *endex: Address | None = None*)

Virtual memory.

Please refer to *cbytesparse.py.Memory* for more information.

See also:

cbytesparse.py.Memory

Methods

__init__
append
append_backup
append_restore
block_span
blocks
bound
clear
clear_backup
clear_restore
collapse_blocks
content_blocks
content_items
content_keys
content_values
copy
count
crop
crop_backup
crop_restore

continues on next page

Table 6 – continued from previous page

cut
delete
delete_backup
delete_restore
equal_span
extend
extend_backup
extend_restore
extract
fill
fill_backup
fill_restore
find
flood
flood_backup
flood_restore
from_blocks
from_bytes
from_items
from_memory
from_values
fromhex
gaps
get
hex
index

continues on next page

Table 6 – continued from previous page

insert
insert_backup
insert_restore
intervals
items
keys
peek
poke
poke_backup
poke_restore
pop
pop_backup
pop_restore
popitem
popitem_backup
popitem_restore
read
readinto
remove
remove_backup
remove_restore
reserve
reserve_backup
reserve_restore
reverse
rfind

continues on next page

Table 6 – continued from previous page

rindex
rvalues
setdefault
setdefault_backup
setdefault_restore
shift
shift_backup
shift_restore
to_blocks
to_bytes
update
update_backup
update_restore
validate
values
view
write
write_backup
write_restore

Attributes

bound_endex
bound_span
bound_start
content_endex
content_endin
content_parts
content_size
content_span
content_start
contiguous
endex
endin
span
start

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

__bool__()
True if self else False

__bytes__()

__contains__(*key*, /)
Return key in self.

__copy__()

__deepcopy__()

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

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

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

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

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

__hash__ = None

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

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

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

__iter__()
Implement iter(self).

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

__len__()
Return len(self).

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

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

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

__new__(***kwargs*)

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

__reduce__()
Memory.__reduce_cython__(self)

__repr__()
Return repr(self).

__reversed__()

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

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

__setstate__()
Memory.__setstate_cython__(self, __pyx_state)

__sizeof__()

__str__()
Return str(self).

_block_index_at(*address*)

_block_index_endex(*address*)

_block_index_start(*address*)

_prebound_endex(*start_min*, *size*)

_prebound_endex_backup(*start_min*, *size*)

_prebound_start(*endex_max*, *size*)

_prebound_start_backup(*endex_max*, *size*)

append(*item*)

append_backup()

append_restore()

block_span(*address*)

blocks(*start*, *endex*)

bound(*start*, *endex*)

clear(*start*, *endex*)

clear_backup(*start*, *endex*)

clear_restore(*backup*)

classmethod collapse_blocks(*blocks*)

content_blocks(*block_index_start*, *block_index_endex*, *block_index_step*)

content_items(*start*, *endex*)

content_keys(*start*, *endex*)

content_values(*start*, *endex*)

copy()

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

crop(*start*, *endex*)

crop_backup(*start*, *endex*)

crop_restore(*backup_start*, *backup_endex*)

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

delete(*start*, *endex*)

delete_backup(*start*, *endex*)

delete_restore(*backup*)

equal_span(*address*)

extend(*items*, *offset*)

extend_backup(*offset*)

extend_restore(*content_endex*)

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

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

fill_backup(*start*, *endex*)

fill_restore(*backup*)

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

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

flood_backup(*start*, *endex*)

flood_restore(*gaps*)

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

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

classmethod from_items(*items*, *offset*, *start*, *endex*, *validate*)

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

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

classmethod fromhex(*string*)

gaps(*start*, *endex*)

get(*address*, *default*)

hex(**args*)

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

insert(*address*, *data*)

insert_backup(*address*, *data*)

insert_restore(*address*, *backup*)

intervals(*start*, *endex*)

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

keys(*start*, *endex*)

peek(*address*)

poke(*address*, *item*)

poke_backup(*address*)

poke_restore(*address*, *item*)

pop(*address*, *default*)

pop_backup(*address*)

pop_restore(*address*, *item*)

popitem()

popitem_backup()

popitem_restore(*address*, *item*)

read(*address*, *size*)

readinto(*address*, *buffer*)

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

remove_backup(*item*, *start*, *endex*)

remove_restore(*backup*)

reserve(*address*, *size*)

reserve_backup(*address*, *size*)

reserve_restore(*address*, *backup*)

reverse()

rfind(*item*, *start*, *endex*)

rindex(*item*, *start*, *endex*)

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

setdefault(*address*, *default*)

setdefault_backup(*address*)

setdefault_restore(*address*, *item*)

shift(*offset*)

shift_backup(*offset*)

shift_restore(*offset*, *backup*)

to_blocks(*start*, *endex*)

to_bytes(*start*, *endex*)

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

update_backup(*data*, *clear*, ***kwargs*)

update_restore(*backups*)

validate()

values(*start*, *endex*, *pattern*)

view(*start*, *endex*)

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

write_backup(*address*, *data*, *clear*)

write_restore(*backups*)

cbytesparse.c.bytesparse

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

Wrapper for more *bytearray* compatibility.

Please refer to *cbytesparse.py.bytesparse* for more information.

See also:

cbytesparse.py.bytesparse

Methods

__init__
append
append_backup
append_restore
block_span
blocks
bound
clear
clear_backup
clear_restore
collapse_blocks
content_blocks
content_items
content_keys

continues on next page

Table 7 – continued from previous page

content_values
copy
count
crop
crop_backup
crop_restore
cut
delete
delete_backup
delete_restore
equal_span
extend
extend_backup
extend_restore
extract
fill
fill_backup
fill_restore
find
flood
flood_backup
flood_restore
from_blocks
from_bytes
from_items
from_memory

continues on next page

Table 7 – continued from previous page

from_values
fromhex
gaps
get
hex
index
insert
insert_backup
insert_restore
intervals
items
keys
peek
poke
poke_backup
poke_restore
pop
pop_backup
pop_restore
popitem
popitem_backup
popitem_restore
read
readinto
remove
remove_backup

continues on next page

Table 7 – continued from previous page

remove_restore
reserve
reserve_backup
reserve_restore
reverse
rfind
rindex
rvalues
setdefault
setdefault_backup
setdefault_restore
shift
shift_backup
shift_restore
to_blocks
to_bytes
update
update_backup
update_restore
validate
values
view
write
write_backup
write_restore

Attributes

bound_endex
bound_span
bound_start
content_endex
content_endin
content_parts
content_size
content_span
content_start
contiguous
endex
endin
span
start

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

__bool__()
True if self else False

__bytes__()

__contains__(*key*, /)
Return key in self.

__copy__()

__deepcopy__()

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

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

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

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

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

__hash__ = None

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

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

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

__iter__()
Implement iter(self).

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

__len__()
Return len(self).

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

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

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

__new__(***kwargs*)

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

__reduce__()
bytesparse.__reduce_cython__(self)

__repr__()
Return repr(self).

__reversed__()

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

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

__setstate__()
bytesparse.__setstate_cython__(self, __pyx_state)

__sizeof__()

__str__()
Return str(self).

_block_index_at(*address*)

_block_index_endex(*address*)

_block_index_start(*address*)

_prebound_endex(*start_min*, *size*)

_prebound_endex_backup(*start_min*, *size*)

_prebound_start(*endex_max*, *size*)

_prebound_start_backup(*endex_max*, *size*)

_rectify_address(*address*)

_rectify_span(*start*, *endex*)

append(*item*)

append_backup()

append_restore()

block_span(*address*)

blocks(*start*, *endex*)

bound(*start*, *endex*)

clear(*start*, *endex*)

clear_backup(*start*, *endex*)

clear_restore(*backup*)

classmethod collapse_blocks(*blocks*)

content_blocks(*block_index_start*, *block_index_endex*, *block_index_step*)

content_items(*start*, *endex*)

content_keys(*start*, *endex*)

content_values(*start*, *endex*)

copy()

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

crop(*start*, *endex*)

crop_backup(*start*, *endex*)

crop_restore(*backup_start*, *backup_endex*)

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

delete(*start*, *endex*)

delete_backup(*start*, *endex*)

delete_restore(*backup*)

equal_span(*address*)

extend(*items*, *offset*)

extend_backup(*offset*)

extend_restore(*content_endex*)

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

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

fill_backup(*start*, *endex*)

fill_restore(*backup*)

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

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

flood_backup(*start*, *endex*)

flood_restore(*gaps*)

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

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

classmethod from_items(*items*, *offset*, *start*, *endex*, *validate*)

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

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

classmethod fromhex(*string*)

gaps(*start*, *endex*)

get(*address*, *default*)

hex(**args*)

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

insert(*address*, *data*)

insert_backup(*address*, *data*)

insert_restore(*address*, *backup*)

intervals(*start*, *endex*)

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

keys(*start*, *endex*)

peek(*address*)

poke(*address*, *item*)

poke_backup(*address*)

poke_restore(*address*, *item*)

pop(*address*, *default*)

pop_backup(*address*)

pop_restore(*address*, *item*)

popitem()

popitem_backup()

popitem_restore(*address*, *item*)

read(*address*, *size*)

readinto(*address*, *buffer*)

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

remove_backup(*item*, *start*, *endex*)

remove_restore(*backup*)

reserve(*address*, *size*)

reserve_backup(*address*, *size*)

reserve_restore(*address*, *backup*)

reverse()

rfind(*item*, *start*, *endex*)

rindex(*item*, *start*, *endex*)

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

setdefault(*address*, *default*)

setdefault_backup(*address*)

setdefault_restore(*address*, *item*)

shift(*offset*)

shift_backup(*offset*)

shift_restore(*offset*, *backup*)

to_blocks(*start*, *endex*)

to_bytes(*start*, *endex*)

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

update_backup(*data*, *clear*, ***kwargs*)

update_restore(*backups*)

validate()

values(*start*, *endex*, *pattern*)

view(*start*, *endex*)

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

write_backup(*address*, *data*, *clear*)

write_restore(*backups*)

3.1.3 cbytesparse.py

Python wrappers.

Useful for dynamically declared stuff, e.g. docstrings and return types.

Classes

BytesMethods	Provides useful methods to a byte buffer.
InplaceView	Provides inplace methods to a byte buffer.
Memory	Mutable virtual memory.
bytesparse	Wrapper for more *bytearray* compatibility.

cbytesparse.py.BytesMethods

class cbytesparse.py.**BytesMethods**(*wrapped*)

Provides useful methods to a byte buffer.

Python's `memoryview` and most *byte-like* objects do not provide many useful methods found instead within the `bytes` or `str` APIs.

This wrapper class adds a low-level implementation of those methods to anything supporting the *buffer protocol*.

Parameters

wrapped (*byte-like*) – The target object supporting the *buffer protocol*.

Examples

```
>>> from cbytesparse import BytesMethods
>>> import numpy
>>> numbers = list(b'ABC')
>>> numbers
[65, 66, 67]
>>> data = numpy.array(numbers, dtype=numpy.ubyte)
>>> data
array([65, 66, 67], dtype=uint8)
```

(continues on next page)

(continued from previous page)

```
>>> data.lower()  # noqa
Traceback (most recent call last):
    ...
AttributeError: 'numpy.ndarray' object has no attribute 'lower'
>>> wrapped = BytesMethods(data)  # noqa
>>> bytes(wrapped.lower())
b'abc'
>>> wrapped = BytesMethods(memoryview(data))
>>> bytes(wrapped.lower())
b'abc'
```

Methods

__init__	
capitalize	byte-like: First character capitalized, the rest lowercase.
center	Return a centered string of length width.
contains	Contains a substring.
count	Counts token occurrences.
decode	Decode the bytes using the codec registered for encoding.
endswith	Return True if B ends with the specified suffix, False otherwise.
find	Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end].
index	Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end].
isalnum	Return True if all characters in B are alphanumeric and there is at least one character in B, False otherwise.
isalpha	Return True if all characters in B are alphabetic and there is at least one character in B, False otherwise.
isascii	Return True if B is empty or all characters in B are ASCII, False otherwise.
isdecimal	Return True if the string is a decimal string, False otherwise.
isdigit	Return True if all characters in B are digits and there is at least one character in B, False otherwise.
isidentifier	Return True if the string is a valid Python identifier, False otherwise.
islower	Return True if all cased characters in B are lowercase and there is at least one cased character in B, False otherwise.
isnumeric	Return True if the string is a numeric string, False otherwise.
isprintable	Return True if the string is printable, False otherwise.
isspace	Return True if all characters in B are whitespace and there is at least one character in B, False otherwise.

continues on next page

Table 8 – continued from previous page

istitle	Return True if B is a titlecased string and there is at least one character in B, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones.
isupper	Return True if all cased characters in B are uppercase and there is at least one cased character in B, False otherwise.
ljust	Return a left-justified string of length width.
lower	Return a copy of B with all ASCII characters converted to lowercase.
lstrip	Strip leading bytes contained in the argument.
maketrans	Return a translation table useable for the bytes or bytearray translate method.
partition	Partition the bytes into three parts using the given separator.
release	Release the underlying buffer exposed by the memoryview object.
removeprefix	Return a bytes object with the given prefix string removed if present.
removesuffix	Return a bytes object with the given suffix string removed if present.
replace	Return a copy with all occurrences of substring old replaced by new.
rfind	Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end].
rindex	Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end].
rjust	Return a right-justified string of length width.
rpartition	Partition the bytes into three parts using the given separator.
rstrip	Strip trailing bytes contained in the argument.
startswith	Return True if B starts with the specified prefix, False otherwise.
strip	Strip leading and trailing bytes contained in the argument.
swapcase	Return a copy of B with uppercase ASCII characters converted to lowercase ASCII and vice versa.
title	Return a titlecased version of B, i.e. ASCII words start with uppercase characters, all remaining cased characters have lowercase.
tobytes	Return the data in the buffer as a byte string.
tolist	Return the data in the buffer as a list of elements.
translate	Return a copy with each character mapped by the given translation table.
upper	Return a copy of B with all ASCII characters converted to uppercase.
zfill	Pad a numeric string with zeros on the left, to fill a field of the given width.

Attributes

c_contiguous	Contiguous array, *C* language style.
contiguous	Contiguous array.
f_contiguous	Contiguous array, *Fortran* language style.
format	A string containing the format (in struct module style)
itemsize	The size in bytes of each element of the memoryview.
nbytes	The amount of space in bytes that the array would use in
ndim	An integer indicating how many dimensions of a multi-dimensional
obj	The underlying object of the memoryview.
readonly	A bool indicating whether the memory is read only.
shape	A tuple of ndim integers giving the shape of the memory
strides	A tuple of ndim integers giving the size in bytes to access
suboffsets	A tuple of integers used internally for PIL-style arrays.

__bool__()

Has any items.

Returns

bool – Has any items.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'')
>>> bool(instance)
False
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> bool(instance)
True
```

__bytes__()

Creates a bytes clone.

Returns

`bytes` – Cloned data.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(bytearray(b''))
>>> bytes(instance)
b''
```

```
>>> instance = BytesMethods(bytearray(b'Hello, World!'))
>>> bytes(instance)
b'Hello, World!'
```

classmethod __class_getitem__(*params*)

Parameterizes a generic class.

At least, parameterizing a generic class is the *main* thing this method does. For example, for some generic class *Foo*, this is called when we do *Foo[int]* - there, with *cls=Foo* and *params=int*.

However, note that this method is also called when defining generic classes in the first place with *class Foo(Generic[T]): ...*.

__contains__(*token*)

Checks if some items are contained.

Parameters
token (*byte-like*) – Token to find.

Returns
bool – Token is contained.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> b'World' in instance
True
>>> b'$' in instance
False
>>> ord('o') in instance
True
>>> ord('$') in instance
False
```

__eq__(*other*)

Equality comparison.

Parameters
other (*byte-like*) – Data to compare with *self*.

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

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> data = bytearray(b'Hello, World!')
>>> instance = BytesMethods(data)
>>> instance == data
True
>>> instance == memoryview(data)
True
>>> instance == b'Something else'
False
```

__ge__(*other*)

Return self>=value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__ge__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

__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.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance[7]  # -> ord('W') = 87
121
>>> bytes(instance[:3])
b'Hel'
>>> bytes(instance[3:10])
b'lo, Wor'
>>> bytes(instance[-1:])
b'!'
>>> bytes(instance[2:10:3])
b'l,o'
>>> bytes(instance[3:10:2])
b'l,Wr'
```

__gt__(*other*)

Return self>value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__gt__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

__hash__ = None

__init__(*wrapped*)

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__iter__()

Iterates over values.

Yields

int – Value as byte integer.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> list(instance)
[72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]
```

__le__(*other*)

Return self<=value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__le__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

__len__()

Actual length.

Computes the actual length of the wrapped data object.

Returns

int – Data length.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'')
>>> len(instance)
0
```

```
>>> instance = BytesMethods(bytes(7))
>>> len(instance)
7
```

```
>>> instance = BytesMethods(memoryview(b'Hello, World!'))
>>> len(instance)
13
```

__lt__(*other*)

Return self<value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__lt__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

__ne__(*other*)

Inquality comparison.

Parameters
other (*byte-like*) – Data to compare with *self*.

Returns
bool – *self* is not equal to *other*.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> data = bytearray(b'Hello, World!')
>>> instance = BytesMethods(data)
>>> instance != data
False
>>> instance != memoryview(data)
False
>>> instance != b'Something else'
True
```

__reversed__()

Iterates over values, reversed order.

Yields
int – Value as byte integer.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> list(instance)
[72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]
>>> list(reversed(instance))
[33, 100, 108, 114, 111, 87, 32, 44, 111, 108, 108, 101, 72]
```

__sizeof__()

int: Allocated byte size.

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)

property c_contiguous: bool

Contiguous array, *C* language style.

Type
bool

capitalize()

byte-like: First character capitalized, the rest lowercase.

center(*width*, *fillchar=b' '*, *factory=<class 'bytes'>*)

Return a centered string of length width.

Padding is done using the specified fill character.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.center()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

contains(*token*, *start=None*, *endex=None*)

Contains a substring.

Parameters

- **token** (*byte-like*) – Token to search.
- **start** (*int*) – Inclusive start of the searched range, `None` to ignore.
- **endex** (*int*) – Exclusive end of the searched range, `None` to ignore.

Returns
bool – Token contained within the wrapped object.

See also:

__contains__()

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance.contains(b'World')
True
>>> instance.contains(b'$')
False
>>> instance.contains(ord('o'))
True
>>> instance.contains(ord('$'))
False
>>> instance.contains(b'Hello', endex=10)
True
>>> instance.contains(b'Hello', endex=3)
False
>>> instance.contains(b'World', start=3)
True
>>> instance.contains(b'World', start=10)
False
>>> instance.contains(b',', start=3, endex=10)
True
>>> instance.contains(b',', start=8, endex=10)
False
```

property contiguous: bool

Contiguous array.

Type
bool

count(*token*, *start=None*, *endex=None*)

Counts token occurrences.

Parameters

- **token** (*byte-like*) – Token to count.
- **start** (*int*) – Inclusive start of the searched range, None to ignore.
- **endex** (*int*) – Exclusive end of the searched range, None to ignore.

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

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance.count(b'o')
2
>>> instance.count(b'l')
3
>>> instance.count(b'll')
3
>>> instance.count(b'World')
1
>>> instance.count(b'$')
0
>>> instance.count(ord('o'))
1
>>> instance.count(ord('$'))
0
>>> instance.count(b'Hello', endex=10)
1
>>> instance.count(b'Hello', endex=3)
0
>>> instance.count(b'World', start=3)
1
>>> instance.count(b'World', start=10)
0
>>> instance.count(b',', start=3, endex=10)
1
>>> instance.count(b',', start=8, endex=10)
0
```

decode(*encoding='utf-8'*, *errors='strict'*)

Decode the bytes using the codec registered for encoding.

encoding
The encoding with which to decode the bytes.

errors
The error handling scheme to use for the handling of decoding errors. The default is 'strict' meaning that decoding errors raise a UnicodeDecodeError. Other possible values are 'ignore' and 'replace' as well as any other name registered with codecs.register_error that can handle UnicodeDecodeErrors.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.decode()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

endswith(*token*)

Return True if B ends with the specified suffix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. suffix can also be a tuple of bytes to try.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.endswith()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

property f_contiguous: bool

Contiguous array, *Fortran* language style.

Type
bool

find(*token*, *start=None*, *endex=None*)

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.find()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

property format: str

A string containing the format (in struct module style)
for each element in the view.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.format()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

index(*token*, *start=None*, *endex=None*)

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the subsection is not found.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.index()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

isalnum()

Return True if all characters in B are alphanumeric and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isalnum()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

isalpha()

Return True if all characters in B are alphabetic and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isalpha()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isascii()`

Return True if B is empty or all characters in B are ASCII, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isascii()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isdecimal()`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

> **Warning:** This method documentation is just a stub, copied directly from `str.isdecimal()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isdigit()`

Return True if all characters in B are digits and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isdigit()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isidentifier()`

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

> **Warning:** This method documentation is just a stub, copied directly from `str.isidentifier()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`islower()`

Return True if all cased characters in B are lowercase and there is at least one cased character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.islower()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isnumeric()`

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

> **Warning:** This method documentation is just a stub, copied directly from `str.isnumeric()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isprintable()`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

> **Warning:** This method documentation is just a stub, copied directly from `str.isprintable()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isspace()`

Return True if all characters in B are whitespace and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isspace()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`istitle()`

Return True if B is a titlecased string and there is at least one character in B, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.istitle()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isupper()`

Return True if all cased characters in B are uppercase and there is at least one cased character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isupper()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property itemsize: int`

The size in bytes of each element of the memoryview.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.itemsize()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`ljust(`*width*, *fillchar=b' '*, *factory=<class 'bytes'>*`)`

Return a left-justified string of length width.

Padding is done using the specified fill character.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.ljust()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`lower()`

Return a copy of B with all ASCII characters converted to lowercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.lower()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`lstrip`(*chars=None*, *factory=<class 'bytes'>*)

Strip leading bytes contained in the argument.

If the argument is omitted or None, strip leading ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.lstrip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`static maketrans`(*chars_from*, *chars_to*)

Return a translation table useable for the bytes or bytearray translate method.

The returned table will be one where each byte in frm is mapped to the byte at the same position in to.

The bytes objects frm and to must be of the same length.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.maketrans()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property nbytes: int`

The amount of space in bytes that the array would use in
a contiguous representation.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.nbytes()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property ndim: int`

An integer indicating how many dimensions of a multi-dimensional
array the memory represents.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.ndim()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property obj: ByteString | memoryview | None`

The underlying object of the memoryview.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.obj()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`partition`(*sep*, *factory=<class 'bytes'>*)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original bytes object and two empty bytes objects.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.partition()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property readonly: bool`

A bool indicating whether the memory is read only.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.readonly()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`release()`

Release the underlying buffer exposed by the memoryview object.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.release()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`removeprefix(`*prefix*, *factory=<class 'bytes'>*`)`

Return a bytes object with the given prefix string removed if present.

If the bytes starts with the prefix string, return bytes[len(prefix):]. Otherwise, return a copy of the original bytes.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.removeprefix()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`removesuffix(`*suffix*, *factory=<class 'bytes'>*`)`

Return a bytes object with the given suffix string removed if present.

If the bytes ends with the suffix string and that suffix is not empty, return bytes[:-len(prefix)]. Otherwise, return a copy of the original bytes.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.removesuffix()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`replace(`*old*, *new*, *count=None*, *start=None*, *endex=None*`)`

Return a copy with all occurrences of substring old replaced by new.

count
: Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.replace()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

rfind(*token*, *start=None*, *endex=None*)

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rfind()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

rindex(*token*, *start=None*, *endex=None*)

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raise ValueError when the subsection is not found.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rindex()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

rjust(*width*, *fillchar=b' '*, *factory=<class 'bytes'>*)

Return a right-justified string of length width.

Padding is done using the specified fill character.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rjust()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

rpartition(*sep*, *factory=<class 'bytes'>*)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty bytes objects and the original bytes object.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rpartition()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

rstrip(*chars=None*, *factory=<class 'bytes'>*)

Strip trailing bytes contained in the argument.

If the argument is omitted or None, strip trailing ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rstrip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

property shape: Tuple[int]

A tuple of ndim integers giving the shape of the memory
as an N-dimensional array.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.shape()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`startswith`(*token*)

Return True if B starts with the specified prefix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. prefix can also be a tuple of bytes to try.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.startswith()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property strides: Tuple[int]`

A tuple of ndim integers giving the size in bytes to access
each element for each dimension of the array.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.strides()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`strip`(*chars=None*, *factory=<class 'bytes'>*)

Strip leading and trailing bytes contained in the argument.

If the argument is omitted or None, strip leading and trailing ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.strip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property suboffsets: Tuple`

A tuple of integers used internally for PIL-style arrays.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.suboffsets()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`swapcase`()

Return a copy of B with uppercase ASCII characters converted to lowercase ASCII and vice versa.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.swapcase()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`title`()

Return a titlecased version of B, i.e. ASCII words start with uppercase characters, all remaining cased characters have lowercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.title()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`tobytes()`

Return the data in the buffer as a byte string.

Order can be {'C', 'F', 'A'}. When order is 'C' or 'F', the data of the original array is converted to C or Fortran order. For contiguous views, 'A' returns an exact copy of the physical memory. In particular, in-memory Fortran order is preserved. For non-contiguous views, the data is converted to C first. order=None is the same as order='C'.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.tobytes()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`tolist()`

Return the data in the buffer as a list of elements.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.tolist()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`translate(`*table*`)`

Return a copy with each character mapped by the given translation table.

table
Translation table, which must be a bytes object of length 256.

All characters occurring in the optional argument delete are removed. The remaining characters are mapped through the given translation table.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.translate()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`upper()`

Return a copy of B with all ASCII characters converted to uppercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.upper()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`zfill(`*width*, *factory*=<*class 'bytes'*>`)`

Pad a numeric string with zeros on the left, to fill a field of the given width.

The original string is never truncated.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.zfill()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

cbytesparse.py.InplaceView

class cbytesparse.py.**InplaceView**(*wrapped*)

Provides inplace methods to a byte buffer.

Standard Python *byte-like* objects (e.g. `memoryview` and `bytearray`) only provide a very small set of methods for *inplace* editing of their underlying bytes buffer.

This wrapper class adds a low-level implementation of those methods to anything supporting the *buffer protocol* with a mutable buffer.

Note: Editing support is only limited to the existing buffer items, i.e. the wrapper and its underlying buffer cannot be resized via the methods provided by this class, just like with standard memory views.

Parameters

wrapped (*byte-like*) – The target object supporting the *buffer protocol*.

Examples

```
>>> from cbytesparse import InplaceView
>>> buffer = bytearray(b'Hello, World!')
>>> wrapped = InplaceView(buffer)
>>> wrapped.lower()
>>> buffer
bytearray(b'hello, world!')
>>> wrapped.replace(b'l', b'p')
>>> buffer
bytearray(b'heppo, worpd!')
>>> wrapped.find(b'w')
7
>>> wrapped.rfind(b'o')
8
>>> wrapped.count(b'o')
2
```

Methods

__init__	
capitalize	byte-like: First character capitalized, the rest lowercase.
center	Return a centered string of length width.
contains	Contains a substring.
count	Counts token occurrences.
decode	Decode the bytes using the codec registered for encoding.
endswith	Return True if B ends with the specified suffix, False otherwise.

continues on next page

Table 9 – continued from previous page

find	Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end].
index	Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end].
isalnum	Return True if all characters in B are alphanumeric and there is at least one character in B, False otherwise.
isalpha	Return True if all characters in B are alphabetic and there is at least one character in B, False otherwise.
isascii	Return True if B is empty or all characters in B are ASCII, False otherwise.
isdecimal	Return True if the string is a decimal string, False otherwise.
isdigit	Return True if all characters in B are digits and there is at least one character in B, False otherwise.
isidentifier	Return True if the string is a valid Python identifier, False otherwise.
islower	Return True if all cased characters in B are lowercase and there is at least one cased character in B, False otherwise.
isnumeric	Return True if the string is a numeric string, False otherwise.
isprintable	Return True if the string is printable, False otherwise.
isspace	Return True if all characters in B are whitespace and there is at least one character in B, False otherwise.
istitle	Return True if B is a titlecased string and there is at least one character in B, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones.
isupper	Return True if all cased characters in B are uppercase and there is at least one cased character in B, False otherwise.
ljust	Return a left-justified string of length width.
lower	Return a copy of B with all ASCII characters converted to lowercase.
lstrip	Strip leading bytes contained in the argument.
maketrans	Return a translation table useable for the bytes or bytearray translate method.
partition	Partition the bytes into three parts using the given separator.
release	Release the underlying buffer exposed by the memoryview object.
removeprefix	Return a bytes object with the given prefix string removed if present.
removesuffix	Return a bytes object with the given suffix string removed if present.
replace	Return a copy with all occurrences of substring old replaced by new.
rfind	Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end].
rindex	Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end].

continues on next page

Table 9 – continued from previous page

rjust	Return a right-justified string of length width.
rpartition	Partition the bytes into three parts using the given separator.
rstrip	Strip trailing bytes contained in the argument.
startswith	Return True if B starts with the specified prefix, False otherwise.
strip	Strip leading and trailing bytes contained in the argument.
swapcase	Return a copy of B with uppercase ASCII characters converted to lowercase ASCII and vice versa.
title	Return a titlecased version of B, i.e. ASCII words start with uppercase characters, all remaining cased characters have lowercase.
tobytes	Return the data in the buffer as a byte string.
tolist	Return the data in the buffer as a list of elements.
toreadonly	Return a readonly version of the memoryview.
translate	Return a copy with each character mapped by the given translation table.
upper	Return a copy of B with all ASCII characters converted to uppercase.
zfill	Pad a numeric string with zeros on the left, to fill a field of the given width.

Attributes

c_contiguous	Contiguous array, *C* language style.
contiguous	Contiguous array.
f_contiguous	Contiguous array, *Fortran* language style.
format	A string containing the format (in struct module style)
itemsize	The size in bytes of each element of the memoryview.
nbytes	The amount of space in bytes that the array would use in
ndim	An integer indicating how many dimensions of a multi-dimensional
obj	The underlying object of the memoryview.
readonly	A bool indicating whether the memory is read only.
shape	A tuple of ndim integers giving the shape of the memory
strides	A tuple of ndim integers giving the size in bytes to access
suboffsets	A tuple of integers used internally for PIL-style arrays.

__bool__()

Has any items.

Returns

bool – Has any items.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'')
>>> bool(instance)
False
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> bool(instance)
True
```

__bytes__()

Creates a bytes clone.

Returns
bytes – Cloned data.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(bytearray(b''))
>>> bytes(instance)
b''
```

```
>>> instance = BytesMethods(bytearray(b'Hello, World!'))
>>> bytes(instance)
b'Hello, World!'
```

classmethod __class_getitem__(*params*)

Parameterizes a generic class.

At least, parameterizing a generic class is the *main* thing this method does. For example, for some generic class *Foo*, this is called when we do *Foo[int]* - there, with *cls=Foo* and *params=int*.

However, note that this method is also called when defining generic classes in the first place with *class Foo(Generic[T]): ...*.

__contains__(*token*)

Checks if some items are contained.

Parameters
token (*byte-like*) – Token to find.

Returns
bool – Token is contained.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> b'World' in instance
True
>>> b'$' in instance
False
>>> ord('o') in instance
True
>>> ord('$') in instance
False
```

__eq__(*other*)

Equality comparison.

Parameters
other (*byte-like*) – Data to compare with *self*.

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

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> data = bytearray(b'Hello, World!')
>>> instance = BytesMethods(data)
>>> instance == data
True
>>> instance == memoryview(data)
True
>>> instance == b'Something else'
False
```

__ge__(*other*)

Return self>=value.

Warning: This method documentation is just a stub, copied directly from `bytes.__ge__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

__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.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance[7]  # -> ord('W') = 87
121
>>> bytes(instance[:3])
b'Hel'
>>> bytes(instance[3:10])
b'lo, Wor'
>>> bytes(instance[-1:])
b'!'
>>> bytes(instance[2:10:3])
b'l,o'
>>> bytes(instance[3:10:2])
b'l,Wr'
```

__gt__(*other*)

Return self>value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__gt__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

__hash__ = None

__init__(*wrapped*)

classmethod __init_subclass__(**args*, ***kwargs*)

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__iter__()

Iterates over values.

Yields

int – Value as byte integer.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> list(instance)
[72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]
```

__le__(*other*)

Return self<=value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__le__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`__len__()`

Actual length.

Computes the actual length of the wrapped data object.

Returns

int – Data length.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'')
>>> len(instance)
0
```

```
>>> instance = BytesMethods(bytes(7))
>>> len(instance)
7
```

```
>>> instance = BytesMethods(memoryview(b'Hello, World!'))
>>> len(instance)
13
```

`__lt__(other)`

Return self<value.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.__lt__()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`__ne__(other)`

Inquality comparison.

Parameters

other (*byte-like*) – Data to compare with *self*.

Returns

bool – *self* is not equal to *other*.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> data = bytearray(b'Hello, World!')
>>> instance = BytesMethods(data)
>>> instance != data
False
>>> instance != memoryview(data)
False
>>> instance != b'Something else'
True
```

__reversed__()

Iterates over values, reversed order.

Yields
int – Value as byte integer.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> list(instance)
[72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100, 33]
>>> list(reversed(instance))
[33, 100, 108, 114, 111, 87, 32, 44, 111, 108, 108, 101, 72]
```

__sizeof__()

int: Allocated byte size.

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)

property c_contiguous: bool

Contiguous array, *C* language style.

Type
bool

capitalize()

byte-like: First character capitalized, the rest lowercase.

center(*width*, *fillchar=b' '*, *factory=<class 'bytes'>*)

Return a centered string of length width.

Padding is done using the specified fill character.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.center()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

contains(*token*, *start=None*, *endex=None*)

Contains a substring.

Parameters

- **token** (*byte-like*) – Token to search.
- **start** (*int*) – Inclusive start of the searched range, `None` to ignore.
- **endex** (*int*) – Exclusive end of the searched range, `None` to ignore.

Returns

bool – Token contained within the wrapped object.

See also:

__contains__()

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance.contains(b'World')
True
>>> instance.contains(b'$')
False
>>> instance.contains(ord('o'))
True
>>> instance.contains(ord('$'))
False
>>> instance.contains(b'Hello', endex=10)
True
>>> instance.contains(b'Hello', endex=3)
False
>>> instance.contains(b'World', start=3)
True
>>> instance.contains(b'World', start=10)
False
>>> instance.contains(b',', start=3, endex=10)
True
>>> instance.contains(b',', start=8, endex=10)
False
```

property contiguous: bool

Contiguous array.

Type

bool

count(*token*, *start=None*, *endex=None*)

Counts token occurrences.

Parameters

- **token** (*byte-like*) – Token to count.
- **start** (*int*) – Inclusive start of the searched range, None to ignore.
- **endex** (*int*) – Exclusive end of the searched range, None to ignore.

Returns

int – The number of items equal to *token*.

Examples

```
>>> from cbytesparse import BytesMethods
```

```
>>> instance = BytesMethods(b'Hello, World!')
>>> instance.count(b'o')
2
>>> instance.count(b'l')
3
>>> instance.count(b'll')
3
>>> instance.count(b'World')
1
>>> instance.count(b'$')
0
>>> instance.count(ord('o'))
1
>>> instance.count(ord('$'))
0
>>> instance.count(b'Hello', endex=10)
1
>>> instance.count(b'Hello', endex=3)
0
>>> instance.count(b'World', start=3)
1
>>> instance.count(b'World', start=10)
0
>>> instance.count(b',', start=3, endex=10)
1
>>> instance.count(b',', start=8, endex=10)
0
```

decode(*encoding='utf-8'*, *errors='strict'*)

Decode the bytes using the codec registered for encoding.

encoding

The encoding with which to decode the bytes.

errors

The error handling scheme to use for the handling of decoding errors. The default is 'strict' meaning that decoding errors raise a UnicodeDecodeError. Other possible values are 'ignore' and 'replace' as well as any other name registered with codecs.register_error that can handle UnicodeDecodeErrors.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.decode()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`endswith`(*token*)

Return True if B ends with the specified suffix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. suffix can also be a tuple of bytes to try.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.endswith()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property f_contiguous: bool`

Contiguous array, *Fortran* language style.

Type
bool

`find`(*token*, *start=None*, *endex=None*)

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.find()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property format: str`

A string containing the format (in struct module style)
for each element in the view.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.format()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`index`(*token*, *start=None*, *endex=None*)

Return the lowest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the subsection is not found.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.index()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isalnum`()

Return True if all characters in B are alphanumeric and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isalnum()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isalpha()`

Return True if all characters in B are alphabetic and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isalpha()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isascii()`

Return True if B is empty or all characters in B are ASCII, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isascii()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isdecimal()`

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

> **Warning:** This method documentation is just a stub, copied directly from `str.isdecimal()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isdigit()`

Return True if all characters in B are digits and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isdigit()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isidentifier()`

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

> **Warning:** This method documentation is just a stub, copied directly from `str.isidentifier()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`islower()`

Return True if all cased characters in B are lowercase and there is at least one cased character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.islower()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isnumeric()`

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

> **Warning:** This method documentation is just a stub, copied directly from `str.isnumeric()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isprintable()`

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

> **Warning:** This method documentation is just a stub, copied directly from `str.isprintable()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isspace()`

Return True if all characters in B are whitespace and there is at least one character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isspace()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`istitle()`

Return True if B is a titlecased string and there is at least one character in B, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.istitle()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`isupper()`

Return True if all cased characters in B are uppercase and there is at least one cased character in B, False otherwise.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.isupper()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property itemsize: int`

The size in bytes of each element of the memoryview.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.itemsize()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`ljust(`*width*, *fillchar=b' '*, *factory=<class 'bytes'>*`)`

Return a left-justified string of length width.

Padding is done using the specified fill character.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.ljust()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`lower()`

Return a copy of B with all ASCII characters converted to lowercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.lower()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`lstrip(`*chars=None*, *factory=<class 'bytes'>*`)`

Strip leading bytes contained in the argument.

If the argument is omitted or None, strip leading ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.lstrip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`static maketrans(`*chars_from*, *chars_to*`)`

Return a translation table useable for the bytes or bytearray translate method.

The returned table will be one where each byte in frm is mapped to the byte at the same position in to.

The bytes objects frm and to must be of the same length.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.maketrans()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property nbytes: int`

The amount of space in bytes that the array would use in
a contiguous representation.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.nbytes()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property ndim: int`

An integer indicating how many dimensions of a multi-dimensional
array the memory represents.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.ndim()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property obj: ByteString | memoryview | None`

The underlying object of the memoryview.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.obj()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

partition(*sep*, *factory*=<*class 'bytes'*>)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original bytes object and two empty bytes objects.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.partition()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

property readonly: bool

A bool indicating whether the memory is read only.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.readonly()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

release()

Release the underlying buffer exposed by the memoryview object.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.release()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

removeprefix(*prefix*, *factory*=<*class 'bytes'*>)

Return a bytes object with the given prefix string removed if present.

If the bytes starts with the prefix string, return bytes[len(prefix):]. Otherwise, return a copy of the original bytes.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.removeprefix()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

removesuffix(*suffix*, *factory*=<*class 'bytes'*>)

Return a bytes object with the given suffix string removed if present.

If the bytes ends with the suffix string and that suffix is not empty, return bytes[:-len(prefix)]. Otherwise, return a copy of the original bytes.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.removesuffix()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

replace(*old*, *new*, *count=None*, *start=None*, *endex=None*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.replace()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

rfind(*token*, *start=None*, *endex=None*)

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rfind()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

rindex(*token*, *start=None*, *endex=None*)

Return the highest index in B where subsection sub is found, such that sub is contained within B[start,end]. Optional arguments start and end are interpreted as in slice notation.

Raise ValueError when the subsection is not found.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rindex()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

rjust(*width*, *fillchar=b' '*, *factory=<class 'bytes'>*)

Return a right-justified string of length width.

Padding is done using the specified fill character.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rjust()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

rpartition(*sep*, *factory=<class 'bytes'>*)

Partition the bytes into three parts using the given separator.

This will search for the separator sep in the bytes, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty bytes objects and the original bytes object.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rpartition()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`rstrip`(*chars=None*, *factory=<class 'bytes'>*)

Strip trailing bytes contained in the argument.

If the argument is omitted or None, strip trailing ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.rstrip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property shape: Tuple[int]`

A tuple of ndim integers giving the shape of the memory
as an N-dimensional array.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.shape()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`startswith`(*token*)

Return True if B starts with the specified prefix, False otherwise. With optional start, test B beginning at that position. With optional end, stop comparing B at that position. prefix can also be a tuple of bytes to try.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.startswith()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property strides: Tuple[int]`

A tuple of ndim integers giving the size in bytes to access
each element for each dimension of the array.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.strides()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`strip`(*chars=None*, *factory=<class 'bytes'>*)

Strip leading and trailing bytes contained in the argument.

If the argument is omitted or None, strip leading and trailing ASCII whitespace.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.strip()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`property suboffsets: Tuple`

A tuple of integers used internally for PIL-style arrays.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.suboffsets()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`swapcase()`

Return a copy of B with uppercase ASCII characters converted to lowercase ASCII and vice versa.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.swapcase()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`title()`

Return a titlecased version of B, i.e. ASCII words start with uppercase characters, all remaining cased characters have lowercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.title()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`tobytes()`

Return the data in the buffer as a byte string.

Order can be {'C', 'F', 'A'}. When order is 'C' or 'F', the data of the original array is converted to C or Fortran order. For contiguous views, 'A' returns an exact copy of the physical memory. In particular, in-memory Fortran order is preserved. For non-contiguous views, the data is converted to C first. order=None is the same as order='C'.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.tobytes()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`tolist()`

Return the data in the buffer as a list of elements.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.tolist()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`toreadonly()`

Return a readonly version of the memoryview.

> **Warning:** This method documentation is just a stub, copied directly from `memoryview.toreadonly()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

`translate(`*table*`)`

Return a copy with each character mapped by the given translation table.

table
Translation table, which must be a bytes object of length 256.

All characters occurring in the optional argument delete are removed. The remaining characters are mapped through the given translation table.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.translate()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

upper()

Return a copy of B with all ASCII characters converted to uppercase.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.upper()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

zfill(*width*, *factory*=<*class 'bytes'*>)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The original string is never truncated.

> **Warning:** This method documentation is just a stub, copied directly from `bytes.zfill()`. It may be incomplete, or with different arguments. The behavior should be very similar though.

cbytesparse.py.Memory

class cbytesparse.py.**Memory**(*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`

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!']]
```

Methods

__init__	
append	Appends a single item.
append_backup	Backups an *append()* operation.
append_restore	Restores an *append()* operation.
block_span	Span of block data.
blocks	Iterates over blocks.
bound	Bounds addresses.
clear	Clears an address range.
clear_backup	Backups a *clear()* operation.
clear_restore	Restores a *clear()* operation.
collapse_blocks	Collapses a generic sequence of blocks.
content_blocks	Iterates over blocks.
content_items	Iterates over content address and value pairs.
content_keys	Iterates over content addresses.
content_values	Iterates over content values.
copy	Creates a deep copy.
count	Counts items.
crop	Keeps data within an address range.
crop_backup	Backups a *crop()* operation.
crop_restore	Restores a *crop()* operation.
cut	Cuts a slice of memory.
delete	Deletes an address range.
delete_backup	Backups a *delete()* operation.
delete_restore	Restores a *delete()* operation.
equal_span	Span of homogeneous data.
extend	Concatenates items.
extend_backup	Backups an *extend()* operation.
extend_restore	Restores an *extend()* operation.
extract	Selects items from a range.
fill	Overwrites a range with a pattern.
fill_backup	Backups a *fill()* operation.
fill_restore	Restores a *fill()* operation.
find	Index of an item.
flood	Fills emptiness between non-touching blocks.
flood_backup	Backups a *flood()* operation.
flood_restore	Restores a *flood()* operation.

continues on next page

Table 10 – continued from previous page

from_blocks	Creates a virtual memory from blocks.
from_bytes	Creates a virtual memory from a byte-like chunk.
from_items	Creates a virtual memory from a iterable address/byte mapping.
from_memory	Creates a virtual memory from another one.
from_values	Creates a virtual memory from a byte-like sequence.
fromhex	Creates a virtual memory from an hexadecimal string.
gaps	Iterates over block gaps.
get	Gets the item at an address.
hex	Converts into an hexadecimal string.
index	Index of an item.
insert	Inserts data.
insert_backup	Backups an *insert()* operation.
insert_restore	Restores an *insert()* operation.
intervals	Iterates over block intervals.
items	Iterates over address and value pairs.
keys	Iterates over addresses.
peek	Gets the item at an address.
poke	Sets the item at an address.
poke_backup	Backups a *poke()* operation.
poke_restore	Restores a *poke()* operation.
pop	Takes a value away.
pop_backup	Backups a *pop()* operation.
pop_restore	Restores a *pop()* operation.
popitem	Pops the last item.
popitem_backup	Backups a *popitem()* operation.
popitem_restore	Restores a *popitem()* operation.
read	Reads data.
readinto	Reads data into a pre-allocated buffer.
remove	Removes an item.
remove_backup	Backups a *remove()* operation.
remove_restore	Restores a *remove()* operation.
reserve	Inserts emptiness.
reserve_backup	Backups a *reserve()* operation.
reserve_restore	Restores a *reserve()* operation.
reverse	Reverses the memory in-place.
rfind	Index of an item, reversed search.
rindex	Index of an item, reversed search.
rvalues	Iterates over values, reversed order.
setdefault	Defaults a value.
setdefault_backup	Backups a *setdefault()* operation.
setdefault_restore	Restores a *setdefault()* operation.
shift	Shifts the items.
shift_backup	Backups a *shift()* operation.
shift_restore	Restores an *shift()* operation.
to_blocks	Exports into blocks.
to_bytes	Exports into bytes.
update	Updates data.
update_backup	Backups an *update()* operation.
update_restore	Restores an *update()* operation.
validate	Validates internal structure.
values	Iterates over values.

continues on next page

Table 10 – continued from previous page

view	Creates a view over a range.
write	Writes data.
write_backup	Backups a *write()* operation.
write_restore	Restores a *write()* operation.

Attributes

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

_Memory

alias of *Memory*

__add__(*value*)

Concatenates items.

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

See also:

`MutableMemory.__iadd__()`

Examples

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

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

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

__bool__()

Has any items.

Returns

bool – Has any items.

Examples

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

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

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

__bytes__()

Creates a bytes clone.

Returns

bytes – Cloned data.

Raises

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

Examples

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

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

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

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

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

__class_getitem__ = <bound method GenericAlias of <class 'cbytesparse.py.Memory'>>

__contains__(*item*)

Checks if some items are contained.

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

Returns
bool – Item is contained.

Examples

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

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

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

__copy__()

Creates a shallow copy.

Returns
ImmutableMemory – Shallow copy.

__deepcopy__()

Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

__delitem__(*key*)

Deletes data.

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

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

Examples

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

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

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

~~~

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

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

__eq__(*other*)

Equality comparison.

Parameters

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

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

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

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

Returns

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

Examples

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

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

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

__getitem__(*key*)

Gets data.

Parameters

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

Returns

items – Items from the requested range.

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

Examples

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

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

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

(continues on next page)

(continued from previous page)

```
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(items)`.

See also:

extend()

Examples

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

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

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

__imul__(*times*)

Concatenates a repeated copy.

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

See also:

extend()

Examples

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

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

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

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

__iter__()

Iterates over values.

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

Yields
int – Value as byte integer, or None.

Examples

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

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

__len__()

Actual length.

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

Returns
int – Memory length.

Examples

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

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

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

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

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

__mul__(*times*)

Concatenates a repeated copy.

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

See also:

`MutableMemory.__imul__()`

Examples

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

```
>>> memory1 = Memory.from_bytes(b'ABC')
>>> 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']]
```

__repr__()

Return repr(self).

__reversed__()

Iterates over values, reversed order.

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

Yields

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

Examples

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

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

__setitem__(*key*, *value*)

Sets data.

Parameters

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

Examples

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

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

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

~~~
~~~

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

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

__sizeof__()

Size of object in memory, in bytes.

__str__()

String representation.

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

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

Returns

str – String representation.

Examples

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

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

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

classmethod __subclasshook__(*C*)

Abstract classes can override this to customize issubclass().

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

__weakref__

list of weak references to the object (if defined)

_block_index_at(*address*)

Locates the block enclosing an address.

Returns the index of the block enclosing the given address.

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

Returns
int – Block index if found, None otherwise.

Examples

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

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

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

_block_index_endex(*address*)

Locates the last block before an address range.

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

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

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

Returns
int – First block index before *address*.

Examples

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

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

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

_block_index_start(*address*)

Locates the first block inside an address range.

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

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

Parameters

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

Returns

int – First block index since *address*.

Examples

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

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

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

_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()

append(*item*)

Appends a single item.

Parameters

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

See also:

append_backup() append_restore()

Examples

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

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

~~~

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

append_backup()

Backups an *append()* operation.

Returns
None – Nothing.

See also:

append() append_restore()

append_restore()

Restores an *append()* operation.

See also:

append() append_backup()

block_span(*address*)

Span of block data.

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

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

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

Parameters
address (*int*) – Reference address.

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

Examples

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

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

~~~

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

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

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

Iterates over blocks.

Iterates over data blocks within an address range.

Parameters

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

Yields

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

See also:

intervals() to_blocks()

Examples

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

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

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

bound(*start*, *endex*)

Bounds addresses.

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

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

In case of bounds limits, *bound_start* or *bound_endex* are used as bounds, when not `None`.

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

Returns

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

Examples

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

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

~~~

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

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

~~~

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

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

property bound_endex: int | None

Bounds exclusive end address.

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

Examples

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

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

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

Type
int

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

Bounds span addresses.

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

Notes

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

Examples

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

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

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

Type
tuple of int

property bound_start: int | None

Bounds start address.

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

Examples

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

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

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

Type
int

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

Clears an address range.

Parameters

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

See also:

clear_backup() clear_restore()

Examples

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

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

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

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

Backups a *clear()* operation.

Parameters

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

Returns
ImmutableMemory – Backup memory region.

See also:

clear() clear_restore()

clear_restore(*backup*)

Restores a *clear()* operation.

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

See also:

clear() clear_backup()

classmethod collapse_blocks(*blocks*)

Collapses a generic sequence of blocks.

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

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

Returns
list of blocks – Collapsed block list.

Examples

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

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

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

~~~
~~~

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

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

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

Iterates over blocks.

Iterates over data blocks within a block index range.

Parameters

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

Yields

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

See also:

content_parts

Examples

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

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

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

(continues on next page)

(continued from previous page)

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

property content_endex: int

Exclusive content end address.

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

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

Bounds considered only for an empty memory.

Examples

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

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

~~~

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

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

~~~

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

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

Type
int

property content_endin: int

Inclusive content end address.

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

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

Bounds considered only for an empty memory.

Examples

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

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

~~~

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

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

~~~

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

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

Type
int

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

Iterates over content address and value pairs.

Parameters

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

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

Yields
int – Content address and value pairs.

See also:

meth:*content_keys* meth:*content_values*

Examples

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

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

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

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

Iterates over content addresses.

Parameters

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

Yields
int – Content addresses.

See also:

meth:*content_items* meth:*content_values*

Examples

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

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

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

(continues on next page)

(continued from previous page)

```
[2, 5, 7, 8]
>>> list(memory.content_keys(3, 5))
[]
```

property content_parts: int

Number of blocks.

Returns

int – The number of blocks.

Examples

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

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

~~~

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

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

~~~

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

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

property content_size: int

Actual content size.

Returns

int – The sum of all block lengths.

Examples

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

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

~~~

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

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

~~~

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

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

property content_span: Tuple[int, int]

Memory content address span.

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

Examples

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

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

~~~
~~~

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

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

Type
tuple of int

property content_start: int

Inclusive content start address.

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

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

Bounds considered only for an empty memory.

Examples

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

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

~~~

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

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

~~~

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

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

Type
int

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

Iterates over content values.

Parameters

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

Yields
int – Content values.

See also:

meth:*content_items* meth:*content_keys*

Examples

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

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

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

property contiguous: bool

Contains contiguous data.

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

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

Examples

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

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

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

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

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

Type
bool

copy()

Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

Examples

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

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

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

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

(continues on next page)

(continued from previous page)

```
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
```

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

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

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

Counts items.

Parameters

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

Returns

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

Examples

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

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

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

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

Keeps data within an address range.

Parameters

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

See also:

crop_backup() crop_restore()

Examples

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

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

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

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

Backups a *crop()* operation.

Parameters

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

Returns

`ImmutableMemory` pair – Backup memory regions.

See also:

crop() crop_restore()

crop_restore(*backup_start*, *backup_endex*)

Restores a *crop()* operation.

Parameters

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

See also:

crop() crop_backup()

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

Cuts a slice of memory.

Parameters

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

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

Examples

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

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

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

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

Deletes an address range.

Parameters

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

See also:

delete_backup() delete_restore()

Examples

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

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

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

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

Backups a *delete()* operation.

Parameters

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

Returns
`ImmutableMemory` – Backup memory region.

See also:

delete() delete_restore()

delete_restore(*backup*)

Restores a *delete()* operation.

Parameters
backup (`ImmutableMemory`) – Backup memory region

See also:

delete() delete_backup()

property endex: int

Exclusive end address.

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

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

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

Examples

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

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

~~~

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

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

~~~

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

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

Type
int

property endin: int

Inclusive end address.

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

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

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

Examples

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

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

~~~

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

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

~~~

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

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

Type
int

equal_span(*address*)

Span of homogeneous data.

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

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

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

Parameters

address (*int*) – Reference address.

Returns

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

Examples

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

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

~~~

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

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

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

Concatenates items.

Appends *items* after *content_endex*. Equivalent to `self += items`.

Parameters

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

See also:

__iadd__() extend_backup() extend_restore()
~~~

Examples

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

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

~~~

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

~~~

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

extend_backup(*offset=0*)

Backups an *extend()* operation.

Parameters

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

Returns

int – Content exclusive end address.

See also:

extend() extend_restore()

extend_restore(*content_endex*)

Restores an *extend()* operation.

Parameters

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

See also:

extend() extend_backup()

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

Selects items from a range.

Parameters

- **start** (*int*) – Inclusive start of the extracted range. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end of the extracted range. If `None`, *endex* is considered.
- **pattern** (*items*) – Optional pattern of items to fill the emptiness.

- **step** (*int*) – Optional address stepping between bytes extracted from the range. It has the same meaning of Python's `slice.step`, but negative steps are ignored. Please note that a *step* greater than 1 could take much more time to process than the default unitary step.
- **bound** (*bool*) – The selected address range is applied to the resulting memory as its bounds range. This retains information about any initial and final emptiness of that range, which would be lost otherwise.

Returns

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

Examples

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

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

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

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

Overwrites a range with a pattern.

Parameters

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

See also:

fill_backup() fill_restore()

Examples

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

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

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

~~~

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

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

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

Backups a *fill()* operation.

Parameters

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

Returns

ImmutableMemory – Backup memory region.

See also:

fill() fill_restore()

fill_restore(*backup*)

Restores a *fill()* operation.

Parameters

backup (ImmutableMemory) – Backup memory region to restore.

See also:

fill() fill_backup()

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

Index of an item.

Parameters
~~~

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

Returns

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

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

Fills emptiness between non-touching blocks.

Parameters

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

See also:

flood_backup() flood_restore()

Examples

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

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

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

~~~

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

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

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

Backups a *flood()* operation.

Parameters

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

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

Returns
list of open intervals – Backup memory gaps.

See also:

flood() flood_restore()

flood_restore(*gaps*)

Restores a *flood()* operation.

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

See also:

flood() flood_backup()

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

Creates a virtual memory from blocks.

Parameters

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

Returns
ImmutableMemory – The resulting memory object.

Raises
ValueError – Some requirements are not satisfied.

See also:

to_blocks()

Examples

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

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

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

(continues on next page)

(continued from previous page)

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

~~~

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

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

Creates a virtual memory from a byte-like chunk.

Parameters

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

Returns

`ImmutableMemory` – The resulting memory object.

Raises

ValueError – Some requirements are not satisfied.

See also:

to_bytes()

Examples

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

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

~~~

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

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

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

Creates a virtual memory from a iterable address/byte mapping.

Parameters

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

Returns

`ImmutableMemory` – The resulting memory object.

Raises

ValueError – Some requirements are not satisfied.

See also:

to_bytes()

Examples

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

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

~~~

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

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

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

Creates a virtual memory from another one.

Parameters

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

Returns

ImmutableMemory – The resulting memory object.

Raises

ValueError – Some requirements are not satisfied.

Examples

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

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

~~~

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

~~~

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

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

Creates a virtual memory from a byte-like sequence.

Parameters

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

Returns

`ImmutableMemory` – The resulting memory object.

Raises

ValueError – Some requirements are not satisfied.

See also:

to_bytes()

Examples

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

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

~~~

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

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

classmethod fromhex(*string*)

Creates a virtual memory from an hexadecimal string.

Parameters

string (*str*) – Hexadecimal string.

Returns

`ImmutableMemory` – The resulting memory object.
~~~

Examples

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

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

~~~

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

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

Iterates over block gaps.

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

Parameters

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

Yields

pair of addresses – Block data interval boundaries.

See also:

intervals()

Examples

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

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

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

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

Gets the item at an address.
~~~

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

Examples

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

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

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

hex(**args*)

Converts into an hexadecimal string.

Parameters

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

Returns
str – Hexadecimal string representation.

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

Examples

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

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

~~~

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

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.

insert(*address*, *data*)

Inserts data.

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

Arguments::

address (int):
Address of the insertion point.

data (bytes):
Data to insert.

See also:

insert_backup() insert_restore()
~~~

Examples

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

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

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

insert_backup(*address*, *data*)

Backups an *insert()* operation.

Parameters

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

Returns

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

See also:

insert() insert_restore()

insert_restore(*address*, *backup*)

Restores an *insert()* operation.

Parameters

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

See also:

insert() insert_backup()

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

Iterates over block intervals.

Iterates over data boundaries within an address range.

Parameters

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

Yields

pair of addresses – Block data interval boundaries.

See also:

blocks() gaps()

Examples

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

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

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

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

Iterates over address and value pairs.

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

Parameters

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

Yields

int – Range address and value pairs.

Examples

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

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

~~~
~~~

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

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

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

Iterates over addresses.

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

Parameters

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

Yields

int – Range address.

Examples

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

```
>>> from itertools import islice
>>> memory = Memory()
>>> list(memory.keys())
[]
>>> list(memory.keys(endex=8))
[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]
```

(continues on next page)
~~~

(continued from previous page)

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

peek(*address*)

Gets the item at an address.

Returns

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

Examples

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

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

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

poke(*address*, *item*)

Sets the item at an address.

Parameters

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

See also:

poke_backup() poke_restore()

Examples

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

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

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

poke_backup(*address*)

Backups a *poke()* operation.

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

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

See also:

poke() poke_restore()

poke_restore(*address*, *item*)

Restores a *poke()* operation.

Parameters

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

See also:

poke() poke_backup()

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

Takes a value away.

Parameters

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

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

See also:

pop_backup() pop_restore()

Examples

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

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

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

pop_backup(*address=None*)

Backups a *pop()* operation.

Parameters

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

Returns

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

See also:

pop() pop_restore()

pop_restore(*address*, *item*)

Restores a *pop()* operation.

Parameters

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

See also:

pop() pop_backup()

popitem()

Pops the last item.

Returns

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

See also:

popitem_backup() popitem_restore()

Examples

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

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

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

popitem_backup()

Backups a *popitem()* operation.

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

See also:

popitem() popitem_restore()

popitem_restore(*address*, *item*)

Restores a *popitem()* operation.

Parameters

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

See also:

popitem() popitem_backup()

read(*address*, *size*)

Reads data.

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

Parameters

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

Returns
memoryview – A view over the addressed chunk.

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

Examples

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

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

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> bytes(memory.read(2, 3))
b'BCD'
>>> bytes(memory.read(9, 1))
b'y'
>>> memory.read(4, 3)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.read(0, 6)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

readinto(*address*, *buffer*)

Reads data into a pre-allocated buffer.

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

Parameters

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

Returns

int – Number of bytes read.

Raises

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

Examples

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

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

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> buffer = bytearray(3)
>>> memory.readinto(2, buffer)
```

(continues on next page)

(continued from previous page)

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

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

Removes an item.

Searches and deletes the first occurrence of an item.

Parameters

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

Raises

ValueError – Item not found.

See also:

remove_backup() remove_restore()

Examples

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

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

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.remove(b'BC')
>>> memory.to_blocks()
[[1, b'AD'], [4, b'$'], [6, b'xyz']]
>>> memory.remove(ord('$'))
>>> memory.to_blocks()
```

(continues on next page)

(continued from previous page)

```
[[1, b'AD'], [5, b'xyz']]
>>> memory.remove(b'?')
Traceback (most recent call last):
    ...
ValueError: subsection not found
```

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

Backups a *remove()* operation.

Parameters

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

Returns

Memory – Backup memory region.

See also:

remove() remove_restore()

remove_restore(*backup*)

Restores a *remove()* operation.

Parameters

backup (*Memory*) – Backup memory region.

See also:

remove() remove_backup()

reserve(*address*, *size*)

Inserts emptiness.

Reserves emptiness at the provided address.

Parameters

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

See also:

reserve_backup() reserve_restore()

Examples

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

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

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

~~~

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

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

reserve_backup(*address*, *size*)

Backups a *reserve()* operation.

Parameters

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

Returns

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

See also:

reserve() reserve_restore()

reserve_restore(*address*, *backup*)

Restores a *reserve()* operation.

Parameters

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

See also:

reserve() reserve_backup()

reverse()

Reverses the memory in-place.

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

Examples

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

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

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

~~~

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

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

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

Index of an item, reversed search.

Parameters

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

Returns

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

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.
~~~

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]
```

(continues on next page)
~~~

(continued from previous page)

```
>>> list(memory.rvalues(3, 8, b'0123'))
[121, 120, 50, 49, 67]
```

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

Defaults a value.

Parameters

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

Returns

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

See also:

setdefault_backup() setdefault_restore()

Examples

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

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

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

setdefault_backup(*address*)

Backups a *setdefault()* operation.

Parameters

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

Returns

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

See also:

setdefault() setdefault_restore()

setdefault_restore(*address*, *item*)

Restores a *setdefault()* operation.

Parameters

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

See also:

setdefault() setdefault_backup()

shift(*offset*)

Shifts the items.

Parameters

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

See also:

shift_backup() shift_restore()

Examples

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

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

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

~~~

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

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

shift_backup(*offset*)

Backups a *shift()* operation.
~~~

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

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

See also:

shift() shift_restore()

shift_restore(*offset*, *backup*)

Restores an *shift()* operation.

Parameters

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

See also:

shift() shift_backup()

property span: Tuple[int, int]

Memory address span.

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

Examples

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

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

~~~

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

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

Type
tuple of int

property start: int

Inclusive start address.

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

If *bound_start* not None, that is returned.
~~~

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

Examples

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

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

~~~

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

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

~~~

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

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

Type
int

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

Exports into blocks.

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

Parameters

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

Returns
list of blocks – Exported data blocks.

See also:

blocks() from_blocks()

Examples

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

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

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

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

Exports into bytes.

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

Parameters

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

Returns

bytes – Exported data bytes.

See also:

from_bytes() view()

Examples

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

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

~~~

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

```
>>> memory = Memory.from_bytes(b'ABCxyz', 2)
>>> memory.to_bytes()
b'ABCxyz'
>>> memory.to_bytes(start=4)
```

(continues on next page)
~~~

(continued from previous page)

```
b'Cxyz'
>>> memory.to_bytes(endex=6)
b'ABCx'
>>> memory.to_bytes(4, 6)
b'Cx'
```

update(*data*, *clear=False*, ***kwargs*)

Updates data.

Parameters

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

See also:

update_backup() update_restore()

Examples

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

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

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

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

Backups an *update()* operation.

Parameters

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

Returns
list of `ImmutableMemory` – Backup memory regions.

See also:

update() update_restore()

`update_restore`(*backups*)

Restores an *update()* operation.

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

See also:

update() update_backup()

`validate`()

Validates internal structure.

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

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

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

Iterates over values.

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

Parameters

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

Yields
int – Range values.

Examples

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

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

```
>>> from itertools import islice
>>> memory = Memory()
>>> list(memory.values(endex=8))
[None, None, None, None, None, None, None, None]
>>> list(memory.values(3, 8))
[None, None, None, None, None]
```

(continues on next page)

(continued from previous page)

```
>>> 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'
```

(continues on next page)
~~~

(continued from previous page)

```
>>> bytes(memory.view(9, 10))
b'y'
>>> memory.view()
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.view(0, 6)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

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

Writes data.

Parameters

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

See also:

write_backup() write_restore()

Examples

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

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

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

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

Backups a *write()* operation.

Parameters

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

Returns

list of `ImmutableMemory` – Backup memory regions.

See also:

write() write_restore()

write_restore(*backups*)

Restores a *write()* operation.

Parameters

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

See also:

write() write_backup()

cbytesparse.py.bytesparse

class cbytesparse.py.**bytesparse**(**args*, *start=None*, *endex=None*)

Wrapper for more *bytearray* compatibility.

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

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

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

See also:

ImmutableMemory MutableMemory

Parameters

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

Examples

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

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

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
[[0, b'Non-ASCII: ']]
```

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

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

Methods

__init__	
append	Appends a single item.
append_backup	Backups an *append()* operation.
append_restore	Restores an *append()* operation.
block_span	Span of block data.
blocks	Iterates over blocks.
bound	Bounds addresses.
clear	Clears an address range.
clear_backup	Backups a *clear()* operation.
clear_restore	Restores a *clear()* operation.
collapse_blocks	Collapses a generic sequence of blocks.
content_blocks	Iterates over blocks.
content_items	Iterates over content address and value pairs.
content_keys	Iterates over content addresses.
content_values	Iterates over content values.
copy	Creates a deep copy.
count	Counts items.
crop	Keeps data within an address range.
crop_backup	Backups a *crop()* operation.
crop_restore	Restores a *crop()* operation.
cut	Cuts a slice of memory.
delete	Deletes an address range.
delete_backup	Backups a *delete()* operation.
delete_restore	Restores a *delete()* operation.
equal_span	Span of homogeneous data.
extend	Concatenates items.
extend_backup	Backups an *extend()* operation.
extend_restore	Restores an *extend()* operation.
extract	Selects items from a range.
fill	Overwrites a range with a pattern.
fill_backup	Backups a *fill()* operation.
fill_restore	Restores a *fill()* operation.
find	Index of an item.

continues on next page

Table 11 – continued from previous page

flood	Fills emptiness between non-touching blocks.
flood_backup	Backups a *flood()* operation.
flood_restore	Restores a *flood()* operation.
from_blocks	Creates a virtual memory from blocks.
from_bytes	Creates a virtual memory from a byte-like chunk.
from_items	Creates a virtual memory from a iterable address/byte mapping.
from_memory	Creates a virtual memory from another one.
from_values	Creates a virtual memory from a byte-like sequence.
fromhex	Creates a virtual memory from an hexadecimal string.
gaps	Iterates over block gaps.
get	Gets the item at an address.
hex	Converts into an hexadecimal string.
index	Index of an item.
insert	Inserts data.
insert_backup	Backups an *insert()* operation.
insert_restore	Restores an *insert()* operation.
intervals	Iterates over block intervals.
items	Iterates over address and value pairs.
keys	Iterates over addresses.
peek	Gets the item at an address.
poke	Sets the item at an address.
poke_backup	Backups a *poke()* operation.
poke_restore	Restores a *poke()* operation.
pop	Takes a value away.
pop_backup	Backups a *pop()* operation.
pop_restore	Restores a *pop()* operation.
popitem	Pops the last item.
popitem_backup	Backups a *popitem()* operation.
popitem_restore	Restores a *popitem()* operation.
read	Reads data.
readinto	Reads data into a pre-allocated buffer.
remove	Removes an item.
remove_backup	Backups a *remove()* operation.
remove_restore	Restores a *remove()* operation.
reserve	Inserts emptiness.
reserve_backup	Backups a *reserve()* operation.
reserve_restore	Restores a *reserve()* operation.
reverse	Reverses the memory in-place.
rfind	Index of an item, reversed search.
rindex	Index of an item, reversed search.
rvalues	Iterates over values, reversed order.
setdefault	Defaults a value.
setdefault_backup	Backups a *setdefault()* operation.
setdefault_restore	Restores a *setdefault()* operation.
shift	Shifts the items.
shift_backup	Backups a *shift()* operation.
shift_restore	Restores an *shift()* operation.
to_blocks	Exports into blocks.
to_bytes	Exports into bytes.
update	Updates data.
update_backup	Backups an *update()* operation.

continues on next page

Table 11 – continued from previous page

update_restore	Restores an *update()* operation.
validate	Validates internal structure.
values	Iterates over values.
view	Creates a view over a range.
write	Writes data.
write_backup	Backups a *write()* operation.
write_restore	Restores a *write()* operation.

Attributes

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

_Memory

alias of *bytesparse*

__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
```

(continues on next page)

(continued from previous page)

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

__bool__()

Has any items.

Returns
bool – Has any items.

Examples

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

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

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

__bytes__()

Creates a bytes clone.

Returns
bytes – Cloned data.

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

Examples

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

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

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

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

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

__class_getitem__ = <bound method GenericAlias of <class 'cbytesparse.py.bytesparse'>>

__contains__(*item*)

Checks if some items are contained.

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

Returns
bool – Item is contained.

Examples

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

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

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

__copy__()

Creates a shallow copy.

Returns
ImmutableMemory – Shallow copy.

__deepcopy__()

Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

__delitem__(*key*)

Deletes data.

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

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

Examples

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

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

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

~~~

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

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

__eq__(*other*)

Equality comparison.

Parameters

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

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

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

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

Returns

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

Examples

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

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

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

__getitem__(*key*)

Gets data.

Parameters

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

Returns

items – Items from the requested range.

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

Examples

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

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

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

(continues on next page)

(continued from previous page)

```
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(items)`.

See also:

extend()

Examples

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

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

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

__imul__(*times*)

Concatenates a repeated copy.

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

See also:

extend()

Examples

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

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

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

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

__iter__()

Iterates over values.

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

Yields
int – Value as byte integer, or None.

Examples

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

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

__len__()

Actual length.

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

Returns
int – Memory length.

Examples

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

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

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

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

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

__mul__(*times*)

Concatenates a repeated copy.

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

See also:

`MutableMemory.__imul__()`

Examples

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

```
>>> memory1 = Memory.from_bytes(b'ABC')
>>> 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']]
```

__repr__()

Return repr(self).

__reversed__()

Iterates over values, reversed order.

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

Yields

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

Examples

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

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

__setitem__(*key*, *value*)

Sets data.

Parameters

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

Examples

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

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

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

~~~
~~~

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

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

__sizeof__()

Size of object in memory, in bytes.

__str__()

String representation.

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

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

Returns

str – String representation.

Examples

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

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

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

classmethod __subclasshook__(*C*)

Abstract classes can override this to customize issubclass().

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

__weakref__

list of weak references to the object (if defined)

_block_index_at(*address*)

Locates the block enclosing an address.

Returns the index of the block enclosing the given address.

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

Returns
int – Block index if found, None otherwise.

Examples

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

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

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

_block_index_endex(*address*)

Locates the last block before an address range.

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

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

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

Returns
int – First block index before *address*.

Examples

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

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

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

_block_index_start(*address*)

Locates the first block inside an address range.

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

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

Parameters

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

Returns

int – First block index since *address*.

Examples

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

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

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

_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.

append(*item*)

Appends a single item.

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

See also:

append_backup() append_restore()

Examples

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

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

~~~

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

append_backup()

Backups an *append()* operation.

Returns
None – Nothing.

See also:

append() append_restore()

append_restore()

Restores an *append()* operation.

See also:

append() append_backup()

block_span(*address*)

Span of block data.

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

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

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

Parameters
address (*int*) – Reference address.

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

Examples

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

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

~~~

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

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

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

Iterates over blocks.

Iterates over data blocks within an address range.

Parameters

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

Yields

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

See also:

intervals() to_blocks()

Examples

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

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

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

bound(*start*, *endex*)

Bounds addresses.

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

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

In case of bounds limits, *bound_start* or *bound_endex* are used as bounds, when not `None`.

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

Returns

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

Examples

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

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

~~~

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

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

~~~

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

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

property bound_endex: int | None

Bounds exclusive end address.

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

Examples

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

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

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

Type
int

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

Bounds span addresses.

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

Notes

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

Examples

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

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_span = (7, 13)
>>> memory.to_blocks()
[[7, b'llo, W']]
>>> memory.bound_span = None
>>> memory.bound_span
(None, None)
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=7, endex=13)
>>> memory.to_blocks()
[[7, b'llo, W']]
```

Type
tuple of int

property bound_start: int | None

Bounds start address.

Any data before this address is automatically discarded. Disabled if `None`.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.bound_start = 10
>>> memory.to_blocks()
[[10, b', World!']]
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=10)
>>> memory.to_blocks()
[[10, b', World!']]
```

Type
int

clear(*start=None*, *endex=None*)

Clears an address range.

Parameters

- **start** (*int*) – Inclusive start address for clearing. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address for clearing. If `None`, *endex* is considered.

See also:

clear_backup() *clear_restore()*

Examples

```
>>> from bytesparse import Memory
```

4	5	6	7	8	9	10	11	12
	[A	B	C]		[x	y	z]	
	[A]					[y	z]	

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.clear(6, 10)
>>> memory.to_blocks()
[[5, b'A'], [10, b'yz']]
```

clear_backup(*start=None*, *endex=None*)

Backups a *clear()* operation.

Parameters

- **start** (*int*) – Inclusive start address for clearing. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address for clearing. If `None`, *endex* is considered.

Returns

`ImmutableMemory` – Backup memory region.

See also:

clear() *clear_restore()*

clear_restore(*backup*)

Restores a *clear()* operation.

Parameters

backup (`ImmutableMemory`) – Backup memory region to restore.

See also:

clear() *clear_backup()*

classmethod collapse_blocks(*blocks*)

Collapses a generic sequence of blocks.

Given a generic sequence of blocks, writes them in the same order, generating a new sequence of non-contiguous blocks, sorted by address.

Parameters

blocks (*sequence of blocks*) – Sequence of blocks to collapse.

Returns

list of blocks – Collapsed block list.

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9
[0	1	2	3	4	5	6	7	8	9]
[A	B	C	D]						
			[E	F]					
[$]									
						[x	y	z]	
[$	B	C	E	F	5	x	y	z	9]

```
>>> blocks = [
...     [0, b'0123456789'],
...     [0, b'ABCD'],
...     [3, b'EF'],
...     [0, b'$'],
...     [6, b'xyz'],
... ]
>>> Memory.collapse_blocks(blocks)
[[0, b'$BCEF5xyz9']]
```

~~~

0	1	2	3	4	5	6	7	8	9
[0	1	2]							
				[A	B]				
						[x	y	z]	
	[$]								
[0	$	2]		[A	B	x	y	z]	

```
>>> blocks = [
...     [0, b'012'],
...     [4, b'AB'],
...     [6, b'xyz'],
...     [1, b'$'],
... ]
>>> Memory.collapse_blocks(blocks)
[[0, b'0$2'], [4, b'ABxyz']]
```

content_blocks(*block_index_start=None*, *block_index_endex=None*, *block_index_step=None*)

Iterates over blocks.

Iterates over data blocks within a block index range.

Parameters

- **block_index_start** (*int*) – Inclusive block start index. A negative index is referred to *content_parts*. If None, `0` is considered.
- **block_index_endex** (*int*) – Exclusive block end index. A negative index is referred to *content_parts*. If None, *content_parts* is considered.
~~~

- **block_index_step** (*int*) – Block index step, which can be negative. If None, 1 is considered.

Yields

(start, memoryview) – Start and data view of each block/slice.

See also:

content_parts

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> [[s, bytes(d)] for s, d in memory.content_blocks()]
[[1, b'AB'], [5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(1, 2)]
[[5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(3, 5)]
[]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_start=-2)]
[[5, b'x'], [7, b'123']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_endex=-1)]
[[1, b'AB'], [5, b'x']]
>>> [[s, bytes(d)] for s, d in memory.content_blocks(block_index_step=2)]
[[1, b'AB'], [7, b'123']]
```

property content_endex: int

Exclusive content end address.

This property holds the exclusive end address of the memory content. By default, it is the current maximum exclusive end address of the last stored block.

If the memory has no data and no bounds, *start* is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().content_endex
0
>>> Memory(endex=8).content_endex
0
>>> Memory(start=1, endex=8).content_endex
1
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_endex
8
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_endex
4
```

Type
int

property content_endin: int

Inclusive content end address.

This property holds the inclusive end address of the memory content. By default, it is the current maximum inclusive end address of the last stored block.

If the memory has no data and no bounds, *start* minus one is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().content_endin
-1
>>> Memory(endex=8).content_endin
-1
>>> Memory(start=1, endex=8).content_endin
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	
~~~

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_endin
7
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_endin
3
```

Type
int

`content_items`(*start=None*, *endex=None*)

Iterates over content address and value pairs.

Parameters

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields
int – Content address and value pairs.

See also:

meth:*content_keys* meth:*content_values*

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> dict(memory.content_items())
{1: 65, 2: 66, 5: 120, 7: 49, 8: 50, 9: 51}
>>> dict(memory.content_items(2, 9))
{2: 66, 5: 120, 7: 49, 8: 50}
>>> dict(memory.content_items(3, 5))
{}
```

`content_keys`(*start=None*, *endex=None*)

Iterates over content addresses.

Parameters
~~~

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields
int – Content addresses.

See also:

meth:*content_items* meth:*content_values*

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> list(memory.content_keys())
[1, 2, 5, 7, 8, 9]
>>> list(memory.content_keys(2, 9))
[2, 5, 7, 8]
>>> list(memory.content_keys(3, 5))
[]
```

property content_parts: int

Number of blocks.

Returns
int – The number of blocks.

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().content_parts
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_parts
2
```
~~~

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_parts
1
```

property content_size: int

Actual content size.

Returns

int – The sum of all block lengths.

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().content_size
0
>>> Memory(start=1, endex=8).content_size
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_size
6
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.content_size
3
```

property content_span: Tuple[int, int]

Memory content address span.

A `tuple` holding both *`content_start`* and *`content_endex`*.
~~~

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().content_span
(0, 0)
>>> Memory(start=1).content_span
(1, 1)
>>> Memory(endex=8).content_span
(0, 0)
>>> Memory(start=1, endex=8).content_span
(1, 1)
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_span
(1, 8)
```

Type
tuple of int

property content_start: int

Inclusive content start address.

This property holds the inclusive start address of the memory content. By default, it is the current minimum inclusive start address of the first stored block.

If the memory has no data and no bounds, 0 is returned.

Bounds considered only for an empty memory.

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().content_start
0
>>> Memory(start=1).content_start
1
>>> Memory(start=1, endex=8).content_start
1
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.content_start
1
```

~~~

0	1	2	3	4	5	6	7	8
	[[[				[x	y	z]	

```
>>> memory = Memory.from_blocks([[5, b'xyz']], start=1)
>>> memory.content_start
5
```

Type
int

`content_values`(*start=None*, *endex=None*)

Iterates over content values.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.

Yields
int – Content values.

See also:

meth:*content_items* meth:*content_keys*

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> list(memory.content_values())
[65, 66, 120, 49, 50, 51]
>>> list(memory.content_values(2, 9))
[66, 120, 49, 50]
>>> list(memory.content_values(3, 5))
[]
```

`property contiguous: bool`

Contains contiguous data.

The memory is considered to have contiguous data if there is no empty space between blocks.
~~~

If bounds are defined, there must be no empty space also towards them.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory.contiguous
True
```

```
>>> memory = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory.contiguous
False
```

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.contiguous
False
```

Type
bool

copy()

Creates a deep copy.

Returns
ImmutableMemory – Deep copy.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory()
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(None, None)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory(start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5)
>>> memory2 = memory1.copy()
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
```

```
>>> memory1 = Memory.from_bytes(b'Hello, World!', offset=5, start=1, endex=20)
>>> memory2 = memory1.copy()
>>> memory2.bound_span
(1, 20)
>>> memory2.to_blocks()
[[5, b'Hello, World!']]
>>> memory2.bound_span = (2, 19)
>>> memory1 == memory2
True
```

```
>>> memory1 = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory2 = memory1.copy()
[[5, b'ABC'], [9, b'xyz']]
>>> memory1 == memory2
True
```

count(*item*, *start=None*, *endex=None*)

Counts items.

Parameters

- **item** (*items*) – Reference value to count.
- **start** (*int*) – Inclusive start of the searched range. If None, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If None, *endex* is considered.

Returns

int – The number of items equal to *value*.

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C]		[B	a	t]		[t	a	b]

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'Bat'], [9, b'tab']])
>>> memory.count(b'a')
2
```

crop(*start=None*, *endex=None*)

Keeps data within an address range.

Parameters

- **start** (*int*) – Inclusive start address for cropping. If None, *start* is considered.

- **endex** (*int*) – Exclusive end address for cropping. If `None`, *endex* is considered.

See also:

crop_backup() crop_restore()

Examples

```
>>> from bytesparse import Memory
```

4	5	6	7	8	9	10	11	12
	[A	B	C]		[x	y	z]	
		[B	C]		[x]			

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.crop(6, 10)
>>> memory.to_blocks()
[[6, b'BC'], [9, b'x']]
```

crop_backup(*start=None*, *endex=None*)

Backups a *crop()* operation.

Parameters

- **start** (*int*) – Inclusive start address for cropping. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address for cropping. If `None`, *endex* is considered.

Returns

`ImmutableMemory` pair – Backup memory regions.

See also:

crop() crop_restore()

crop_restore(*backup_start*, *backup_endex*)

Restores a *crop()* operation.

Parameters

- **backup_start** (`ImmutableMemory`) – Backup memory region to restore at the beginning.
- **backup_endex** (`ImmutableMemory`) – Backup memory region to restore at the end.

See also:

crop() crop_backup()

cut(*start=None*, *endex=None*, *bound=True*)

Cuts a slice of memory.

Parameters

- **start** (*int*) – Inclusive start address for cutting. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address for cutting. If `None`, *endex* is considered.
- **bound** (*bool*) – The selected address range is applied to the resulting memory as its bounds range. This retains information about any initial and final emptiness of that range, which would be lost otherwise.

Returns
Memory – A copy of the memory from the selected range.

Examples

```
>>> from bytesparse import Memory
```

4	5	6	7	8	9	10	11	12
	[A	B	C]		[x	y	z]	
		[B	C]		[x]			
	[A]					[y	z]	

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> taken = memory.cut(6, 10)
>>> taken.to_blocks()
[[6, b'BC'], [9, b'x']]
>>> memory.to_blocks()
[[5, b'A'], [10, b'yz']]
```

delete(*start=None*, *endex=None*)

Deletes an address range.

Parameters

- **start** (*int*) – Inclusive start address for deletion. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address for deletion. If `None`, *endex* is considered.

See also:

delete_backup() delete_restore()

Examples

```
>>> from bytesparse import Memory
```

4	5	6	7	8	9	10	11	12	13
	[A	B	C]			[x	y	z]	
	[A	y	z]						

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.delete(6, 10)
>>> memory.to_blocks()
[[5, b'Ayz']]
```

delete_backup(*start=None*, *endex=None*)

Backups a *delete()* operation.

Parameters

- **start** (*int*) – Inclusive start address for deletion. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address for deletion. If `None`, *endex* is considered.

Returns
`ImmutableMemory` – Backup memory region.

See also:

delete() delete_restore()

delete_restore(*backup*)

Restores a *delete()* operation.

Parameters
backup (`ImmutableMemory`) – Backup memory region

See also:

delete() delete_backup()

property endex: int

Exclusive end address.

This property holds the exclusive end address of the virtual space. By default, it is the current maximmum exclusive end address of the last stored block.

If *bound_endex* not `None`, that is returned.

If the memory has no data and no bounds, *start* is returned.

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().endex
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.endex
8
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.endex
8
```

Type
int

property endin: int

Inclusive end address.

This property holds the inclusive end address of the virtual space. By default, it is the current maximmum inclusive end address of the last stored block.

If *bound_endex* not None, that minus one is returned.

If the memory has no data and no bounds, *start* is returned.

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().endin
-1
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.endin
7
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]					)))

```
>>> memory = Memory.from_blocks([[1, b'ABC']], endex=8)
>>> memory.endin
7
```

Type
int

equal_span(*address*)

Span of homogeneous data.

It searches for the biggest chunk of data adjacent to the given address, with the same value at that address.

If the address is within a gap, its bounds are returned, and its value is `None`.

If the address is before or after any data, bounds are `None`.

Parameters

address (*int*) – Reference address.

Returns

tuple – Start bound, exclusive end bound, and reference value.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory()
>>> memory.equal_span(0)
(None, None, None)
```

~~~

0	1	2	3	4	5	6	7	8	9	10
[A	B	B	B	C]			[C	C	D]	
65	66	66	66	67			67	67	68	

```
>>> memory = Memory.from_blocks([[0, b'ABBBC'], [7, b'CCD']])
>>> memory.equal_span(2)
(1, 4, 66)
>>> memory.equal_span(4)
(4, 5, 67)
>>> memory.equal_span(5)
(5, 7, None)
>>> memory.equal_span(10)
(10, None, None)
```

extend(*items*, *offset=0*)

Concatenates items.

Appends *items* after *content_endex*. Equivalent to `self += items`.

Parameters

- **items** (*items*) – Items to append at the end of the current virtual space.
- **offset** (*int*) – Optional offset w.r.t. *content_endex*.

See also:

__iadd__() extend_backup() extend_restore()
~~~

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.extend(b'123')
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz123']]
```

~~~

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.extend(range(49, 52), offset=4)
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz'], [12, b'123']]
```

~~~

```
>>> memory1 = Memory.from_blocks([[1, b'ABC']])
>>> memory2 = Memory.from_blocks([[5, b'xyz']])
>>> memory1.extend(memory2)
>>> memory1.to_blocks()
[[1, b'ABC'], [9, b'xyz']]
```

extend_backup(*offset=0*)

Backups an *extend()* operation.

Parameters

offset (*int*) – Optional offset w.r.t. *content_endex*.

Returns

int – Content exclusive end address.

See also:

extend() extend_restore()

extend_restore(*content_endex*)

Restores an *extend()* operation.

Parameters

content_endex (*int*) – Content exclusive end address to restore.

See also:

extend() extend_backup()

extract(*start=None*, *endex=None*, *pattern=None*, *step=None*, *bound=True*)

Selects items from a range.

Parameters

- **start** (*int*) – Inclusive start of the extracted range. If None, *start* is considered.
- **endex** (*int*) – Exclusive end of the extracted range. If None, *endex* is considered.
- **pattern** (*items*) – Optional pattern of items to fill the emptiness.

- **step** (*int*) – Optional address stepping between bytes extracted from the range. It has the same meaning of Python's `slice.step`, but negative steps are ignored. Please note that a *step* greater than 1 could take much more time to process than the default unitary step.
- **bound** (*bool*) – The selected address range is applied to the resulting memory as its bounds range. This retains information about any initial and final emptiness of that range, which would be lost otherwise.

Returns

`ImmutableMemory` – A copy of the memory from the selected range.

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.extract()._blocks
[[1, b'ABCD'], [6, b'$'], [8, b'xyz']]
>>> memory.extract(2, 9)._blocks
[[2, b'BCD'], [6, b'$'], [8, b'x']]
>>> memory.extract(start=2)._blocks
[[2, b'BCD'], [6, b'$'], [8, b'xyz']]
>>> memory.extract(endex=9)._blocks
[[1, b'ABCD'], [6, b'$'], [8, b'x']]
>>> memory.extract(5, 8).span
(5, 8)
>>> memory.extract(pattern=b'.')._blocks
[[1, b'ABCD.$.xyz']]
>>> memory.extract(pattern=b'.', step=3)._blocks
[[1, b'AD.z']]
```

fill(*start=None*, *endex=None*, *pattern=0*)

Overwrites a range with a pattern.

Parameters

- **start** (*int*) – Inclusive start address for filling. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address for filling. If `None`, *endex* is considered.
- **pattern** (*items*) – Pattern of items to fill the range.

See also:

fill_backup() fill_restore()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	[1	2	3	1	2	3	1	2]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.fill(pattern=b'123')
>>> memory.to_blocks()
[[1, b'12312312']]
```

~~~

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	[A	B	1	2	3	1	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.fill(3, 7, b'123')
>>> memory.to_blocks()
[[1, b'AB1231yz']]
```

fill_backup(*start=None*, *endex=None*)

Backups a *fill()* operation.

Parameters

- **start** (*int*) – Inclusive start address for filling. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for filling. If None, *endex* is considered.

Returns

ImmutableMemory – Backup memory region.

See also:

fill() fill_restore()

fill_restore(*backup*)

Restores a *fill()* operation.

Parameters

backup (ImmutableMemory) – Backup memory region to restore.

See also:

fill() fill_backup()

find(*item*, *start=None*, *endex=None*)

Index of an item.

Parameters
~~~

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
- **start** (*int*) – Inclusive start of the searched range. If None, *endex* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If None, *endex* is considered.

Returns

int – The index of the first item equal to *value*, or -1.

flood(*start=None*, *endex=None*, *pattern=0*)

Fills emptiness between non-touching blocks.

Parameters

- **start** (*int*) – Inclusive start address for flooding. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address for flooding. If None, *endex* is considered.
- **pattern** (*items*) – Pattern of items to fill the range.

See also:

flood_backup() flood_restore()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	[A	B	C	1	2	x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.flood(pattern=b'123')
>>> memory.to_blocks()
[[1, b'ABC12xyz']]
```

~~~

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	[A	B	C	2	3	x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.flood(3, 7, b'123')
>>> memory.to_blocks()
[[1, b'ABC23xyz']]
```

flood_backup(*start=None*, *endex=None*)

Backups a *flood()* operation.

Parameters

- **start** (*int*) – Inclusive start address for filling. If None, *start* is considered.
~~~

- **endex** (*int*) – Exclusive end address for filling. If None, *endex* is considered.

Returns
list of open intervals – Backup memory gaps.

See also:

flood() flood_restore()

flood_restore(*gaps*)

Restores a *flood()* operation.

Parameters
gaps (*list of open intervals*) – Backup memory gaps to restore.

See also:

flood() flood_backup()

classmethod from_blocks(*blocks*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

Creates a virtual memory from blocks.

Parameters

- **blocks** (*list of blocks*) – A sequence of non-overlapping blocks, sorted by address.
- **offset** (*int*) – Some address offset applied to all the blocks.
- **start** (*int*) – Optional memory start address. Anything before will be deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **copy** (*bool*) – Forces copy of provided input data.
- **validate** (*bool*) – Validates the resulting ImmutableMemory object.

Returns
ImmutableMemory – The resulting memory object.

Raises
ValueError – Some requirements are not satisfied.

See also:

to_blocks()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8
	[A	B	C]					
					[x	y	z]	

```
>>> blocks = [[1, b'ABC'], [5, b'xyz']]
>>> memory = Memory.from_blocks(blocks)
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'xyz']]
```

(continues on next page)

(continued from previous page)

```
>>> memory = Memory.from_blocks(blocks, offset=3)
>>> memory.to_blocks()
[[4, b'ABC'], [8, b'xyz']]
```

~~~

```
>>> # Loads data from an Intel HEX record file
>>> # NOTE: Record files typically require collapsing!
>>> import hexrec.records as hr  # noqa
>>> blocks = hr.load_blocks('records.hex')
>>> memory = Memory.from_blocks(Memory.collapse_blocks(blocks))
>>> memory
    ...
```

classmethod from_bytes(*data*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

Creates a virtual memory from a byte-like chunk.

Parameters

- **data** (*byte-like data*) – A byte-like chunk of data (e.g. `bytes`, `bytearray`, `memoryview`).
- **offset** (*int*) – Start address of the block of data.
- **start** (*int*) – Optional memory start address. Anything before will be deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **copy** (*bool*) – Forces copy of provided input data into the underlying data structure.
- **validate** (*bool*) – Validates the resulting `ImmutableMemory` object.

Returns

`ImmutableMemory` – The resulting memory object.

Raises

ValueError – Some requirements are not satisfied.

See also:

to_bytes()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'')
>>> memory.to_blocks()
[]
```

~~~

0	1	2	3	4	5	6	7	8
		[A	B	C	x	y	z]	

```
>>> memory = Memory.from_bytes(b'ABCxyz', 2)
>>> memory.to_blocks()
[[2, b'ABCxyz']]
```

classmethod from_items(*items*, *offset=0*, *start=None*, *endex=None*, *validate=True*)

Creates a virtual memory from a iterable address/byte mapping.

Parameters

- **items** (*iterable address/byte mapping*) – An iterable mapping of address to byte values. Values of `None` are translated as gaps. When an address is stated multiple times, the last is kept.
- **offset** (*int*) – An address offset applied to all the values.
- **start** (*int*) – Optional memory start address. Anything before will be deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **validate** (*bool*) – Validates the resulting `ImmutableMemory` object.

Returns

`ImmutableMemory` – The resulting memory object.

Raises

ValueError – Some requirements are not satisfied.

See also:

to_bytes()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_values({})
>>> memory.to_blocks()
[]
```

~~~

0	1	2	3	4	5	6	7	8
		[A	Z]		[x]			

```
>>> items = [
...     (0, ord('A')),
...     (1, ord('B')),
...     (3, ord('x')),
...     (1, ord('Z')),
... ]
>>> memory = Memory.from_items(items, offset=2)
>>> memory.to_blocks()
[[2, b'AZ'], [5, b'x']]
```
~~~

classmethod from_memory(*memory*, *offset=0*, *start=None*, *endex=None*, *copy=True*, *validate=True*)

Creates a virtual memory from another one.

Parameters

- **memory** (Memory) – A ImmutableMemory to copy data from.
- **offset** (*int*) – Some address offset applied to all the blocks.
- **start** (*int*) – Optional memory start address. Anything before will be deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **copy** (*bool*) – Forces copy of provided input data into the underlying data structure.
- **validate** (*bool*) – Validates the resulting MemorImmutableMemory object.

Returns

ImmutableMemory – The resulting memory object.

Raises

ValueError – Some requirements are not satisfied.

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory1 = Memory.from_bytes(b'ABC', 5)
>>> memory2 = Memory.from_memory(memory1)
>>> memory2._blocks
[[5, b'ABC']]
>>> memory1 == memory2
True
>>> memory1 is memory2
False
>>> memory1._blocks is memory2._blocks
False
```

~~~

```
>>> memory1 = Memory.from_bytes(b'ABC', 10)
>>> memory2 = Memory.from_memory(memory1, -3)
>>> memory2._blocks
[[7, b'ABC']]
>>> memory1 == memory2
False
```

~~~

```
>>> memory1 = Memory.from_bytes(b'ABC', 10)
>>> memory2 = Memory.from_memory(memory1, copy=False)
>>> all((b1[1] is b2[1])  # compare block data
...     for b1, b2 in zip(memory1._blocks, memory2._blocks))
True
```

classmethod from_values(*values*, *offset=0*, *start=None*, *endex=None*, *validate=True*)

Creates a virtual memory from a byte-like sequence.

Parameters

- **values** (*iterable byte-like sequence*) – An iterable sequence of byte values. Values of `None` are translated as gaps.
- **offset** (*int*) – An address offset applied to all the values.
- **start** (*int*) – Optional memory start address. Anything before will be deleted.
- **endex** (*int*) – Optional memory exclusive end address. Anything at or after it will be deleted.
- **validate** (*bool*) – Validates the resulting `ImmutableMemory` object.

Returns

`ImmutableMemory` – The resulting memory object.

Raises

ValueError – Some requirements are not satisfied.

See also:

to_bytes()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_values(range(0))
>>> memory.to_blocks()
[]
```

~~~

0	1	2	3	4	5	6	7	8
		[A	B	C	D	E]		

```
>>> memory = Memory.from_values(range(ord('A'), ord('F')), offset=2)
>>> memory.to_blocks()
[[2, b'ABCDE']]
```

classmethod fromhex(*string*)

Creates a virtual memory from an hexadecimal string.

Parameters

string (*str*) – Hexadecimal string.

Returns

`ImmutableMemory` – The resulting memory object.
~~~

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.fromhex('')
>>> bytes(memory)
b''
```

~~~

```
>>> memory = Memory.fromhex('48656C6C6F2C20576F726C6421')
>>> bytes(memory)
b'Hello, World!'
```

gaps(*start=None*, *endex=None*)

Iterates over block gaps.

Iterates over gaps emptiness bounds within an address range. If a yielded bound is `None`, that direction is infinitely empty (valid before or after global data bounds).

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.

Yields

pair of addresses – Block data interval boundaries.

See also:

intervals()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> list(memory.gaps())
[(None, 1), (3, 5), (6, 7), (10, None)]
>>> list(memory.gaps(0, 11))
[(0, 1), (3, 5), (6, 7), (10, 11)]
>>> list(memory.gaps(*memory.span))
[(3, 5), (6, 7)]
>>> list(memory.gaps(2, 6))
[(3, 5)]
```

get(*address*, *default=None*)

Gets the item at an address.
~~~

Returns
int – The item at *address*, *default* if empty.

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.get(3)  # -> ord('C') = 67
67
>>> memory.get(6)  # -> ord('$') = 36
36
>>> memory.get(10)  # -> ord('z') = 122
122
>>> memory.get(0)  # -> empty -> default = None
None
>>> memory.get(7)  # -> empty -> default = None
None
>>> memory.get(11)  # -> empty -> default = None
None
>>> memory.get(0, 123)  # -> empty -> default = 123
123
>>> memory.get(7, 123)  # -> empty -> default = 123
123
>>> memory.get(11, 123)  # -> empty -> default = 123
123
```

hex(**args*)

Converts into an hexadecimal string.

Parameters

- **sep** (*str*) – Separator string between bytes. Defaults to an emoty string if not provided. Available since Python 3.8.
- **bytes_per_sep** (*int*) – Number of bytes grouped between separators. Defaults to one byte per group. Available since Python 3.8.

Returns
str – Hexadecimal string representation.

Raises
ValueError – Data not contiguous (see *contiguous*).

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().hex() == ''
True
```

~~~

```
>>> memory = Memory.from_bytes(b'Hello, World!')
>>> memory.hex()
48656c6c6f2c20576f726c6421
>>> memory.hex('.')
48.65.6c.6c.6f.2c.20.57.6f.72.6c.64.21
>>> memory.hex('.', 4)
48.656c6c6f.2c20576f.726c6421
```

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.

insert(*address*, *data*)

Inserts data.

Inserts data, moving existing items after the insertion address by the size of the inserted data.

Arguments::

address (int):
Address of the insertion point.

data (bytes):
Data to insert.

See also:

insert_backup() insert_restore()
~~~

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C]			[x	y	z]			
	[A	B	C]			[x	y	z]		[$]	
	[A	B	C]			[x	y	1	z]		[$]

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.insert(10, b'$')
>>> memory.to_blocks()
[[1, b'ABC'], [6, b'xyz'], [10, b'$']]
>>> memory.insert(8, b'1')
>>> memory.to_blocks()
[[1, b'ABC'], [6, b'xy1z'], [11, b'$']]
```

insert_backup(*address*, *data*)

Backups an *insert()* operation.

Parameters

- **address** (*int*) – Address of the insertion point.
- **data** (*bytes*) – Data to insert.

Returns

(int, ImmutableMemory) – Insertion address, backup memory region.

See also:

insert() insert_restore()

insert_restore(*address*, *backup*)

Restores an *insert()* operation.

Parameters

- **address** (*int*) – Address of the insertion point.
- **backup** (*Memory*) – Backup memory region to restore.

See also:

insert() insert_backup()

intervals(*start=None*, *endex=None*)

Iterates over block intervals.

Iterates over data boundaries within an address range.

Parameters

- **start** (*int*) – Inclusive start address. If None, *start* is considered.
- **endex** (*int*) – Exclusive end address. If None, *endex* is considered.

Yields

pair of addresses – Block data interval boundaries.

See also:

blocks() gaps()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> list(memory.intervals())
[(1, 3), (5, 6), (7, 10)]
>>> list(memory.intervals(2, 9))
[(2, 3), (5, 6), (7, 9)]
>>> list(memory.intervals(3, 5))
[]
```

items(*start=None*, *endex=None*, *pattern=None*)

Iterates over address and value pairs.

Iterates over address and value pairs, from *start* to *endex*. Implemets the interface of `dict`.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered. If `Ellipsis`, the iterator is infinite.
- **pattern** (*items*) – Pattern of values to fill emptiness.

Yields

int – Range address and value pairs.

Examples

```
>>> from bytesparse import Memory
```

```
>>> from itertools import islice
>>> memory = Memory()
>>> list(memory.items(endex=8))
[(0, None), (1, None), (2, None), (3, None), (4, None), (5, None), (6, None),␣
↪(7, None)]
>>> list(memory.items(3, 8))
[(3, None), (4, None), (5, None), (6, None), (7, None)]
>>> list(islice(memory.items(3, ...), 7))
[(3, None), (4, None), (5, None), (6, None), (7, None), (8, None), (9, None)]
```

~~~
~~~

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	65	66	67			120	121	122	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> list(memory.items())
[(1, 65), (2, 66), (3, 67), (4, None), (5, None), (6, 120), (7, 121), (8, 122)]
>>> list(memory.items(3, 8))
[(3, 67), (4, None), (5, None), (6, 120), (7, 121)]
>>> list(islice(memory.items(3, ...), 7))
[(3, 67), (4, None), (5, None), (6, 120), (7, 121), (8, 122), (9, None)]
```

keys(*start=None*, *endex=None*)

Iterates over addresses.

Iterates over addresses, from *start* to *endex*. Implemets the interface of `dict`.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered. If `Ellipsis`, the iterator is infinite.

Yields

int – Range address.

Examples

```
>>> from bytesparse import Memory
```

```
>>> from itertools import islice
>>> memory = Memory()
>>> list(memory.keys())
[]
>>> list(memory.keys(endex=8))
[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]
```

(continues on next page)
~~~

(continued from previous page)

```
>>> list(memory.keys(endex=8))
[1, 2, 3, 4, 5, 6, 7]
>>> list(memory.keys(3, 8))
[3, 4, 5, 6, 7]
>>> list(islice(memory.keys(3, ...), 7))
[3, 4, 5, 6, 7, 8, 9]
```

peek(*address*)

Gets the item at an address.

Returns

int – The item at *address*, `None` if empty.

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.peek(3)  # -> ord('C') = 67
67
>>> memory.peek(6)  # -> ord('$') = 36
36
>>> memory.peek(10)  # -> ord('z') = 122
122
>>> memory.peek(0)
None
>>> memory.peek(7)
None
>>> memory.peek(11)
None
```

poke(*address*, *item*)

Sets the item at an address.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to set, `None` to clear the cell.

See also:

poke_backup() poke_restore()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.poke(3, b'@')
>>> memory.peek(3)  # -> ord('@') = 64
64
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.poke(5, b'@')
>>> memory.peek(5)  # -> ord('@') = 64
64
```

poke_backup(*address*)

Backups a *poke()* operation.

Parameters
address (*int*) – Address of the target item.

Returns
(int, int) – *address*, item at *address* (None if empty).

See also:

poke() poke_restore()

poke_restore(*address*, *item*)

Restores a *poke()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to restore.

See also:

poke() poke_backup()

pop(*address=None*, *default=None*)

Takes a value away.

Parameters

- **address** (*int*) – Address of the byte to pop. If None, the very last byte is popped.
- **default** (*int*) – Value to return if *address* is within emptiness.

Returns
int – Value at *address*; *default* within emptiness.

See also:

pop_backup() pop_restore()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
	[A	B	C	D]		[$]		[x	y]		
	[A	B	D]		[$]		[x	y]			

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.pop()  # -> ord('z') = 122
122
>>> memory.pop(3)  # -> ord('C') = 67
67
>>> memory.pop(6, 63)  # -> ord('?') = 67
63
```

pop_backup(*address=None*)

Backups a *pop()* operation.

Parameters

address (*int*) – Address of the byte to pop. If None, the very last byte is popped.

Returns

(int, int) – *address*, item at *address* (None if empty).

See also:

pop() pop_restore()

pop_restore(*address*, *item*)

Restores a *pop()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to restore, None if empty.

See also:

pop() pop_backup()

popitem()

Pops the last item.

Returns

(int, int) – Address and value of the last item.

See also:

popitem_backup() popitem_restore()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A]								[y	z]	

```
>>> memory = Memory.from_blocks([[1, b'A'], [9, b'yz']])
>>> memory.popitem()  # -> ord('z') = 122
(10, 122)
>>> memory.popitem()  # -> ord('y') = 121
(9, 121)
>>> memory.popitem()  # -> ord('A') = 65
(1, 65)
>>> memory.popitem()
Traceback (most recent call last):
    ...
KeyError: empty
```

popitem_backup()

Backups a *popitem()* operation.

Returns
(int, int) – Address and value of the last item.

See also:

popitem() popitem_restore()

popitem_restore(*address*, *item*)

Restores a *popitem()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to restore.

See also:

popitem() popitem_backup()

read(*address*, *size*)

Reads data.

Reads a chunk of data from an address, with a given size. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.
- **size** (*int*) – Chunk size.

Returns
memoryview – A view over the addressed chunk.

Raises
ValueError – Data not contiguous (see *contiguous*).

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> bytes(memory.read(2, 3))
b'BCD'
>>> bytes(memory.read(9, 1))
b'y'
>>> memory.read(4, 3)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.read(0, 6)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

readinto(*address*, *buffer*)

Reads data into a pre-allocated buffer.

Provided a pre-allocated writable buffer (*e.g.* a `bytearray` or a `memoryview` slice of it), this method reads a chunk of data from an address, with the size of the target buffer. Data within the range is required to be contiguous.

Parameters

- **address** (*int*) – Start address of the chunk to read.
- **buffer** (*writable*) – Pre-allocated buffer to fill with data.

Returns

int – Number of bytes read.

Raises

ValueError – Data not contiguous (see *contiguous*).

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> buffer = bytearray(3)
>>> memory.readinto(2, buffer)
```

(continues on next page)

(continued from previous page)

```
3
>>> buffer
bytearray(b'BCD')
>>> view = memoryview(buffer)
>>> memory.readinto(9, view[1:2])
1
>>> buffer
bytearray(b'ByD')
>>> memory.readinto(4, buffer)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.readinto(0, bytearray(6))
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

remove(*item*, *start=None*, *endex=None*)

Removes an item.

Searches and deletes the first occurrence of an item.

Parameters

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
- **start** (*int*) – Inclusive start of the searched range. If None, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If None, *endex* is considered.

Raises

ValueError – Item not found.

See also:

remove_backup() remove_restore()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
	[A	D]		[$]		[x	y	z]			
	[A	D]			[x	y	z]				

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.remove(b'BC')
>>> memory.to_blocks()
[[1, b'AD'], [4, b'$'], [6, b'xyz']]
>>> memory.remove(ord('$'))
>>> memory.to_blocks()
```

(continues on next page)

(continued from previous page)

```
[[1, b'AD'], [5, b'xyz']]
>>> memory.remove(b'?')
Traceback (most recent call last):
    ...
ValueError: subsection not found
```

remove_backup(*item*, *start=None*, *endex=None*)

Backups a *remove()* operation.

Parameters

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
- **start** (*int*) – Inclusive start of the searched range. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If `None`, *endex* is considered.

Returns

Memory – Backup memory region.

See also:

remove() remove_restore()

remove_restore(*backup*)

Restores a *remove()* operation.

Parameters

backup (*Memory*) – Backup memory region.

See also:

remove() remove_backup()

reserve(*address*, *size*)

Inserts emptiness.

Reserves emptiness at the provided address.

Parameters

- **address** (*int*) – Start address of the emptiness to insert.
- **size** (*int*) – Size of the emptiness to insert.

See also:

reserve_backup() reserve_restore()

Examples

```
>>> from bytesparse import Memory
```

2	3	4	5	6	7	8	9	10	11	12
	[A	B	C]		[x	y	z]			
	[A]			B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[3, b'ABC'], [7, b'xyz']])
>>> memory.reserve(4, 2)
>>> memory.to_blocks()
[[3, b'A'], [6, b'BC'], [9, b'xyz']]
```

~~~

2	3	4	5	6	7	8	9	10	11	12
			[A	B	C]		[x	y	z]	)))
								[A	B]	)))

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']], endex=12)
>>> memory.reserve(5, 5)
>>> memory.to_blocks()
[[10, b'AB']]
```

reserve_backup(*address*, *size*)

Backups a *reserve()* operation.

Parameters

- **address** (*int*) – Start address of the emptiness to insert.
- **size** (*int*) – Size of the emptiness to insert.

Returns

(int, ImmutableMemory) – Reservation address, backup memory region.

See also:

reserve() reserve_restore()

reserve_restore(*address*, *backup*)

Restores a *reserve()* operation.

Parameters

- **address** (*int*) – Address of the reservation point.
- **backup** (ImmutableMemory) – Backup memory region to restore.

See also:

reserve() reserve_backup()

reverse()

Reverses the memory in-place.

Data is reversed within the memory *span*.
~~~

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	
	[z	y	x]		[$]		[D	C	B	A]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.reverse()
>>> memory.to_blocks()
[[1, b'zyx'], [5, b'$'], [7, b'DCBA']]
```

~~~

0	1	2	3	4	5	6	7	8	9	10	11
		[[[		[A	B	C]				)))	
		[[[			[C	B	A]			)))	

```
>>> memory = Memory.from_bytes(b'ABCD', 3, start=2, endex=10)
>>> memory.reverse()
>>> memory.to_blocks()
[[5, b'CBA']]
```

rfind(*item*, *start=None*, *endex=None*)

Index of an item, reversed search.

Parameters

- **item** (*items*) – Value to find. Can be either some byte string or an integer.
- **start** (*int*) – Inclusive start of the searched range. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end of the searched range. If `None`, *endex* is considered.

Returns

int – The index of the last item equal to *value*, or -1.

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.
~~~

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]
```

(continues on next page)
~~~

(continued from previous page)

```
>>> list(memory.rvalues(3, 8, b'0123'))
[121, 120, 50, 49, 67]
```

setdefault(*address*, *default=None*)

Defaults a value.

Parameters

- **address** (*int*) – Address of the byte to set.
- **default** (*int*) – Value to set if *address* is within emptiness.

Returns

int – Value at *address*; *default* within emptiness.

See also:

setdefault_backup() setdefault_restore()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
	[A	B	C	D]		[$]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABCD'], [6, b'$'], [8, b'xyz']])
>>> memory.setdefault(3, b'@')  # -> ord('C') = 67
67
>>> memory.peek(3)  # -> ord('C') = 67
67
>>> memory.setdefault(5, 64)  # -> ord('@') = 64
64
>>> memory.peek(5)  # -> ord('@') = 64
64
>>> memory.setdefault(9) is None
False
>>> memory.peek(9) is None
False
>>> memory.setdefault(7) is None
True
>>> memory.peek(7) is None
True
```

setdefault_backup(*address*)

Backups a *setdefault()* operation.

Parameters

address (*int*) – Address of the byte to set.

Returns

(int, int) – *address*, item at *address* (`None` if empty).

See also:

setdefault() setdefault_restore()

setdefault_restore(*address*, *item*)

Restores a *setdefault()* operation.

Parameters

- **address** (*int*) – Address of the target item.
- **item** (*int or byte*) – Item to restore, None if empty.

See also:

setdefault() setdefault_backup()

shift(*offset*)

Shifts the items.

Parameters

offset (*int*) – Signed amount of address shifting.

See also:

shift_backup() shift_restore()

Examples

```
>>> from bytesparse import Memory
```

2	3	4	5	6	7	8	9	10	11	12
			[A	B	C]		[x	y	z]	
	[A	B	C]		[x	y	z]			

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']])
>>> memory.shift(-2)
>>> memory.to_blocks()
[[3, b'ABC'], [7, b'xyz']]
```

~~~

2	3	4	5	6	7	8	9	10	11	12
	[[[		[A	B	C]		[x	y	z]	
	[y	z]								

```
>>> memory = Memory.from_blocks([[5, b'ABC'], [9, b'xyz']], start=3)
>>> memory.shift(-8)
>>> memory.to_blocks()
[[2, b'yz']]
```

shift_backup(*offset*)

Backups a *shift()* operation.
~~~

Parameters
offset (*int*) – Signed amount of address shifting.

Returns
(int, ImmutableMemory) – Shifting, backup memory region.

See also:

shift() shift_restore()

shift_restore(*offset*, *backup*)

Restores an *shift()* operation.

Parameters

- **offset** (*int*) – Signed amount of address shifting.
- **backup** (ImmutableMemory) – Backup memory region to restore.

See also:

shift() shift_backup()

property span: Tuple[int, int]

Memory address span.

A `tuple` holding both *start* and *endex*.

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().span
(0, 0)
>>> Memory(start=1, endex=8).span
(1, 8)
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.span
(1, 8)
```

Type
tuple of int

property start: int

Inclusive start address.

This property holds the inclusive start address of the virtual space. By default, it is the current minimum inclusive start address of the first stored block.

If *bound_start* not None, that is returned.
~~~

If the memory has no data and no bounds, 0 is returned.

Examples

```
>>> from bytesparse import Memory
```

```
>>> Memory().start
0
```

~~~

0	1	2	3	4	5	6	7	8
	[A	B	C]		[x	y	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [5, b'xyz']])
>>> memory.start
1
```

~~~

0	1	2	3	4	5	6	7	8
	[[[				[x	y	z]	

```
>>> memory = Memory.from_blocks([[5, b'xyz']], start=1)
>>> memory.start
1
```

Type
int

`to_blocks`(*start=None*, *endex=None*)

Exports into blocks.

Exports data blocks within an address range, converting them into standalone `bytes` objects.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.

Returns
list of blocks – Exported data blocks.

See also:

blocks() from_blocks()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10
	[A	B]			[x]		[1	2	3]	

```
>>> memory = Memory.from_blocks([[1, b'AB'], [5, b'x'], [7, b'123']])
>>> memory.to_blocks()
[[1, b'AB'], [5, b'x'], [7, b'123']]
>>> memory.to_blocks(2, 9)
[[2, b'B'], [5, b'x'], [7, b'12']]
>>> memory.to_blocks(3, 5)]
[]
```

to_bytes(*start=None*, *endex=None*)

Exports into bytes.

Exports data within an address range, converting into a standalone `bytes` object.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered.

Returns

bytes – Exported data bytes.

See also:

from_bytes() view()

Examples

```
>>> from bytesparse import Memory
```

```
>>> memory = Memory.from_bytes(b'')
>>> memory.to_bytes()
b''
```

~~~

0	1	2	3	4	5	6	7	8
		[A	B	C	x	y	z]	

```
>>> memory = Memory.from_bytes(b'ABCxyz', 2)
>>> memory.to_bytes()
b'ABCxyz'
>>> memory.to_bytes(start=4)
```

(continues on next page)
~~~

(continued from previous page)

```
b'Cxyz'
>>> memory.to_bytes(endex=6)
b'ABCx'
>>> memory.to_bytes(4, 6)
b'Cx'
```

update(*data*, *clear=False*, ***kwargs*)

Updates data.

Parameters

- **data** (*iterable*) – Data to update with. Can be either another memory, an (address, value) mapping, or an iterable of (address, value) pairs.
- **clear** (*bool*) – Clears the target range before writing data. Useful only if *data* is a *Memory* with empty spaces.

See also:

update_backup() update_restore()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9	10	11
					[A	B	C]				
	[x	y]			[A	B	C]				
	[x	y	@]		[A	?	C]				

```
>>> memory = Memory()
>>> memory.update(Memory.from_bytes(b'ABC', 5))
>>> memory.to_blocks()
[[5, b'ABC']]
>>> memory.update({1: b'x', 2: ord('y')})
>>> memory.to_blocks()
[[1, b'xy'], [5, b'ABC']]
>>> memory.update([(6, b'?'), (3, ord('@'))])
>>> memory.to_blocks()
[[1, b'xy@'], [5, b'A?C']]
```

update_backup(*data*, *clear=False*, ***kwargs*)

Backups an *update()* operation.

Parameters

- **data** (*iterable*) – Data to update with. Can be either another memory, an (address, value) mapping, or an iterable of (address, value) pairs.
- **clear** (*bool*) – Clears the target range before writing data. Useful only if *data* is a *Memory* with empty spaces.

Returns
list of `ImmutableMemory` – Backup memory regions.

See also:

update() update_restore()

`update_restore`(*backups*)

Restores an *update()* operation.

Parameters
backups (list of `ImmutableMemory`) – Backup memory regions to restore.

See also:

update() update_backup()

`validate`()

Validates internal structure.

It makes sure that all the allocated blocks are sorted by block start address, and that all the blocks are non-overlapping.

Raises
ValueError – Invalid data detected (see exception message).

`values`(*start=None*, *endex=None*, *pattern=None*)

Iterates over values.

Iterates over values, from *start* to *endex*. Implemets the interface of `dict`.

Parameters

- **start** (*int*) – Inclusive start address. If `None`, *start* is considered.
- **endex** (*int*) – Exclusive end address. If `None`, *endex* is considered. If `Ellipsis`, the iterator is infinite.
- **pattern** (*items*) – Pattern of values to fill emptiness.

Yields
int – Range values.

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9
			A	B	C	D	A		
			65	66	67	68	65		

```
>>> from itertools import islice
>>> memory = Memory()
>>> list(memory.values(endex=8))
[None, None, None, None, None, None, None, None]
>>> list(memory.values(3, 8))
[None, None, None, None, None]
```

(continues on next page)

(continued from previous page)

```
>>> 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'
```

(continues on next page)
~~~

(continued from previous page)

```
>>> bytes(memory.view(9, 10))
b'y'
>>> memory.view()
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
>>> memory.view(0, 6)
Traceback (most recent call last):
    ...
ValueError: non-contiguous data within range
```

write(*address*, *data*, *clear=False*)

Writes data.

Parameters

- **address** (*int*) – Address where to start writing data.
- **data** (*bytes*) – Data to write.
- **clear** (*bool*) – Clears the target range before writing data. Useful only if *data* is a `ImmutableMemory` with empty spaces.

See also:

write_backup() write_restore()

Examples

```
>>> from bytesparse import Memory
```

0	1	2	3	4	5	6	7	8	9
	[A	B	C]			[x	y	z]	
	[A	B	C]		[1	2	3	z]	

```
>>> memory = Memory.from_blocks([[1, b'ABC'], [6, b'xyz']])
>>> memory.write(5, b'123')
>>> memory.to_blocks()
[[1, b'ABC'], [5, b'123z']]
```

write_backup(*address*, *data*, *clear=False*)

Backups a *write()* operation.

Parameters

- **address** (*int*) – Address where to start writing data.
- **data** (*bytes*) – Data to write.
- **clear** (*bool*) – Clears the target range before writing data. Useful only if *data* is a *Memory* with empty spaces.

Returns

list of `ImmutableMemory` – Backup memory regions.

See also:

write() *write_restore()*

write_restore(*backups*)

Restores a *write()* operation.

Parameters

backups (list of `ImmutableMemory`) – Backup memory regions to restore.

See also:

write() *write_backup()*

CHAPTER

FOUR

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.2 Documentation improvements

cbytesparse could always use more documentation, whether as part of the official cbytesparse docs, in docstrings, or even on the web in blog posts, articles, and such.

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/TexZK/cbytesparse/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

4.4 Development

To set up *cbytesparse* for local development:

1. Fork cbytesparse (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/cbytesparse.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

4.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`).
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

4.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

CHAPTER

FIVE

AUTHORS

- Andrea Zoppi - main developer - https://github.com/TexZK
- Caroline Russell - *Build Wheels* workflow - https://github.com/cerrussell

CHAPTER

SIX

CHANGELOG

6.1 0.0.6 (2023-02-18)

- Following the `bytesparse` Python package, version `0.0.6`.
- Added automatic wheel builder within GitHub Actions CI.
- Added *byte-like* methods and inplace editing.
- Added support to Python 3.11, removed 3.6.
- Added some minor features.
- Improved documentation.
- Improved testing.
- Improved repository layout (`pyproject.toml`).
- Minor fixes.

6.2 0.0.5 (2022-02-22)

- Following the `bytesparse` Python package, version `0.0.5`.
- Added `bytesparse` class, closer to `bytearray` than `Memory`.
- Added registration to the abstract base classes.
- Added missing abstract and ported methods.
- Added cut feature.
- Added more helper methods.
- Fixed values iteration.
- Improved extraction performance.
- Improved testing.

6.3 0.0.4 (2022-01-09)

- Following the `bytesparse` Python package, version `0.0.4`.
- Refactored current implementation as the `c` sub-module.
- Removed experimental backup feature.
- Added dedicated methods to backup/restore mutated state.
- Fixed some write/insert bugs.
- Fixed some trim/bound bugs.
- Methods sorted by name.
- Removed useless functions.

6.4 0.0.2 (2022-01-03)

- Forced extension compilation.
- Using explicit factory methods instead of constructor arguments.
- Added block collapsing helper function.
- Minor fixes.
- Improved test suite.

6.5 0.0.1 (2021-12-27)

- First release on PyPI.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z