bitlist

Release 1.2.0

Andrei Lapets

Aug 17, 2023

CONTENTS

1 Purpose **3**

2 Installation and Usage **5**
2.1 Examples 5

3 Development **7**
3.1 Documentation 7
3.2 Testing and Conventions 7
3.3 Contributions 8
3.4 Versioning 8
3.5 Publishing 8
3.5.1 bitlist module 8
3.5.2 examples 17

Python Module Index **19**

Index **21**

Pure-Python library for working with bit vectors.

CHAPTER

ONE

PURPOSE

This library allows programmers to work with bit vectors using a pure-Python data structure. Its design prioritizes interoperability with built-in Python classes and operators.

CHAPTER

TWO

INSTALLATION AND USAGE

This library is available as a package on PyPI:

```
python -m pip install bitlist
```

The library can be imported in the usual way:

```
import bitlist
from bitlist import bitlist
```

2.1 Examples

This library makes it possible to construct bit vectors from a variety of representations (including integers, bytes-like objects, strings of binary digits, lists of binary digits, and other bit vectors). Integer arguments are converted into a big-endian binary representation:

```
>>> bitlist(123)
bitlist('1111011')
>>> bitlist(bytes([255, 254]))
bitlist('1111111111111110')
>>> bitlist('101')
bitlist('101')
>>> bitlist([1, 0, 1, 1])
bitlist('1011')
>>> bitlist(bitlist('1010'))
bitlist('1010')
```

The optional `length` parameter can be used to specify the length of the created bit vector (padding consisting of zero bits is applied automatically *on the left-hand size*, if necessary):

```
>>> bitlist(bytes([123]), length=16)
bitlist('0000000001111011')
>>> bitlist(16, 64)
bitlist('0000000000000000000000000000000000000000000000000000000000010000')
>>> bitlist(bitlist(123), 8)
bitlist('01111011')
```

If the `length` parameter has a value that is less than the minimum number of bits that would be included according to the default constructor behaviors, the bit vector is truncated *on the left-hand side* to match the specified length:

```
>>> bitlist(bytes([123]), length=7)
bitlist('1111011')
>>> bitlist(bytes([123]), 4)
bitlist('1011')
>>> bitlist(bytes([123]), 2)
bitlist('11')
>>> bitlist(bytes([123]), 0)
bitlist()
```

Bit vectors are iterable sequences of individual bits (where each bit is represented as an integer). Both slice notation and retrieval of individual bits by index are supported. Furthermore, methods are available for converting a bit vector into other common representations:

```
>>> b = bitlist('1111011')
>>> b[1:-1]
bitlist('11101')
>>> b[0]
1
>>> [bit for bit in b]
[1, 1, 1, 1, 0, 1, 1]
>>> b.bin()
'1111011'
>>> b.hex()
'7b'
>>> list(b.to_bytes())
[123]
```

Concatenation, partitioning, subscription and slicing, shift and rotation, comparison, and logical operations are also supported by instances of the `bitlist` class. The larger example below – a bitwise addition function – illustrates the use of various operators supported by instances of the `bitlist` class:

```
>>> def add(x, y):
...     """Bitwise addition algorithm."""
...     r = bitlist(0)
...     carry = 0
...     # Use negative indices for big-endian interface.
...     for i in range(1, max(len(x), len(y)) + 1):
...         r[-i] = (x[-i] ^ y[-i]) ^ carry
...         carry = (x[-i] & y[-i]) | (x[-i] & carry) | (y[-i] & carry)
...     r[-(max(len(x), len(y)) + 1)] = carry
...     return r
...
>>> int(add(bitlist(123), bitlist(456)))
579
```

The testing script that accompanies this library contains additional examples of bitwise arithmetic operations implemented with the help of `bitlist` operators.

CHAPTER

THREE

DEVELOPMENT

All installation and development dependencies are fully specified in `pyproject.toml`. The `project.optional-dependencies` object is used to specify optional requirements for various development tasks. This makes it possible to specify additional options (such as `docs`, `lint`, and so on) when performing installation using pip:

```
python -m pip install .[docs,lint]
```

3.1 Documentation

The documentation can be generated automatically from the source files using Sphinx:

```
python -m pip install .[docs]
cd docs
sphinx-apidoc -f -E --templatedir=_templates -o _source .. && make html
```

3.2 Testing and Conventions

All unit tests are executed and their coverage is measured when using pytest (see the `pyproject.toml` file for configuration details):

```
python -m pip install .[test]
python -m pytest
```

The subset of the unit tests included in the module itself and the *documentation examples* that appear in the testing script can be executed separately using doctest:

```
python src/bitlist/bitlist.py -v
python test/test_bitlist.py -v
```

Style conventions are enforced using Pylint:

```
python -m pip install .[lint]
python -m pylint src/bitlist test/test_bitlist.py
```

3.3 Contributions

In order to contribute to the source code, open an issue or submit a pull request on the GitHub page for this library.

3.4 Versioning

Beginning with version 0.3.0, the version number format for this library and the changes to the library associated with version number increments conform with Semantic Versioning 2.0.0.

3.5 Publishing

This library can be published as a package on PyPI by a package maintainer. First, install the dependencies required for packaging and publishing:

```
python -m pip install .[publish]
```

Ensure that the correct version number appears in `pyproject.toml`, and that any links in this README document to the Read the Docs documentation of this package (or its dependencies) have appropriate version numbers. Also ensure that the Read the Docs project for this library has an automation rule that activates and sets as the default all tagged versions. Create and push a tag for this version (replacing `?.?.?` with the version number):

```
git tag ?.?.?
git push origin ?.?.?
```

Remove any old build/distribution files. Then, package the source into a distribution archive:

```
rm -rf build dist src/*.egg-info
python -m build --sdist --wheel .
```

Finally, upload the package distribution archive to PyPI:

```
python -m twine upload dist/*
```

3.5.1 bitlist module

Pure-Python library for working with bit vectors.

class bitlist.bitlist.**bitlist**(*argument: Optional[Union[int, str, bytes, bytearray, bitlist.bitlist.bitlist, Iterable[int]]] = None*, *length: Optional[int] = None*)

Bases: `object`

Data structure for representing bit vectors. The constructor accepts a variety of input types (including integers, bytes-like objects, strings of binary digits, iterables of binary digits, and other *`bitlist`* instances) and parses them in an appropriate manner to build a bit vector. Integer arguments are converted into a big-endian binary representation.

```
>>> bitlist(1)
bitlist('1')
>>> bitlist(123)
bitlist('1111011')
```

(continues on next page)

(continued from previous page)

```
>>> bitlist('1111011')
bitlist('1111011')
>>> bitlist(bytes([255, 254]))
bitlist('1111111111111110')
>>> bitlist([1, 0, 1, 1])
bitlist('1011')
>>> bitlist(bitlist('1010'))
bitlist('1010')
```

The *fromhex* method can be used to convert a hexadecimal string into an instance. This is equivalent to converting the hexadecimal string into a bytes-like object and then creating a bit vector from that object.

```
>>> bitlist.fromhex('abcd')
bitlist('1010101111001101')
>>> bitlist(bytes.fromhex('abcd'))
bitlist('1010101111001101')
```

A *bitlist* instance can be converted into an integer using the built-in `int` function. By default, a big-endian representation of integers is used. The recommended approach for switching to a little-endian representation is to reverse the bit vector.

```
>>> b = bitlist('1111011')
>>> int(b)
123
>>> int(bitlist(list(reversed(b))))
111
```

An instance can be converted into a string of binary characters using the *bin* method and into a hexadecimal string using the *hex* method. Conversion to a bytes-like object is possible via the built-in *to_bytes* method.

```
>>> b.bin()
'1111011'
>>> b.hex()
'7b'
>>> b.to_bytes().hex()
'7b'
>>> list(b.to_bytes())
[123]
```

An instance is itself a `Sequence` and is iterable. Iterating over an instance yields the sequence of bits (*i.e.*, integers that are `0` or `1`).

```
>>> [bit for bit in bitlist('10010011')]
[1, 0, 0, 1, 0, 0, 1, 1]
```

An individual bit can be retrieved (and assigned a new value) via its index. Slice notation is also supported.

```
>>> b = bitlist('0101')
>>> b[3]
1
>>> b[3] = 0
>>> b
bitlist('0100')
```

(continues on next page)

(continued from previous page)

```
>>> b[1:-1]
bitlist('10')
```

Instances are mutable. Conversion to an integer, binary string, or tuple is recommended if a corresponding immutable object is required.

```
>>> tuple(bitlist('1010'))
(1, 0, 1, 0)
```

When the constructor is supplied a *bitlist* instance, a distinct copy of the supplied instance is created.

```
>>> b = bitlist(123, 8)
>>> c = bitlist(b)
>>> c[0] = 1
>>> b
bitlist('01111011')
>>> c
bitlist('11111011')
```

When the constructor is applied to a bytes-like object, the leading zero digits (*i.e.*, those on the left-hand side) **are retained** (up to the least multiple of eight larger than the minimum number of binary digits required).

```
>>> bitlist(bytes([123]))
bitlist('01111011')
>>> bitlist(bytearray([123, 123]))
bitlist('0111101101111011')
```

When the constructor is applied to a string (consisting only of characters that correspond to binary digits), leading zero digits are also retained.

```
>>> bitlist('01111011')
bitlist('01111011')
```

However, when the constructor is applied to an integer argument, the created bit vector has no leading (*i.e.*, left-hand) zeros and contains the minimum number of bits necessary to represent the supplied argument (using a big-endian representation).

```
>>> bitlist(2)
bitlist('10')
>>> bitlist(16)
bitlist('10000')
```

The above implies that the empty bit vector represents (and is equivalent to) the numerical value of zero.

```
>>> bitlist() == bitlist(0)
True
>>> bitlist() == bitlist('0') == bitlist('00') == bitlist('000')
True
>>> bitlist() == bitlist('')
True
>>> bitlist() == bitlist([])
True
```

While the equality method *__eq__* determines equality between two bit vectors based on the integer values they represent, the fact that leading zero digits are retained in some cases means that two equivalent bit vectors *may have different lengths*.

```
>>> bitlist('0') == bitlist('000')
True
>>> len(bitlist('0')) == len(bitlist('000'))
False
```

For all other (non-integer) input types, the length of the vector (and consequently the number of leading zeos) is preserved.

```
>>> bitlist('0000')
bitlist('0000')
>>> bitlist([0, 1, 1])
bitlist('011')
>>> bitlist([0, 0, 1, 1])
bitlist('0011')
>>> bitlist(bitlist('00010'))
bitlist('00010')
```

Only strings that consist of characters corresponding to binary digits are accepted by the constructor.

```
>>> bitlist('abcd')
Traceback (most recent call last):
  ...
ValueError: each character in string must be '0' or '1'
```

To convert the underlying binary representation of a string into a bit vector, it is necessary to encode the string as a bytes-like object.

```
>>> bitlist('abcd'.encode('utf8'))
bitlist('01100001011000100110001101100100')
```

The `length` parameter can be used to specify the length of the bit vector, overriding the default behaviors. If the length parameter has a value that is greater than the number of bits that would be included according to a default behavior, the bit vector is padded with zero bits *on the left-hand side* to match the specified length.

```
>>> bitlist(bytes([123]), 16)
bitlist('0000000001111011')
>>> bitlist(16, 64)
bitlist('0000000000000000000000000000000000000000000000000000000000010000')
>>> bitlist(bitlist(123), 8)
bitlist('01111011')
```

If the `length` parameter has a value that is less than the minimum number of bits that would be included according to a default behavior, the bit vector is truncated *on the left-hand side* to match the specified length.

```
>>> bitlist(bytes([123]), 7)
bitlist('1111011')
>>> bitlist(bytes([123]), 4)
bitlist('1011')
>>> bitlist(bytes([123]), 2)
bitlist('11')
```

(continues on next page)

(continued from previous page)

```
>>> bitlist(bytes([123]), 0)
bitlist()
>>> bitlist(123, 0)
bitlist()
>>> bitlist([1, 1, 1], 0)
bitlist()
```

Any attempt to construct an instance using unsupported arguments raises an exception.

```
>>> bitlist([1.1, 2.2, 3.3])
Traceback (most recent call last):
  ...
TypeError: items in iterable must be integers
>>> bitlist([2, 3, 4, 5])
Traceback (most recent call last):
  ...
ValueError: each integer in iterable must be 0 or 1
>>> bitlist(float(1))
Traceback (most recent call last):
  ...
TypeError: bitlist constructor received unsupported argument
```

static fromhex(*s: str*) → *bitlist.bitlist.bitlist*

Build an instance from a hexadecimal string.

```
>>> bitlist.fromhex('abcd')
bitlist('1010101111001101')
```

__str__() → str

Return a string representation (that can also be evaluated as a valid Python expression if the class is in the namespace).

```
>>> bitlist('01')
bitlist('01')
```

__repr__() → str

Return a string representation (that can also be evaluated as a valid Python expression if the class is in the namespace).

__int__() → int

Interpret the bit vector as a big-endian representation of an integer and return that integer.

```
>>> int(bitlist(bytes([128, 129]))) == int.from_bytes(bytes([128, 129]), 'big')
True
```

to_bytes() → bytes

Return a bytes-like object representation. Note that the number of bits will be padded (on the left) to a multiple of eight.

```
>>> int.from_bytes(bitlist('10000000').to_bytes(), 'big')
128
>>> int.from_bytes(bitlist('1000000010000011').to_bytes(), 'big')
32899
```

(continues on next page)

(continued from previous page)

```
>>> int.from_bytes(bitlist('110000000').to_bytes(), 'big')
384
>>> bitlist(129 + 128*256).to_bytes().hex()
'8081'
>>> bitlist('11').to_bytes().hex()
'03'
```

bin() → str
Return a binary string representation. This matches the string emitted as part of the output of the default string conversion method.

```
>>> bitlist('010011').bin()
'010011'
```

hex() → str
Return a hexadecimal string representation. Note that the number of bits will be padded (on the left) to a multiple of eight.

```
>>> bitlist(bytes([123])).hex()
'7b'
```

__len__() → int
Return length of bit vector (defined to be the number of bits it contains).

```
>>> bitlist('11') + bitlist('10')
bitlist('1110')
```

__add__(*other:* bitlist.bitlist.bitlist) → *bitlist.bitlist.bitlist*
The addition operator can be used for concatenation, as with other objects that have sequence types.

```
>>> bitlist('11') + bitlist('10')
bitlist('1110')
```

__mul__(*other: int*) → *bitlist.bitlist.bitlist*
The multiplication operator can be used for repetition, as with other objects that have sequence types.

```
>>> bitlist(256)*2
bitlist('1000000000100000000')
>>> bitlist(256)*'a'
Traceback (most recent call last):
  ...
ValueError: repetition parameter must be an integer
```

__truediv__(*other: Union[int, Set[int], Sequence[int]]*) → Sequence[*bitlist.bitlist.bitlist*]
The division operator can be used to partition a bit vector into the specified number of parts, into parts of a specified length, or into a sequence of parts in which each part's length is specified in a sequence of integers (leveraging and mirroring the capabilities of the `parts` function).

```
>>> bitlist('11010001') / 2
[bitlist('1101'), bitlist('0001')]
>>> bitlist('11010001') / [2, 6]
[bitlist('11'), bitlist('010001')]
>>> bitlist('11010001') / {4}
```

(continues on next page)

(continued from previous page)

```
[bitlist('1101'), bitlist('0001')]
>>> bitlist('11010001') / 3
[bitlist('110'), bitlist('100'), bitlist('01')]
```

__getitem__(*key: Union[int, slice]*) → Union[int, *bitlist.bitlist.bitlist*]

Retrieve the bit at the specified index, or construct a slice of the bit vector.

```
>>> bitlist('1111011')[2]
1
>>> bitlist('0111011')[0]
0
>>> bitlist('10101000')[0:5]
bitlist('10101')
>>> bitlist('1010100010101000101010001010100 0')[0:16]
bitlist('1010100010101000')
>>> bitlist('101')[4]
Traceback (most recent call last):
  ...
IndexError: bitlist index out of range
>>> bitlist('101')['a']
Traceback (most recent call last):
  ...
TypeError: bitlist indices must be integers or slices
```

__setitem__(*i: int*, *b: int*)

Set the bit at the specified index to the supplied value.

```
>>> x = bitlist('1111011')
>>> x[2] = 0
>>> x
bitlist('1101011')
>>> x[7] = 0
Traceback (most recent call last):
  ...
IndexError: bitlist index out of range
```

__lshift__(*n: Union[int, Set[int]]*) → *bitlist.bitlist.bitlist*

The left shift operator can be used for both performing a bit shift in the traditional manner (increasing the length of the bit vector) or for bit rotation (if the second parameter is a set).

```
>>> bitlist('11') << 2
bitlist('1100')
>>> bitlist('11011') << {0}
bitlist('11011')
>>> bitlist('11011') << {1}
bitlist('10111')
>>> bitlist('11011') << {2}
bitlist('01111')
>>> bitlist('11011') << {3}
bitlist('11110')
>>> bitlist('11011') << {13}
bitlist('11110')
```

(continues on next page)

(continued from previous page)

```
>>> bitlist('1') << {13}
bitlist('1')
```

__rshift__(*n: Union[int, Set[int]]*) → *bitlist.bitlist.bitlist*

The right shift operator can be used for both performing a bit shift in the traditional manner (truncating bits on the right-hand side as necessary) or for bit rotation (if the second parameter is a set).

```
>>> bitlist('1111') >> 2
bitlist('11')
>>> bitlist('11011') >> {0}
bitlist('11011')
>>> bitlist('11011') >> {1}
bitlist('11101')
>>> bitlist('11011') >> {2}
bitlist('11110')
>>> bitlist('11011') >> {3}
bitlist('01111')
>>> bitlist('11011') >> {13}
bitlist('01111')
>>> bitlist('1') >> {13}
bitlist('1')
```

__and__(*other:* bitlist.bitlist.bitlist) → *bitlist.bitlist.bitlist*

Logical operators are applied bitwise without changing the length.

```
>>> bitlist('0100') & bitlist('1100')
bitlist('0100')
>>> bitlist('010') & bitlist('11')
Traceback (most recent call last):
  ...
ValueError: arguments to logical operations must have equal lengths
```

__or__(*other:* bitlist.bitlist.bitlist) → *bitlist.bitlist.bitlist*

Logical operators are applied bitwise without changing the length.

```
>>> bitlist('0100') | bitlist('1100')
bitlist('1100')
>>> bitlist('010') | bitlist('11')
Traceback (most recent call last):
  ...
ValueError: arguments to logical operations must have equal lengths
```

__xor__(*other:* bitlist.bitlist.bitlist) → *bitlist.bitlist.bitlist*

Logical operators are applied bitwise without changing the length.

```
>>> bitlist('0100') ^ bitlist('1101')
bitlist('1001')
>>> bitlist('010') ^ bitlist('11')
Traceback (most recent call last):
  ...
ValueError: arguments to logical operations must have equal lengths
```

__invert__() → *bitlist.bitlist.bitlist*

Logical operators are applied bitwise without changing the length. Inversion flips all bits and corresponds

to bitwise logical negation.

```
>>> ~bitlist('0100')
bitlist('1011')
```

__bool__() → bool

Any non-zero instance is interpreted as `True`.

```
>>> bool(bitlist('0100'))
True
>>> bool(bitlist('0000'))
False
```

__eq__(*other:* bitlist.bitlist.bitlist) → bool

Instances are interpreted as integers when relational operators are applied.

```
>>> bitlist('111') == bitlist(7)
True
>>> bitlist(123) == bitlist(0)
False
>>> bitlist(123) == bitlist('0001111011')
True
>>> bitlist('001') == bitlist('1')
True
```

__ne__(*other:* bitlist.bitlist.bitlist) → bool

Instances are interpreted as integers when relational operators are applied.

```
>>> bitlist('111') != bitlist(7)
False
>>> bitlist(123) != bitlist(0)
True
>>> bitlist('001') != bitlist('1')
False
```

__lt__(*other:* bitlist.bitlist.bitlist) → bool

Instances are interpreted as integers when relational operators are applied.

```
>>> bitlist(123) < bitlist(0)
False
>>> bitlist(123) < bitlist(123)
False
>>> bitlist(12) < bitlist(23)
True
```

__le__(*other:* bitlist.bitlist.bitlist) → bool

Instances are interpreted as integers when relational operators are applied.

```
>>> bitlist(123) <= bitlist(0)
False
>>> bitlist(123) <= bitlist(123)
True
>>> bitlist(12) <= bitlist(23)
True
```

__gt__(*other:* bitlist.bitlist.bitlist) → bool
Instances are interpreted as integers when relational operators are applied.

```
>>> bitlist(123) > bitlist(0)
True
>>> bitlist(123) > bitlist(123)
False
>>> bitlist(12) > bitlist(23)
False
```

__ge__(*other:* bitlist.bitlist.bitlist) → bool
Instances are interpreted as integers when relational operators are applied.

```
>>> bitlist(123) >= bitlist(0)
True
>>> bitlist(123) >= bitlist(123)
True
>>> bitlist(12) >= bitlist(23)
False
```

3.5.2 examples

Test suite containing examples and functional unit tests (based on bit vector implementations of common arithmetic operations) that demonstrate how this library's features can be used.

To view the source code of an example function, click its **[source]** link.

test.test_bitlist.**add**(*x:* bitlist.bitlist.bitlist, *y:* bitlist.bitlist.bitlist) → *bitlist.bitlist.bitlist*
Bitwise addition algorithm.

```
>>> int(add(bitlist(123), bitlist(456)))
579
```

test.test_bitlist.**mul**(*x:* bitlist.bitlist.bitlist, *y:* bitlist.bitlist.bitlist) → *bitlist.bitlist.bitlist*
Bitwise multiplication algorithm.

```
>>> int(mul(bitlist(123), bitlist(456)))
56088
```

test.test_bitlist.**exp**(*x:* bitlist.bitlist.bitlist, *y:* bitlist.bitlist.bitlist) → *bitlist.bitlist.bitlist*
Bitwise exponentiation algorithm.

```
>>> int(exp(bitlist(123), bitlist(5)))
28153056843
```

test.test_bitlist.**div**(*x:* bitlist.bitlist.bitlist, *y:* bitlist.bitlist.bitlist) → *bitlist.bitlist.bitlist*
Bitwise integer division algorithm.

```
>>> int(div(bitlist(12345), bitlist(678)))
18
```

PYTHON MODULE INDEX

b

bitlist.bitlist, 8

t

test.test_bitlist, 17

INDEX

Symbols

__add__() (*bitlist.bitlist.bitlist method*), 13
__and__() (*bitlist.bitlist.bitlist method*), 15
__bool__() (*bitlist.bitlist.bitlist method*), 16
__eq__() (*bitlist.bitlist.bitlist method*), 16
__ge__() (*bitlist.bitlist.bitlist method*), 17
__getitem__() (*bitlist.bitlist.bitlist method*), 14
__gt__() (*bitlist.bitlist.bitlist method*), 16
__int__() (*bitlist.bitlist.bitlist method*), 12
__invert__() (*bitlist.bitlist.bitlist method*), 15
__le__() (*bitlist.bitlist.bitlist method*), 16
__len__() (*bitlist.bitlist.bitlist method*), 13
__lshift__() (*bitlist.bitlist.bitlist method*), 14
__lt__() (*bitlist.bitlist.bitlist method*), 16
__mul__() (*bitlist.bitlist.bitlist method*), 13
__ne__() (*bitlist.bitlist.bitlist method*), 16
__or__() (*bitlist.bitlist.bitlist method*), 15
__repr__() (*bitlist.bitlist.bitlist method*), 12
__rshift__() (*bitlist.bitlist.bitlist method*), 15
__setitem__() (*bitlist.bitlist.bitlist method*), 14
__str__() (*bitlist.bitlist.bitlist method*), 12
__truediv__() (*bitlist.bitlist.bitlist method*), 13
__xor__() (*bitlist.bitlist.bitlist method*), 15

A

add() (*in module test.test_bitlist*), 17

B

bin() (*bitlist.bitlist.bitlist method*), 13
bitlist (*class in bitlist.bitlist*), 8
bitlist.bitlist
    module, 8

D

div() (*in module test.test_bitlist*), 17

E

exp() (*in module test.test_bitlist*), 17

F

fromhex() (*bitlist.bitlist.bitlist static method*), 12

H

hex() (*bitlist.bitlist.bitlist method*), 13

M

module
    bitlist.bitlist, 8
    test.test_bitlist, 17
mul() (*in module test.test_bitlist*), 17

T

test.test_bitlist
    module, 17
to_bytes() (*bitlist.bitlist.bitlist method*), 12