bitarray 0.8.1

efficient arrays of booleans -- C extension

Download
bitarray-0.8.1.tar.gz

This module provides an object type which efficiently represents an array of booleans. Bitarrays are sequence types and behave very much like usual lists. Eight bits are represented by one byte in a contiguous block of memory. The user can select between two representations; little-endian and big-endian. All of the functionality is implemented in C. Methods for accessing the machine representation are provided. This can be useful when bit level access to binary files is required, such as portable bitmap image files (.pbm). Also, when dealing with compressed data which uses variable bit length encoding, you may find this module useful.

Key features

- All functionality implemented in C.
- Bitarray objects behave very much like a list object, in particular slicing (including slice assignment and deletion) is supported.
- The bit endianness can be specified for each bitarray object, see below.
- On 32bit systems, a bitarray object can contain up to 2^34 elements, that is 16 Gbits (on 64bit machines up to 2^63 elements in theory – on Python 2.4 only 2^31 elements, see **PEP 353** (added in Python 2.5)).
- Packing and unpacking to other binary data formats, e.g. **numpy.ndarray**, is possible.
- Fast methods for encoding and decoding variable bit length prefix codes
- Sequential search (as list or iterator)
- Bitwise operations: `&, |, ^, &=, |=, ^=, ~`
- Pickling and unpickling of bitarray objects possible.
- Bitarray objects support the buffer protocol (Python 2.7 only)

Installation

bitarray can be installed from source:

```
$ tar xzf bitarray-0.8.1.tar.gz
$ cd bitarray-0.8.1
$ python setup.py install
```

On Unix systems, the latter command may have to be executed with root privileges. If you have **distribute** installed, you can easy_install bitarray. Once you have installed the package, you may want to test it:

```
$ python -c 'import bitarray; bitarray.test()'
bitarray is installed in: /usr/local/lib/python2.7/site-packages/bitarray
bitarray version: 0.8.1
2.7.2 (r271:86832, Nov 29 2010) [GCC 4.2.1 (SUSE Linux)]
.....................................................................
.......................................
----------------------------------------------------------------------
Ran 134 tests in 1.396s

OK
```

You can always import the function test, and `test().wasSuccessful()` will return True when the test

went well.

Using the module

As mentioned above, bitarray objects behave very much like lists, so there is not too new to learn. The biggest difference to list objects is the ability to access the machine representation of the object. When doing so, the bit endianness is of importance, this issue is explained in detail in the section below. Here, we demonstrate the basic usage of bitarray objects:

```
>>> from bitarray import bitarray
>>> a = bitarray()            # create empty bitarray
>>> a.append(True)
>>> a.extend([False, True, True])
>>> a
bitarray('1011')
```

Bitarray objects can be instantiated in different ways:

```
>>> a = bitarray(2**20)       # bitarray of length 1048576 (uninitialized)
>>> bitarray('1001011')       # from a string
bitarray('1001011')
>>> lst = [True, False, False, True, False, True, True]
>>> bitarray(lst)             # from list, tuple, iterable
bitarray('1001011')
```

Bits can be assigned from any Python object, if the value can be interpreted as a truth value. You can think of this as Python's built-in function bool() being applied, whenever casting an object:

```
>>> a = bitarray([42, '', True, {}, 'foo', None])
>>> a
bitarray('101010')
>>> a.append(a)      # note that bool(a) is True
>>> a.count(42)      # counts occurrences of True (not 42)
4L
>>> a.remove('')     # removes first occurrence of False
>>> a
bitarray('110101')
```

Like lists, bitarray objects support slice assignment and deletion:

```
>>> a = bitarray(50)
>>> a.setall(False)
>>> a[11:37:3] = 9 * bitarray([True])
>>> a
bitarray('00000000000100100100100100100100100100000000000000')
>>> del a[12::3]
>>> a
bitarray('00000000000101010101010101000000000')
>>> a[-6:] = bitarray('10011')
>>> a
bitarray('00000000000101010101010101100010011')
>>> a += bitarray('000111')
>>> a[9:]
bitarray('0010101010101010101100010011000111')
```

In addition, slices can be assigned to booleans, which is easier (and faster) than assigning to a bitarray in which all values are the same:

```
>>> a = 20 * bitarray('0')
```

```
>>> a[1:15:3] = True
>>> a
bitarray('01001001001001000000')
```

This is easier and faster than:

```
>>> a = 20 * bitarray('0')
>>> a[1:15:3] = 5 * bitarray('1')
>>> a
bitarray('01001001001001000000')
```

Note that in the latter we have to create a temporary bitarray whose length must be known or calculated.

Bit endianness

Since a bitarray allows addressing of individual bits, where the machine represents 8 bits in one byte, there two obvious choices for this mapping; little- and big-endian. When creating a new bitarray object, the endianness can always be specified explicitly:

```
>>> a = bitarray(endian='little')
>>> a.frombytes(b'A')
>>> a
bitarray('10000010')
>>> b = bitarray('11000010', endian='little')
>>> b.tobytes()
'C'
```

Here, the low-bit comes first because little-endian means that increasing numeric significance corresponds to an increasing address (index). So a[0] is the lowest and least significant bit, and a[7] is the highest and most significant bit.

```
>>> a = bitarray(endian='big')
>>> a.frombytes(b'A')
>>> a
bitarray('01000001')
>>> a[6] = 1
>>> a.tobytes()
'C'
```

Here, the high-bit comes first because big-endian means “most-significant first”. So a[0] is now the lowest and most significant bit, and a[7] is the highest and least significant bit.

The bit endianness is a property attached to each bitarray object. When comparing bitarray objects, the endianness (and hence the machine representation) is irrelevant; what matters is the mapping from indices to bits:

```
>>> bitarray('11001', endian='big') == bitarray('11001', endian='little')
True
```

Bitwise operations (`&, |, ^, &=, |=, ^=, ~`) are implemented efficiently using the corresponding byte operations in C, i.e. the operators act on the machine representation of the bitarray objects. Therefore, one has to be cautious when applying the operation to bitarrays with different endianness.

When converting to and from machine representation, using the `tobytes`, `frombytes`, `tofile` and `fromfile` methods, the endianness matters:

```
>>> a = bitarray(endian='little')
>>> a.frombytes(b'\x01')
```

```
>>> a
bitarray('10000000')
>>> b = bitarray(endian='big')
>>> b.frombytes(b'\x80')
>>> b
bitarray('10000000')
>>> a == b
True
>>> a.tobytes() == b.tobytes()
False
```

The endianness can not be changed once an object is created. However, since creating a bitarray from another bitarray just copies the memory representing the data, you can create a new bitarray with different endianness:

```
>>> a = bitarray('11100000', endian='little')
>>> a
bitarray('11100000')
>>> b = bitarray(a, endian='big')
>>> b
bitarray('00000111')
>>> a == b
False
>>> a.tobytes() == b.tobytes()
True
```

The default bit endianness is currently big-endian, however this may change in the future, and when dealing with the machine representation of bitarray objects, it is recommended to always explicitly specify the endianness.

Unless, explicitly converting to machine representation, using the `tobytes`, `frombytes`, `tofile` and `fromfile` methods, the bit endianness will have no effect on any computation, and one can safely ignore setting the endianness, and other details of this section.

Buffer protocol

Python 2.7 provides memoryview objects, which allow Python code to access the internal data of an object that supports the buffer protocol without copying. Bitarray objects support this protocol, with the memory being interpreted as simple bytes.

```
>>> a = bitarray('01000001' '01000010' '01000011', endian='big')
>>> v = memoryview(a)
>>> len(v)
3
>>> v[-1]
'C'
>>> v[:2].tobytes()
'AB'
>>> v.readonly  # changing a bitarray's memory is also possible
False
>>> v[1] = 'o'
>>> a
bitarray('010000010110111101000011')
```

Variable bit length prefix codes

The method `encode` takes a dictionary mapping symbols to bitarrays and an iterable, and extends the

bitarray object with the encoded symbols found while iterating. For example:

```
>>> d = {'H':bitarray('111'), 'e':bitarray('0'),
...      'l':bitarray('110'), 'o':bitarray('10')}
...
>>> a = bitarray()
>>> a.encode(d, 'Hello')
>>> a
bitarray('111011011010')
```

Note that the string `'Hello'` is an iterable, but the symbols are not limited to characters, in fact any immutable Python object can be a symbol. Taking the same dictionary, we can apply the `decode` method which will return a list of the symbols:

```
>>> a.decode(d)
['H', 'e', 'l', 'l', 'o']
>>> ''.join(a.decode(d))
'Hello'
```

Since symbols are not limited to being characters, it is necessary to return them as elements of a list, rather than simply returning the joined string.

Reference

The bitarray class:

`bitarray([initial], [endian=string])`

Return a new bitarray object whose items are bits initialized from the optional initial, and endianness. If no object is provided, the bitarray is initialized to have length zero. The initial object may be of the following types:

int, long
: Create bitarray of length given by the integer. The initial values in the array are random, because only the memory allocated.

string
: Create bitarray from a string of '0's and '1's.

list, tuple, iterable
: Create bitarray from a sequence, each element in the sequence is converted to a bit using truth value value.

bitarray
: Create bitarray from another bitarray. This is done by copying the memory holding the bitarray data, and is hence very fast.

The optional keyword arguments 'endian' specifies the bit endianness of the created bitarray object. Allowed values are 'big' and 'little' (default is 'big').

Note that setting the bit endianness only has an effect when accessing the machine representation of the bitarray, i.e. when using the methods: tofile, fromfile, tobytes, frombytes.

A bitarray object supports the following methods:

`all()` -> bool
: Returns True when all bits in the array are True.

`any()` -> bool
: Returns True when any bit in the array is True.

`append(item)`

Append the value bool(item) to the end of the bitarray.

`buffer_info()` -> tuple

Return a tuple (address, size, endianness, unused, allocated) giving the current memory address, the size (in bytes) used to hold the bitarray's contents, the bit endianness as a string, the number of unused bits (e.g. a bitarray of length 11 will have a buffer size of 2 bytes and 5 unused bits), and the size (in bytes) of the allocated memory.

`bytereverse()`

For all bytes representing the bitarray, reverse the bit order (in-place). Note: This method changes the actual machine values representing the bitarray; it does not change the endianness of the bitarray object.

`copy()` -> bitarray

Return a copy of the bitarray.

`count([value])` -> int

Return number of occurrences of value (defaults to True) in the bitarray.

`decode(code)` -> list

Given a prefix code (a dict mapping symbols to bitarrays), decode the content of the bitarray and return the list of symbols.

`encode(code, iterable)`

Given a prefix code (a dict mapping symbols to bitarrays), iterates over iterable object with symbols, and extends the bitarray with the corresponding bitarray for each symbols.

`endian()` -> string

Return the bit endianness as a string (either 'little' or 'big').

`extend(object)`

Append bits to the end of the bitarray. The objects which can be passed to this method are the same iterable objects which can given to a bitarray object upon initialization.

`fill()` -> int

Adds zeros to the end of the bitarray, such that the length of the bitarray is not a multiple of 8. Returns the number of bits added (0..7).

`frombytes(bytes)`

Append from a byte string, interpreted as machine values.

`fromfile(f, [n])`

Read n bytes from the file object f and append them to the bitarray interpreted as machine values. When n is omitted, as many bytes are read until EOF is reached.

`fromstring(string)`

Append from a string, interpreting the string as machine values. Deprecated since version 0.4.0, use `frombytes()` instead.

`index(value, [start, [stop]])` -> int

Return index of the first occurrence of bool(value) in the bitarray. Raises ValueError if the value is not present.

`insert(i, item)`

Insert bool(item) into the bitarray before position i.

`invert()`

Invert all bits in the array (in-place), i.e. convert each 1-bit into a 0-bit and vice versa.

`iterdecode(code)` -> iterator

Given a prefix code (a dict mapping symbols to bitarrays), decode the content of the bitarray and iterate over the symbols.

`itersearch(bitarray)` -> iterator

Searches for the given a bitarray in self, and return an iterator over the start positions where bitarray matches self.

`length()` -> int

Return the length, i.e. number of bits stored in the bitarray. This method is preferred over __len__ (used when typing `len(a)`), since __len__ will fail for a bitarray object with `2^31` or more elements on a 32bit machine, whereas this method will return the correct value, on 32bit and 64bit machines.

`pack(bytes)`

Extend the bitarray from a byte string, where each characters corresponds to a single bit. The

character b'x00' maps to bit 0 and all other characters map to bit 1. This method, as well as the unpack method, are meant for efficient transfer of data between bitarray objects to other python objects (for example NumPy's ndarray object) which have a different view of memory.

`pop([i])` -> item
: Return the i-th (default last) element and delete it from the bitarray. Raises IndexError if bitarray is empty or index is out of range.

`remove(item)`
: Remove the first occurrence of bool(item) in the bitarray. Raises ValueError if item is not present.

`reverse()`
: Reverse the order of bits in the array (in-place).

`search(bitarray, [limit])` -> list
: Searches for the given a bitarray in self, and returns the start positions where bitarray matches self as a list. The optional argument limits the number of search results to the integer specified. By default, all search results are returned.

`setall(value)`
: Set all bits in the bitarray to bool(value).

`sort(reverse=False)`
: Sort the bits in the array (in-place).

`to01()` -> string
: Return a string containing '0's and '1's, representing the bits in the bitarray object. Note: To extend a bitarray from a string containing '0's and '1's, use the extend method.

`tobytes()` -> bytes
: Return the byte representation of the bitarray. When the length of the bitarray is not a multiple of 8, the few remaining bits (1..7) are set to 0.

`tofile(f)`
: Write all bits (as machine values) to the file object f. When the length of the bitarray is not a multiple of 8, the remaining bits (1..7) are set to 0.

`tolist()` -> list
: Return an ordinary list with the items in the bitarray. Note that the list object being created will require 32 or 64 times more memory than the bitarray object, which may cause a memory error if the bitarray is very large. Also note that to extend a bitarray with elements from a list, use the extend method.

`tostring()` -> string
: Return the string representing (machine values) of the bitarray. When the length of the bitarray is not a multiple of 8, the few remaining bits (1..7) are set to 0. Deprecated since version 0.4.0, use `tobytes()` instead.

`unpack(zero=b'\x00', one=b'\xff')` -> bytes
: Return a byte string containing one character for each bit in the bitarray, using the specified mapping. See also the pack method.

Functions defined in the module:

`test(verbosity=1, repeat=1)` -> TextTestResult
: Run self-test, and return unittest.runner.TextTestResult object.

`bitdiff(a, b)` -> int
: Return the difference between two bitarrays a and b. This is function does the same as (a ^ b).count(), but is more memory efficient, as no intermediate bitarray object gets created

`bits2bytes(n)` -> int
: Return the number of bytes necessary to store n bits.

Change log

0.8.1 (2013-03-30):

- fix issue #10, i.e. int(bitarray()) segfault

- added tests for using a bitarray object as an argument to functions like int, long (on Python 2), float, list, tuple, dict

0.8.0 (2012-04-04):

- add Python 2.4 support
- add (module level) function bitdiff for calculating the difference between two bitarrays

0.7.0 (2012-02-15):

- add iterdecode method (C level), which returns an iterator but is otherwise like the decode method
- improve memory efficiency and speed of pickling large bitarray objects

Please find the complete change log **here**.

File	Type	Py Version	Uploaded on	Size
bitarray-0.8.1.tar.gz (**md5**)	Source		2013-03-31	45KB

Downloads (All Versions):
646 downloads in the last day
2505 downloads in the last week
10359 downloads in the last month

Author: Ilan Schnell
Home Page: https://github.com/ilanschnell/bitarray
License: PSF
Categories
Development Status :: 5 - Production/Stable
Intended Audience :: Developers
License :: OSI Approved :: Python Software Foundation License
Operating System :: OS Independent
Programming Language :: C
Programming Language :: Python :: 2
Programming Language :: Python :: 2.4
Programming Language :: Python :: 2.5
Programming Language :: Python :: 2.6
Programming Language :: Python :: 2.7
Programming Language :: Python :: 3
Programming Language :: Python :: 3.1
Programming Language :: Python :: 3.2
Programming Language :: Python :: 3.3
Topic :: Utilities
Package Index Owner: ilanschnell
DOAP record: bitarray-0.8.1.xml