011000100110100101110100011100110111

bitstring

010001110010011010010110111001100111

A Python module to help you manage your bits

This document describes version 0.4.2 of the bitstring module.
To download the latest version go to http://python-bitstring.googlecode.com.

Wednesday, 27 May 2009

Introduction

The bitstring module is a pure Python module designed to allow binary data to be read, interpreted, created and modified with as much ease as possible. A single class, `BitString`, is provided that stores the binary data and offers a rich variety of methods for all your binary needs.

A `BitString` object is designed to be as lightweight as possible and is best considered as just a list of binary digits. Although there are a variety of ways of creating and viewing the binary data, the `BitString` itself just stores the data and all views are calculated as needed, and are not stored as part of the object.

The different views or interpretations on the data are accessed through properties such as `hex`, `bin` and `int`.

A flavour is given here, and will be covered in greater detail in the next few pages of this tutorial.

```
from bitstring import BitString

# Just some of the ways to create BitStrings
a = BitString('0b001')                      # from a binary string
b = BitString('0xff470001')                 # from a hexadecimal string
c = BitString(filename='somefile.ext')      # straight from a file
d = BitString(int=540, length=11)           # from an integer

# Easily construct new BitStrings
e = 5*a + '0xcdcd'                          # 5 copies of 'a' followed by two new
                                            # bytes
e.prepend('0b1')                            # put a single bit on the front
f = e[7:]                                   # cut the first 7 bits off
f[1:4] = '0o775'                            # replace 3 bits with 9 bits from
                                            # octal string
f.replace('0b01', '0xee34', False)          # find and replace 2 bit string with
                                            # 16 bit string

# Interpret the BitString however you want
print e.hex                                 # 0x9249cdcd
print e.int                                 # -1840656947 (signed 2's complement
                                            # integer)
print e.uint                                # 2454310349  (unsigned integer)
open('somefile.ext', 'rb').write(e.data)    # Output raw byte data to a file
```

Getting Started

First download the latest release for either Python 2.4 / 2.5 / 2.6 or 3.0 (see the Downloads tab on the project's homepage) and extract the contents of the zip. You should find:

- `bitstring.py` : The bitstring module itself.
- `test_bitstring.py` : Unit tests for the module.
- `setup.py` : The setup script.
- `readme.txt` : A short readme.
- `release_notes.txt` : History of changes in this and previous versions.
- `test/` : Directory for test files
  - `test.m1v` : An example file (MPEG-1 video) for testing purposes.
  - `smalltestfile` : Another small file for testing.

To install, run

```
python setup.py install
```

This will copy `bitstring.py` to your Python installation's `site-packages` directory. If you prefer you can do this by hand, or just make sure that your Python program can see `bitstring.py`, for example by putting in the same directory as the program that will use it.

The module comes with comprehensive unit tests. To run them yourself use

```
python test_bitstring.py
```

which should run all the tests (over 200) and say OK. If tests fail then either your version of Python isn't supported (there's one version of bitstring for Python 2.4, 2.5 and 2.6 and a separate version for Python 3.0) or something unexpected has happened - in which case please tell me about it.

Creation and Interpretation

You can create `BitString` objects in a variety of ways. Internally, `BitString` objects are stored as byte arrays. This means that no space is wasted and a `BitString` containing 10MB of binary data will only take up 10MB of memory. When a `BitString` is created all that is stored is the byte array, the length in bits and possibly an offset to the first used bit in the byte array. This means that the actual initialiser used to create the `BitString` isn't stored itself - if you create using a hex string for example then if you ask for the hex interpretation it has to be calculated from the stored byte array.

Using the constructor

When initialising a `BitString` you need to specify at most one initialiser. These will be explained in full below, but briefly they are:

- `auto` : Either a string prefixed with '`0x`', '`0o`' or '`0b`' to interpret as hexadecimal, octal or binary, another `BitString` or a list or tuple.
- `data` : A Python string, for example read from a binary file.
- `hex`, `oct`, `bin`: Hexadecimal, octal or binary string.
- `int`, `uint`: Signed or unsigned binary integers.
- `se`, `ue` : Signed or unsigned exponential-Golomb coded integers.
- `filename` : Directly from a file.

For some of the initialisers you need to also specify the length in bits, for some it is optional and for others it is an error and this will be detailed below.

The auto initialiser

```
>>> fromhex = BitString('0x01ffc9')
>>> frombin = BitString('0b01')
>>> fromoct = BitString('0o7550')
>>> acopy   = BitString(fromoct)
>>> alist   = BitString([True, False, False])
```

The simplest way to create a `BitString` is often to use the `auto` parameter, which is the first parameter in the `__init__` function and so the `auto=` can be omitted. It accepts a number of different objects. Strings that start with '`0x`' are interpreted as hexadecimal, '`0o`' implies octal, and strings starting with '`0b`' are interpreted as binary. It also accepts another `BitString`, to create a copy of it and lists and tuples are interpreted as boolean arrays.

Note that as always the `BitString` doesn't know how it was created. Initialising with octal or hex might be more convenient or natural for a particular example but it is exactly equivalent to initialising with the corresponding binary string.

```
>>> fromoct.oct
‘0o7550’
>>> fromoct.hex
```

```
'0xf68'
>>> fromoct.bin
'0b111101101000'
>>> fromoct.uint
3994
>>> fromoct.int
-152
>>> BitString('0o7777') == '0xfff'
True
>>> BitString('0xf') == '0b1111'
True
```

From raw data

For most initialisers you can also use the `length` and `offset` parameters to specify the length in bits and an offset at the start to be ignored. This is particularly useful when initialising from raw data for from a file.

```
a = BitString(data='\x00\x01\x02\xff', length=28, offset=1)
b = BitString(data=open("somefile", 'rb').read())
```

The `length` parameter is optional; it defaults to the length of the data in bits (and so will be a multiple of 8). You can use it to truncate some bits from the end of the `BitString`. The `offset` parameter is used to ignore bits at the start of the data.

From a file

Using the `filename` initialiser allows a file to be analysed without the need to read it all into memory. The way to create a file-based `BitString` is:

```
p = BitString(filename="my2GBfile")
```

which will open the file in binary read-only mode. The file will only be read as and when other operations require it, and the contents of the file will not be changed by any operations. Something to watch out for are operations that could cause a copy of large parts of the object to be made in memory, for example

```
p2 = p[8:]
p += '0x00'
```

will create two new memory-based `BitString` objects with about the same size as the whole of the file's data. This is probably not what is wanted as the reason for using the `filename` initialiser is likely to be because you don't want the whole file in memory.

From a hexadecimal string

```
c = BitString(hex='0x000001b3')
```

The initial '`0x`' or '`0X`' is optional, as once again is a `length` parameter, which can be used to truncate the end. Whitespace is also allowed and is ignored. Note that the leading zeros are significant, so the length of `c` will be `32`.

If you include the initial '`0x`' then you can use the `auto` initialiser, which just happens to be the first parameter in `__init__`, so this will work equally well:

```
c = BitString('0x000001b3')
```

From a binary string

```
>>> d = BitString(bin='0011 000', length=6)
>>> d.bin
```

```
'0b001100'
```

An initial '`0b`' or '`0B`' is optional. Once again a `length` can optionally be supplied to truncate the `BitString` and whitespace will be ignored.

As with `hex`, the `auto` initialiser will work for binary strings prefixed by '`0b`':

```
>>> d = BitString('0b001100')
```

From an octal string

```
>>> o = BitString(oct='34100')
>>> o.oct
'0o34100'
```

An initial '`0o`' or '`0O`' is optional, but '`0o`' is preferred as it is slightly more readable. Once again a `length` can optionally be supplied to truncate the `BitString` and whitespace will be ignored.

As with `hex` and `bin`, the `auto` initialiser will work for octal strings prefixed by '`0o`':

```
>>> o = BitString('0o34100')
```

From an integer

```
>>> e = BitString(uint=45, length=12)
>>> f = BitString(int=-1, length=7)
>>> e.bin
'0b000000101101'
>>> f.bin
'0b1111111'
```

For initialisation with signed and unsigned binary integers (`int` and `uint` repectively) the `length` parameter is mandatory, and must be large enough to contain the integer. So for example if `length` is `8` then `uint` can be in the range `0` to `255`, while `int` can range from `-128` to `127`. Two's complement is used to represent negative numbers.

```
>>> g = BitString(ue=12)
>>> h = BitString(se=-402)
>>> g.bin
'0b0001101'
>>> h.bin
'0b0000000001100100101'
```

Here we initialise again with integers, but this time the binary representation will be exponential-Golomb codes (`ue` is unsigned, `se` is signed). For these initialisers the length of the `BitString` is fixed by the value it is initialised with, so the `length` parameter must not be supplied and it is an error to do so. If you don't know what exponential-Golomb codes are then you probably don't need to know, but they are quite interesting, so I've included an appendix on them (see Appendix A).

Interpreting BitStrings

`BitString` objects don't know or care how they were created; they are just collections of bits. This means that you are quite free to interpret them in any way that makes sense.

Several Python properties are used to create interpretations for the `BitString`. These properties call functions such as `_gethex()` and `_getuint()` which will calculate and return the appropriate interpretation. These don't change the `BitString` in any way and it remains just a collection of bits. If you use the property again then the calculation will be repeated.

For the properties described below we will use these:

```
>>> a = BitString('0x123')
>>> b = BitString('0b111')
```

bin

The most fundamental interpretation is perhaps as a binary number. The `bin` property returns a string of the binary representation of the `BitString` prefixed with `0b`. All `BitString` objects can use this property and it is used to test equality between `BitString` objects.

```
>>> a.bin
'0b000100100011'
>>> b.bin
'0b111'
```

Note that the initial zeros are significant; for `BitString` objects they're just as important as the ones!

hex

For whole-byte `BitString` objects the most natural interpretation is often as hexadecimal, with each byte represented by two hex digits. Hex values are prefixed with `0x`.

If the `BitString` does not have a length that is a multiple of four then a `ValueError` exception will be raised. This is done in preference to truncating or padding the value, which could hide errors in user code.

The `hex` built-in function can also be used, with exactly the same effect.

```
>>> a.hex
'0x123'
>>> hex(a)
'0x123'
>>> b.hex
ValueError: Cannot convert to hex unambiguously - not multiple of 4 bits.
```

oct

For an octal interpretation use the `oct` property or the `oct` built-in function. Octal values are prefixed with `0o`, which is the Python 2.6 / 3.0 way of doing things (rather than just starting with `0`).

If the `BitString` does not have a length that is a multiple of three then a `ValueError` exception will be raised.

```
>>> a.oct
'0o0443'
>>> oct(a)
'0o0443'
>>> b.oct
'0o7'
>>> (b + '0b0').oct
ValueError: Cannot convert to octal unambiguously - not multiple of 3 bits.
```

uint

To interpret the `BitString` as a binary (base-2) unsigned integer (i.e. a non-negative integer) use the `uint` property.

```
>>> a.uint
283
```

```
>>> b.uint
7
```

int

For a two's complement interpretation as a base-2 signed integer use the `int` property. If the first bit of the `BitString` is zero then the `int` and `uint` interpretations will be equal, otherwise the `int` will represent a negative number.

```
>>> a.int
283
>>> b.int
-1
```

data

A common need is to retrieve the raw bytes from a `BitString` for further processing or for writing to a file. For this use the `data` interpretation, which returns an ordinary Python string.

If the length of the `BitString` isn't a multiple of eight then it will be padded with between one and seven zero bits up to a byte boundary.

```
>>> open('somefile', 'wb').write(a.data)
>>> a2 = BitString(filename='somefile')
>>> a2.hex
0x1230
```

Note the extra four bits that were needed to byte align.

ue

The `ue` property interprets the `BitString` as a single unsigned exponential-Golomb code and returns an integer. If the `BitString` is not exactly one code then a `BitStringError` is raised instead. If you wish to read the next bits in the stream and interpret them as a code the use the `readue` function. See Appendix A in this tutorial for a short explanation of this type of integer representation.

se

The `se` property does much the same as `ue` and the provisos there all apply. The obvious difference is that it interprets the `BitString` as a signed exponential-Golomb rather than unsigned - see Appendix A in this tutorial for more information.

Slicing, Dicing and Splicing

Slicing

Slicing can be done in couple of ways. The `slice` function takes two arguments: the first bit position you want and one past the last bit position you want, so for example `a.slice(10,12)` will return a 2-bit `BitString` of the 10th and 11th bits in `a`.

An equivalent method is to use indexing: `a[10:12]`. Note that as always the unit is bits, rather than bytes.

```
>>> a = BitString('0b00011110')
>>> b = a[3:7]
>>> c = a.slice(3, 7)            # s.slice(x, y) is equivalent to s[x:y]
>>> print a, b, c
0x1e 0xf 0xf
```

Indexing also works for missing and negative arguments, just as it does for other containers.

```
>>> a = BitString('0b00011110')
>>> print a[:5]         # first 5 bits
0b00011
>>> print a[3:]         # everything except first 3 bits
0b11110
>>> print a[-4:]        # final 4 bits
0xe
>>> print a[:-1]        # everything except last bit
0b0001111
>>> print a[-6:-4]      # from 6 from the end to 4 from the end
0b01
```

Joining

To join together a couple of `BitString` objects use the + or += operators, or the `append` and `prepend` functions.

```
# Six ways of creating the same BitString:
a1 = BitString(bin='000') + BitString(hex='f')
a2 = BitString('0b000') + BitString('0xf')
a3 = BitString('0b000') + '0xf'
a4 = BitString('0b000').append('0xf')
a5 = BitString('0xf').prepend('0b000')
a6 = BitString('0b000')
a6 += '0xf'
```

If you want to join a large number of `BitString` objects then the function `join` can be used to improve efficiency and readability.

```
# Don't do it this way!
s = BitString()
for bs in bslist:
    s = s + bs

# This is much more efficient:
s = bitstring.join(bslist)
```

Truncating

The truncate functions modify the `BitString` that they operate on, but also return themselves.

```
>>> a = BitString('0x001122')
>>> a.truncateend(8)
BitString('0x0011')
>>> b = a.truncatestart(8)
>>> a == b == '0x11'
True
```

A similar effect can be obtained using slicing - the major difference being that a new `BitString` is returned and the `BitString` being operated on remains unchanged.

Inserting, deleting and overwriting

`insert` takes one `BitString` and inserts it into another. A bit position can be specified, but if not present then the current `bitpos` is used.

```
>>> a = BitString('0x00112233')
>>> b = BitString('0xffff')
>>> a.insert(b, 16)
>>> a.hex
'0x0011ffff2233'
```

You can also use a string with `insert`, which will be interpreted as a binary or hexadecimal string. So the previous example could be written without using `b` as:

```
>>> a.insert('0xffff', 16)
```

`overwrite` does much the same as `insert`, but as you might expect the `BitString` object's data is overwritten by the new data.

```
>>> a = BitString('0x00112233')
>>> a.bitpos = 4
>>> a.overwrite('0b1111')         # Uses current bitpos as default
>>> a.hex
'0x0f112233'
```

`deletebits` and `deletebytes` remove sections of the `BitString`. By default they remove at the current `bitpos` - this must be at a byte boundary if using `deletebytes`:

```
>>> a = BitString('0b00011000')
>>> a.deletebits(2, 3)            # remove 2 bits at bitpos 3
>>> a.bin
'0b000000'

>>> b = BitString('0x112233445566')
>>> b.bytepos = 3
>>> b.deletebytes(2)
>>> b.hex
'0x11223366'
```

Splitting

Sometimes it can be very useful to use a delimiter to split a `BitString` into sections. The `split` function returns a generator for the sections.

```
>>> a = BitString('0x4700004711472222')
>>> for s in a.split('0x47'):
...     print "Empty" if s.empty() else s.hex
Empty
0x470000
0x4711
0x472222
```

Note that the first item returned is always the `BitString` before the first occurrence of the delimiter, even if it is empty.

A BitString is a list

If you treat a `BitString` object as a list whose elements are all either '`1`' or '`0`' then you won't go far wrong. Many operations can be performed using standard slice notation, although there are generally named functions to do the same jobs:

Using functions	Using slices
`s.truncatestart(bits)`	`del s[:bits]`
`s.truncateend(bits)`	`del s[-bits:]`
`s.slice(startbit, endbit)`	`s[startbit:endbit]`
`s.insert(bs, bitpos)`	`s[bitpos:bitpos] = bs`
`s.overwrite(bs, bitpos)`	`s[bitpos:bitpos+len(bs)] = bs`
`s.deletebits(bits, bitpos)`	`del s[bitpos:bitpos+bits]`
`s.deletebytes(bytes, bytepos)`	`del s[bytepos:bytepos+bytes:8]`
`s.append(bs)`	`s[len(s):len(s)] = bs`
`s.prepend(bs)`	`s[0:0] = bs`

Reading and Navigating

Reading

A common need is to parse a large `BitString` into smaller syntax elements. Functions for reading in bytes and bits are provided and will return new `BitString` objects. These new objects are top-level `BitString` objects and can be interpreted using properties as in the next example or could be read from to form a hierarchy of reads.

Every `BitString` has a property `bitpos` which is the current position from which reads occur. `bitpos` can range from zero (its value on construction) to the length of the `BitString`, a position from which all reads will fail as it is past the last bit.

This example does some simple parsing of the supplied MPEG-1 video stream.

```
s = BitString(filename='test/test.m1v')
start_code = s.readbytes(4).hex
width = s.readbits(12).uint
height = s.readbits(12).uint
s.advancebits(37)
flags = s.readbits(2)
constrained_parameters_flag = flags.readbit().uint
load_intra_quantiser_matrix = flags.readbit().uint
```

In addition to the `read` functions there are matching `peek` functions. These are identical to the `read` except that they do not advance the position in the `BitString`.

```
s = BitString('0x4732aa34')
if s.peekbyte() == '0x47':
    t = s.readbytes(2)          # t.hex == '0x4732'
else:
    s.find('0x47')
```

The full list of functions is `readbit()`, `readbits(n)`, `readbyte()`, `readbytes(n)`, `peekbit()`, `peekbits(n)`, `peekbyte()` and `peekbytes(n)`.

Seeking

The properties `bitpos` and `bytepos` are available for getting and setting the position, which is zero on creation of the `BitString`. There are also `advance`, `retreat` and `seek` functions that perform equivalent actions:

Using functions	Using properties
`advancebit()`	`bitpos += 1`
`advancebits(n)`	`bitpos += n`
`advancebyte()`	`bytepos += 1`
`advancebytes(n)`	`bytepos += n`
`retreatbit()`	`bitpos -= 1`
`retreatbits(n)`	`bitpos -= n`
`retreatbyte()`	`bytepos -= 1`
`retreatbytes(n)`	`bytepos -= n`
`seekbit(p)`	`bitpos = p`
`seekbyte(p)`	`bytepos = p`

For example:

```
>>> s = BitString('0x123456')
>>> s.bitpos
0
>>> s.bytepos += 2
>>> s.bitpos              # note bitpos verses bytepos
16
>>> s.advancebits(4)
>>> print s.read(4).bin    # the final nibble '0x6'
0b0110
```

Finding

To search for a sub-string use the `find` function. If the find succeeds it will set the position to the start of the next occurrence of the searched for string and return `True`, otherwise it will return `False`. By default the sub-string will only be found on byte boundaries; to allow it to be found at any position set `bytealigned=False`.

```
>>> s = BitString('0x00123400001234')
>>> found = s.find('0x1234')
>>> print found, s.bytepos
True 1
>>> found = s.find('0xff')
>>> print found, s.bytepos
False 1
```

Miscellany

Other Functions

`empty()`

Returns `True` if the BitString contains no data (i.e. has zero length). Otherwise returns `False`.

```
>>> a = BitString()
>>> print a.empty()
True
```

`bytealign()`

This function advances between zero and seven bits to make the `bitpos` a multiple of eight. It returns the number of bits advanced.

```
>>> a = BitString('0x11223344')
>>> a.bitpos = 1
>>> skipped = a.bytealign()
>>> print skipped, a.bitpos
7 8
>>> skipped = a.bytealign()
>>> print skipped, a.bitpos
0 8
```

`reversebits()`

This simply reverses all of the bits of the `BitString` in place.

```
>>> a = BitString('0b000001101')
>>> a.reversebits()
>>> a.bin
'0b101100000'
```

Special Methods

A few of the special methods have already been covered, for example `__add__` and `__iadd__` (the + and += operators) and `__getitem__` and `__setitem__` (reading and setting slices via `[]`). Here are the rest:

`__len__`

This implements the `len` function and returns the length of the `BitString` in bits. There's not much more to say really, except to emphasise that it is always in bits and never bytes.

```
>>> len(BitString('0x00'))
8
```

`__str__ , __repr__`

These get called when you try to print a `BitString`. As `BitString` objects have no preferred interpretation the form printed might not be what you want - if not then use the `hex`, `bin`, `int` etc. properties. The main use here is in interactive sessions when you just want a quick look at the `BitString`. The `__repr__` tries to give a code fragment which if evaluated would give an equal `BitString`.

The form used for the `BitString` is generally the one which gives it the shortest representation. If the resulting string is too long then it will be truncated with '...' - this prevents very long `BitString` objects from tying up your interactive session printing themselves.

```
>>> a = BitString('0b1111 111')
>>> print a
'0b1111111'
>>> a
BitString('0b1111111')
>>> a += '0b1'
>>> print a
0xff
>>> print a.bin
0b11111111
```

__eq__ , __ne__

The equality of two `BitString` objects is determined by their binary representations being equal. If you have a different criterion you wish to use then code it explicitly, for example `a.int == b.int` could be true even if `a == b` wasn't (as they could be different lengths).

Note that two `BitString` objects can have different offsets, but still be equal if their binary representations are equal.

```
>>> BitString('0b0010') == '0x2'
True
>>> BitString('0x2') != '0o2'
True
```

__hex__ , __oct__

You can if you wish use the built-in functions `hex()` and oct`()` instead of the `hex` and `oct` properties, although for consistency it is probably better to stick to using the properties. Note that octals are always prefixed by '`0o`' rather than just '`0`'. Note also that although a `bin()` built-in function was introduced in Python 2.6 there doesn't seem to be a corresponding __`bin`__ special function, for reasons that escape me, so you can't use `bin()` on a `BitString`.

```
>>> a = BitString('0o7777')
>>> a.oct
'0o7777'
>>> oct(a)
'0o7777'
>>> a.hex
'0xfff'
>>> hex(a)
'0xfff'
```

__invert__

To invert all the bits in a `BitString` use the ~ operator.

```
>>> a = BitString('0b0001100111')
>>> print a
0b0001100111
>>> print ~a
0b1110011000
>>> ~~a == a
True
```

__lshift__ , __rshift__ , __ilshift__ , __irshift__

Bitwise shifts can be achieved using <<, >>, <<= and >>=. Bits shifted off the left or right are replaced with zero bits. If you need special behaviour, such as keeping the sign of two's complement integers then do the shift on the property instead.

```
a = BitString('0b10011001')
b = a << 2
print b                       # 0b01100100
a >>= 2
print a                       # 0b00100110
```

__mul__ , __imul__ , __rmul__

Multiplication of a `BitString` by an integer means the same as it does for ordinary strings: concatenation of multiple copies of the `BitString`.

```
a = BitString('0b10')
b = a*10
print b                       # 0b10101010101010101010
a *= 2
print a                       # 0b1010
```

__copy__

This allows the `BitString` to be copied via the copy module.

```
import copy
a = BitString('0x4223fbddec2231')
b = copy.copy(a)
```

It's not terribly exciting, and isn't even the preferred method of making a copy. Using `b = BitString(a)` is my favourite, but `b = a[:]` may be more familiar to some.

__and__ , __or__ , __xor__

Bit-wise AND, OR and XOR are provided for `BitString` objects of equal length only (otherwise a `ValueError` is raised). The right-hand-side of expression can be a string to use in the `auto` initialiser.

```
a = BitString('0b00001111')
b = BitString('0b01010101')
print (a&b).bin               # 0b00000101
print (a|b).bin               # 0b01011111
print (a^b).bin               # 0b01010000
b &= '0x1f'
print b.bin                   # 0b00010101
```

Reference

One class is provided, `BitString`, which has the following public methods and properties.

`__add__ / __radd__`

```
s1 + s2
```

Concatenate two `BitString` and return the result. Either `s1` or `s2` can be a string to be used with the `auto` initialiser.

```
s = BitString(uint=34, length=8) + '0xff'
s2 = '0b101' + s
```

`advancebit`

```
s.advancebit()
```

Advances position by 1 bit. Equivalent to `s.bitpos += 1`.

`advancebits`

```
s.advancebits(bits)
```

Advances position by `bits` bits. Equivalent to `s.bitpos += bits`.

`advancebyte`

```
s.advancebyte()
```

Advances position by 8 bits. Equivalent to `s.bitpos += 8`. Unlike the alternative, `s.bytepos += 1`, `advancebyte` will not raise a `BitStringError` if the current position is not byte-aligned.

`advancebytes`

```
s.advancebytes(bytes)
```

Advances position by `8*bytes` bits. Equivalent to `s.bitpos += 8*bytes`. Unlike the alternative, `s.bytepos += bytes`, `advancebytes` will not raise a `BitStringError` if the current position is not byte-aligned.

`__and__ / __rand__`

```
s1 & s2
```

Returns the bit-wise AND between `s1` and `s2`, which must have the same length otherwise a `ValueError` is raised. Either `s1` or `s2` can be a string for the `auto` initialiser.

```
>>> print BitString('0x33') & '0x0f'
0x03
```

`append`

```
s.append(bs)
```

Join a `BitString` to the end of the current `BitString`. Returns `self`. `bs` can be either a `BitString` or a string for the `auto` initialiser.

```
s.append(BitString(hex='ffab'))
s.append('0xffab')
```

bin

`s.bin`

Read and write property for setting and getting the representation of the `BitString` as a binary string starting with '`0b`'. When used as a getter, the returned value is always calculated - the value is never cached. When used as a setter the length of the `BitString` will be adjusted to fit its new contents.

```
if s.bin == '0b001':
    s.bin = '0b1111'

s.bin += '1' # Equivalent to s.append('0b1')
```

bitpos

`s.bitpos`

Read and write property for setting and getting the current bit position in the `BitString`. Can be set to any value from `0` to `length`.

```
if s.bitpos < 100:
    s.bitpos += 10
```

bytealign

`s.bytealign()`

Aligns to the start of the next byte (so that `bitpos` is a multiple of 8) and returns the number of bits skipped. If the current position is already byte aligned then it is unchanged.

bytepos

`s.bytepos`

Read and write property for setting and getting the current byte position in the `BitString`. When used as a getter will raise a `BitStringError` if the current position in not byte aligned.

__contains__

`bs in s`

Returns `True` if `bs` can be found in `s`, otherwise returns `False`. Equivalent to using `find` with `bytealigned=False`, except that `bitpos` will not be changed.

```
>>> '0b11' in BitString('0x06')
True
>>> '0b111' in BitString('0x06')
False
```

__copy__

`s2 = copy.copy(s1)`

This allows the `copy` module to correctly copy `BitString` objects. Other equivalent methods are to initialise a new `BitString` with the old one or to take a complete slice.

```
>>> import copy
>>> s = BitString('0o775')
>>> s_copy1 = copy.copy(s)
>>> s_copy2 = BitString(s)
>>> s_copy3 = s[:]
>>> s == s_copy1 == s_copy2 == s_copy3
True
```

data

s.data

Read and write property for setting and getting the underlying byte data that contains the `BitString`. Set using an ordinary Python string - the length will be adjusted to contain the data. When used as a getter the `BitString` will be padded with between zero and seven '`0`' bits to make it byte aligned.

deletebits

s.deletebits(bits, bitpos=None)

Removes bits bits from the `BitString` at position `bitpos` and returns `self`. If `bitpos` is not specified then the current position is used.

deletebytes

s.deletebytes(bytes, bytepos=None)

Removes `bytes` bytes from the `BitString` at position `bytepos` and returns `self`. If `bytepos` is not specified then the current position is used, provided it is byte aligned, otherwise `BitStringError` is raised.

__delitem__

del s[a:b:c]

empty

s.empty()

Returns `True` if the `BitString` is empty, i.e. has `length==0`. Otherwise returns `False`.

__eq__

s1 == s2

Compares two `BitString` objects for equality, returning `True` if they have the same binary representation, otherwise returning `False`.

```
>>> BitString('0o7777') == '0xfff'
True
>>> BitString(uint=13, length=8) == BitString(uint=13, length=10)
False
```

find

s.find(bs, bytealigned=True, startbit=None, endbit=None)

Searches for `bs` (a `BitString` or string to initialise via `auto`) in the current `BitString` and returns `True` if found. If `bytealigned` is `True` then it will look for `bs` only at byte aligned positions (which is generally much faster than searching for it in every possible bit position). `startbit` and `endbit` give the search range and default to `0` and `self.length` respectively.

findall

`s.findall(bs, bytealigned=True, startbit=None, endbit=None)`

Searches for all occurences of `bs` (even overlapping ones) and returns a generator their bit positions. If `bytealigned` is `True` then `bs` will only be looked for at byte aligned positions. `startbit` and `endbit` optionally define a slice and default to `0` and `self.length` respectively.

__getitem__

`s[a:b:c]`

__hex__

`hex(s)`

Returns the hexadecimal representation of the `BitString`, i.e. a string starting with '`0x`'. Equivalent to using the `hex` property, and so will raise a `ValueError` if the BitString is not a multiple of four bits long.

hex

`s.hex`

Read and write property for setting and getting the hexadecimal representation of the `BitString`. When used as a getter the value will be preceded by '`0x`', which is optional when setting the value. If the `BitString` is not a multiple of four bits long then getting its `hex` value will raise a `ValueError`.

```
>>> s = BitString(bin='1111 0000 1111')
>>> s.hex
'0xf0f'
>>> s.hex = 'abcdef'
>>> s.hex
'0xabcdef'
```

__iadd__

`s1 += s2`

Append a `BitString` to the current `BitString` and return the result. `s2` can be a string to be used with the `auto` initialiser.

```
>>> s = BitString(ue=423)
>>> s += BitString(ue=12)
>>> s.readue()
423
>>> s.readue()
12
```

__ilshift__

`s <<= n`

__imul__

`s *= n`

__init__

```
s = BitString(auto=None, length=None, offset=0, data=None, filename=None,
hex=None, bin=None, oct=None, uint=None, ue=None, se=None)
```

insert

```
s.insert(bs, bitpos=None)
```

Inserts `bs` (a `BitString` or string to initialise via `auto`) at `bitpos` and returns `self`. `bitpos` defaults to the current position.

int

```
s.int
```

__invert__

```
~s
```

__irshift__

```
s >>= n
```

__len__

```
len(s)
```

Returns the length of the `BitString` in bits.

length

```
s.length
```

Read-only property that gives the length of the `BitString` in bits.

__lshift__

```
s << n
```

__mul__ / __rmul__

```
s * n / n * s
```

__ne__

```
s1 != s2
```

__oct__

```
oct(s)
```

oct

```
s.oct
```

Read and write property for setting and getting the octal representation of the `BitString`. When used as a getter the value will be preceded by '`0o`', which is optional when setting the value. If the `BitString` is not a multiple of three bits long then getting its `oct` value will raise a `ValueError`.

`__or__ / __ror__`

`s1 | s2`

Returns the bit-wise OR between `s1` and `s2`, which must have the same length otherwise a `ValueError` is raised. Either `s1` or `s2` can be a string for the `auto` initialiser.

```
>>> print BitString('0x33') | '0x0f'
0x3f
```

`overwrite`

`s.overwrite(bs, bitpos=None)`

Replaces the contents of the current `BitString` with bs (a `BitString` or string to initialise via `auto`) at `bitpos` and returns `self`. `bitpos` defaults to the current position.

`peekbit`

`s.peekbit()`

Returns the next bit in the current `BitString` as a new `BitString` but does not advance the position.

`peekbits`

`s.peekbits(bits)`

Returns the next `bits` bits of the current `BitString` as a new `BitString` but does not advance the position.

`peekbyte`

`s.peekbyte()`

Returns the next byte of the current `BitString` as a new `BitString` but does not advance the position.

`peekbytes`

`s.peekbytes(bytes)`

Returns the next `bytes` bytes of the current `BitString` as a new `BitString` but does not advance the position.

`prepend`

`s.prepend(bs)`

Inserts `bs` (which can be either a `BitString` or a string for the `auto` initialiser) at the beginning of the current `BitString`. Returns `self`.

`readbit`

`s.readbit()`

Returns the next bit of the current `BitString` as a new `BitString` and advances the position.

readbits

```
s.readbits(bits)
```

Returns the next `bits` bits of the current `BitString` as a new `BitString` and advances the position.

readbyte

```
s.readbyte()
```

Returns the next byte of the current `BitString` as a new `BitString` and advances the position.

readbytes

```
s.readbytes(bytes)
```

Returns the next `bytes` bytes of the current `BitString` as a new `BitString` and advances the position.

readse

```
s.readse()
```

readue

```
s.readue()
```

replace

```
s.replace(old, new, bytealigned=True, startbit=None, endbit=None, count=None)
```

__repr__

```
repr(s)
```

A representation of the `BitString` that could be used to create it (which will often not be the form used to create it). If the result is too long then it will be truncated with '...' and the length of the whole `BitString` will be given.

```
>>> BitString('0b11100011')
BitString('0xe3')
```

retreatbit

```
s.retreatbit()
```

Retreats position by 1 bit. Equivalent to `bitpos -= 1`.

retreatbits

```
s.retreatbits(bits)
```

Retreats position by `bits` bits. Equivalent to `bitpos -= bits`.

retreatbyte

```
s.retreatbyte()
```

Retreats position by 8 bits. Equivalent to `bitpos -= 8`. Unlike the alternative, `bytepos -= 1`, `retreatbyte` will not raise a `BitStringError` if the current position is not byte-aligned.

retreatbytes

`s.retreatbytes(bytes)`

Retreats position by `bytes*8` bits. Equivalent to `bitpos -= 8*bytes`. Unlike the alternative, `bytepos -= bytes`, `retreatbytes` will not raise a `BitStringError` if the current position is not byte-aligned.

reversebits

`s.reversebits(startbit=None, endbit=None)`

Reverses all of the bits in the `BitString` in-place and returns self. `startbit` and `endbit` give the range and default to `0` and `self.length` respectively.

```
>>> BitString('0b00010111').reversebits().bin
'0b11101000'
```

rfind

`s.rfind(bs, bytealigned=True, startbit=None, endbit=None)`

Searches backwards for `bs` (a `BitString` or string to initialise via `auto`) in the current `BitString` and returns `True` if found. If `bytealigned` is `True` then it will look for `bs` only at byte aligned positions. `startbit` and `endbit` give the search range and default to `0` and `self.length` respectively. Note that as it's a reverse search it will start at `endbit` and finish at `startbit`.

__rshift__

`s >> n`

se

`s.se`

seekbit

`s.seekbit(bitpos)`

Moves the current position to `bitpos`. Equivalent to `s.bitpos = bitpos`.

seekbyte

`s.seekbyte(bytepos)`

Moves the current position to `bytepos`. Equivalent to `s.bytepos = bytepos`.

__setitem__

`s1[a:b:c] = s2`

slice

`s.slice(startbit, endbit)`

split

```
s.split(delimiter, bytealigned=True, maxsplit=None)
```

`__str__`

```
print s
```

`tellbit`

```
s.tellbit()
```

Returns the current bit position. Equivalent to using the `bitpos` property as a getter.

`tellbyte`

```
s.tellbyte()
```

Returns the current byte position. Equivalent to using the `bytepos` property as a getter, and will raise a `BitStringError` is the `BitString` is not byte aligined.

`truncateend`

```
s.truncateend(bits)
```

Remove the last `bits` bits from the end of the `BitString`. Returns `self`.

`truncatestart`

```
s.truncatestart(bits)
```

Remove the first `bits` bits from the start of the `BitString`. Returns self.

`ue`

```
s.ue
```

`uint`

```
s.uint
```

`__xor__ / __rxor__`

```
s1 ^ s2
```

Returns the bit-wise XOR between `s1` and `s2`, which must have the same length otherwise a `ValueError` is raised. Either `s1` or `s2` can be a string for the `auto` initialiser.

```
>>> print BitString('0x33') ^ '0x0f'
0x3c
```

Appendix A: Exponential-Golomb codes

As this type of representation of integers isn't as well known as the standard base-2 representation I thought that a short explanation of them might be welcome. This section can be safely skipped if you're not interested.

Exponential-Golomb codes represent integers using bit patterns that get longer for larger numbers. For unsigned and signed numbers (the `BitString` properties `ue` and `se` respectively) the patterns start like this:

Unsigned	Signed	Bit pattern
0	0	`1`
1	1	`10`
2	-1	`11`
3	2	`100`
4	-2	`101`
5	3	`110`
6	-3	`111`
7	4	`1000`
8	-4	`1001`
9	5	`1010`
10	-5	`1011`
11	6	`1100`
...	...	`...`

They consist of a sequence of `n` `'0'` bits, followed by a `'1'` bit, followed by `n` more bits. The bits after the first `'1'` bit count upwards as ordinary base-2 binary numbers until they run out of space and an extra `'0'` bit needs to get included at the start.

The advantage of this method of representing integers over many other methods is that it can be quite efficient at representing small numbers without imposing a limit on the maximum number that can be represented.

Exercise: Using the table above decode this sequence of unsigned Exponential Golomb codes:

```
001001101101101011000100100101
```

The answer is that it decodes to 3, 0, 0, 2, 2, 1, 0, 0, 8, 4. Note how you don’t need to know how many bits are used for each code in advance - there’s only one way to decode it. To create this bitstring you could have written something like:

```
a = bitstring.join([BitString(ue=i) for i in [3,0,0,2,2,1,0,0,8,4]])
```

and to read it back:

```
while a.bitpos != a.length:
    print a.readue()
```

The notation `ue` and `se` for the exponential-Golomb code properties comes from the H.264 video standard, which uses these types of code a lot. The particular way that the signed integers are represented might be peculiar to this standard as I haven't seen it elsewhere (and an obvious alternative is minus the one given here), but the unsigned mapping seems to be universal.