01100010011010010111010001110011 0111

bitstring

01000111001001101001011011100110 0111

A Python module to help you manage your bits

by Scott Griffiths

version 0.5.2

11 September 2009

python-bitstring.googlecode.com

Contents

1. Introduction

bitstring is a pure Python module designed to help make the creation and analysis of binary data as painless as possible.

While it is certainly possible to manipulate binary data in Python, for example using the `struct` and `array` modules, it can be quite fiddly, especially if you are not dealing only with whole-byte data.

The bitstring module provides a single class, `BitString`, instances of which can be constructed from integers, hex, octal, binary, strings or files, but they all just represent a string of binary digits. They can be sliced, joined, reversed, inserted into, overwritten, packed, unpacked etc. with simple functions or slice notation. They can also be read from, searched in, and navigated in, similar to a file or stream.

`BitString` objects are designed to be as lightweight as possible and can be considered to be just a list of binary digits. They are however stored very efficiently - although there are a variety of ways of creating and viewing the binary data, the `BitString` itself just stores the byte 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`, and an extensive set of functions is supplied for modifying, navigating and analysing the binary data.

A complete reference for the module is given in Appendix A, while the rest of this manual acts more like a tutorial or guided tour. Below is just a few examples to whet your appetite; everything here will be covered in greater detail in the rest of this manual.

```
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')                 # 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
```

1.1. Getting Started

First download the latest release for either Python 2.4 / 2.5 / 2.6 or 3.0 / 3.1 (see the Downloads tab on the project's homepage[1]). If you're using Windows and just want to install for your default Python installation then get the `.exe` and run it to install - you'll need to download this manual separately, but I guess you must have worked that out.

Otherwise you'll need to download 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/test.m1v` : An example file (MPEG-1 video) for testing purposes.

`test/smalltestfile` : Another small file for testing.

`bitstring_manual.pdf` : This document

To install, run

```
python setup.py install
```
[2]

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 300) 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 and 3.1) or something unexpected has happened - in which case please tell me about it.

2. Creation and Interpretation

You can create `BitString` objects in a variety of ways. Internally, `BitString` objects are stored as byte arrays (in particular an `array` module byte array). 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, an offset to the first used bit in the byte array plus a bit position in the `BitString`, used for reading etc. 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.

1 http://python-bitstring.googlecode.com

2 You may need to use 'sudo' or similar depending on your permissions on your system.

2.1. 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 specially formatted string, a list or tuple, a file object or another `BitString`.

`data` : A Python string, for example read from a binary file.

`hex`, `oct`, `bin`: Hexadecimal, octal or binary strings.

`int`, `uint`: Signed or unsigned bit-wise big-endian binary integers.

`intle, uintle`: Signed or unsigned byte-wise little-endian binary integers.

`intbe, uintbe`: Signed or unsigned byte-wise big-endian binary integers.

`intne, uintne`: Signed or unsigned byte-wise native-endian binary integers.

`se`, `ue` : Signed or unsigned exponential-Golomb coded integers.

`filename` : Directly from a file, without reading into memory.

From a hexadecimal string

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

The initial `0x` or `0X` is optional, as is a `length` parameter, which can be used to truncate bits from 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 instead. As it is the first parameter in `__init__` 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` (here it is used to remove the final `'0'`) and whitespace will be ignored.

As with `hex`, the `auto` initialiser will work if the binary string is 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` (a zero and lower-case 'o') 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 if the octal string is 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` respectively) 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.

The `auto` initialise can be used by giving a colon and the length in bits immediately after the `int` or `uint` token, followed by an equals sign then the value:

```
>>> e = BitString('uint:12=45')
>>> f = BitString('int:7=-1')
```

The plain `int` and `uint` initialisers are bit-wise big-endian. That is to say that the most significant bit comes first and the least significant bit comes last, so the unsigned number one will have a '`1`' as its final bit with all other bits set to '`0`'. These can be any number of bits long. For whole-byte `BitString` objects there are more options available with different endiannesses.

Big and little-endian integers

```
>>> big_endian = BitString(uintbe=1, length=16)
>>> little_endian = BitString(uintle=1, length=16)
>>> native_endian = BitString(uintne=1, length=16)
```

There are unsigned and signed versions of three additional 'endian' types. The unsigned versions are used above to create three `BitString` objects.

The first of these, `big_endian`, is equivalent to just using the plain bit-wise big-endian `uint` initialiser, except that all `intbe` or `uintbe` interpretations must be of whole-byte `BitString` objects, otherwise a `ValueError` is raised.

The second, `little_endian`, is interpreted as least significant byte first, i.e. it is a byte reversal of `big_endian`. So we have:

```
>>> big_endian.hex
'0x0001'
>>> little_endian.hex
'0x0100'
```

Finally we have `native_endian`, which will equal either `big_endian` or `little_endian`, depending on whether you are running on a big or little-endian machine (if you really need to check then use "`import sys; sys.byteorder`").

Exponential-Golomb codes

Initialisation with integers represented by exponential-Golomb codes is also possible. `ue` is an unsigned code while `se` is a signed code:

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

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 are in good company, but they are quite interesting, so I've included an appendix on them (see Appendix B).

The `auto` initialiser may also be used by giving an equals sign and the value immediately after a `ue` or `se` token:

```
>>> g = BitString('ue=12')
>>> h = BitString('se=-402')
```

You may wonder why you would bother with `auto` in this case as the syntax is slightly longer. Hopefully all will become clear in the next section.

From raw data

For most initialisers you can 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 or 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 also optional and is used to truncate 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")
```

This 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. If only a portion of the file is needed then the `offset` and `length` parameters (specified in bits) can be used.

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.

It's also possible to use the `auto` initialiser for file objects. It's as simple as:

```
f = open('my2GBfile', 'rb')
p = BitString(f)
```

2.2. The auto initialiser

The `auto` parameter is the first parameter in the `__init__` function and so the `auto=` can be omitted when using it. It accepts either a string, a list or tuple, another `BitString` or a file object. Strings starting with `0x` or `hex` are interpreted as hexadecimal, `0o` or `oct` implies octal, and strings starting with `0b` or `bin` are interpreted as binary. You can also initialise with the various integer initialisers as described above. If given another `BitString` it will create a copy of it, lists and tuples are interpreted as boolean arrays and file objects acts a source of binary data.

```
>>> fromhex = BitString('0x01ffc9')
>>> frombin = BitString('0b01')
>>> fromoct = BitString('0o7550')
>>> fromint = BitString('int:32=10')
>>> acopy   = BitString(fromoct)
>>> fromlist = BitString([True, False, False])
>>> f = open('somefile', 'rb')
>>> fromfile = BitString(f)
```

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
>>> frombin[::-1] + '0b0' == fromlist
True
```

Note how in the final examples above only one half of the == needs to be a `BitString`, the other half gets '`auto`' initialised before the comparison is made. This is in common with many other functions and operators.

You can also chain together string initialisers with commas, which causes the individual `BitString` object to be concatenated.

```
>>> s = BitString('0x12, 0b1, uint:5=2, ue=5, se=-1, se=4')
>>> s.find('uint:5=2, ue=5')
True
>>> s.insert('0o332, 0b11, int:23=300', 4)
BitString('0o0555300001130242146l0')
```

Again, note how the format used in the `auto` initialiser can be used in many other places where a `BitString` is needed.

2.3. Packing

Another method of creating `BitString` objects is to use the `pack` function. This takes a format specifier which is a string with comma separated tokens, and a number of items to pack according to it. It's signature is `bitstring.pack(format, *values, **kwargs)`.

For example using just the `*values` arguments we can say:

```
s = bitstring.pack('hex:32, uint:12, uint:12', '0x000001b3', 352, 288)
```

which is equivalent to initialising as:

```
s = BitString('0x0000001b3, uint:12=352, uint:12=288')
```

The advantage of the `pack` method is if you want to write more general code for creation.

```
def foo(a, b, c, d):
    return bitstring.pack('uint:8, 0b110, int:6, bin, bits', a, b, c, d)

s1 = foo(12, 5, '0b00000', '')
s2 = foo(101, 3, '0b11011', s1)
```

Note how you can use some tokens without sizes (such as `bin` and `bits` in the above example), and use values of any length to fill them. If the size had been specified then a `ValueError` would be raised if the parameter given was the wrong length. Note also how `BitString` literals can be used (the `'0b110'` in the `BitString` returned by `foo`) and these don't consume any of the itmes in `*values`.

You can also include keyword, value pairs (or an equivalent dictionary) as the final parameter(s). The values are then packed according to the positions of the keywords in the format string. This is most easily explained with some examples. Firstly the format string needs to contain parameter names:

```
format = 'hex:32=start_code, uint:12=width, uint:12=height'
```

Then we can make a dictionary with these parameters as keys and pass it to `pack`:

```
d = {'start_code': '0x000001b3', 'width': 352, 'height': 288}
s = bitstring.pack(format, **d)
```

Another method is to pass the same information as keywords at the end of `pack`'s parameter list:

```
s = bitstring.pack(format, width=352, height=288, start_code='0x000001b3')
```

The tokens in the format string that you must provide values for are:

`int:`*n*	→	*n* bits as an unsigned integer.
`uint:`*n*	→	*n* bits as a signed integer.
`intbe:`*n*	→	*n* bits as a big-endian whole byte unsigned integer.
`uintbe:`*n*	→	*n* bits as a big-endian whole byte signed integer.
`intle:`*n*	→	*n* bits as a little-endian whole byte unsigned integer.
`uintle:`*n*	→	*n* bits as a little-endian whole byte signed integer.
`intne:`*n*	→	*n* bits as a native-endian whole byte unsigned integer.
`uintne:`*n*	→	*n* bits as a native-endian whole byte signed integer.
`hex:`*n*	→	*n* bits as a hexadecimal string.
`oct:`*n*	→	*n* bits as an octal string.
`bin:`*n*	→	*n* bits as a binary string.
`bits:`*n*	→	*n* bits as a new `BitString`.
`bytes:`*n*	→	*n* bytes as a new `BitString`.

`ue`	→	an unsigned integer as an exponential-Golomb code.
`se`	→	a signed integer as an exponential-Golomb code.

and you can also include constant `BitString` tokens constructed from any of the following:

`0b...`	→	binary literal.
`0o...`	→	octal literal.
`0x...`	→	hexadecimal literal.
`int:`*n*`=`*m*	→	signed integer *m* in *n* bits.
`uint:`*n*`=`*m*	→	unsigned integer *m* in *n* bits.
`intbe:`*n*`=`*m*	→	big-endian whole byte signed integer *m* in *n* bits.
`uintbe:`*n*`=`*m*	→	big-endian whole byte unsigned integer *m* in *n* bits.
`intle:`*n*`=`*m*	→	little-endian whole byte signed integer *m* in *n* bits.
`uintle:`*n*`=`*m*	→	little-endian whole byte unsigned integer *m* in *n* bits.
`intne:`*n*`=`*m*	→	native-endian whole byte signed integer *m* in *n* bits.
`uintne:`*n*`=`*m*	→	native-endian whole byte unsigned integer *m* in *n* bits.
`ue=`*m*	→	exponential-Golomb code for unsigned integer *m*.
`se=`*m*	→	exponential-Golomb code for signed integer *m*.

You can also use a keyword for the length specifier in the token, for example

```
s = bitstring.pack('int:n=-1', n=100)
```

And finally it is also possible just to use a keyword as a token:

```
s = bitstring.pack('hello, world', world='0x123', hello='0b110')
```

As you would expect, there is also an `unpack` function that takes a `BitString` and unpacks it according to a very similar format string. This is covered later in more detail, but a quick example is:

```
>>> s = bitstring.pack('ue, oct:3, hex:8, uint:14', 3, '0o7', '0xff', 90)
>>> s.unpack('ue, oct:3, hex:8, uint:14')
[3, '0o7', '0xff', 90]
```

Compact format strings

Another option when using `pack` is to use a format specifier similar to those used in the `struct` and `array` modules. These consist of a character to give the endianness, followed by more single characters to give the format.

The endianness character must start the format string and unlike in the `struct` module it is not optional:

`>`	→	Big-endian
`<`	→	Little-endian
`@`	→	Native-endian

For 'network' endianness use '`>`' as network and big-endian are equivalent. This is followed by at least one of these format characters:

`b`	→	8 bit signed integer
`B`	→	8 bit unsigned integer
`h`	→	16 bit signed integer
`H`	→	16 bit unsigned integer
`l`	→	32 bit signed integer
`L`	→	32 bit unsigned integer

`q`	→	64 bit signed integer
`Q`	→	64 bit unsigned integer

The exact type is determined by combining the endianness character with the format character, but rather than give an exhaustive list a single example should explain:

`>h`	→	Big-endian 16 bit signed integer	→	`intbe:16`
`<h`	→	Little-endian 16 bit signed integer	→	`intle:16`
`@h`	→	Native-endian 16 bit signed integer	→	`intne:16`

As you can see all three are signed integers in 16 bits, the only difference is the endianness. The native-endian `@h` will equal the big-endian `>h` on big-endian systems, and equal the little-endian `<h` on little-endian systems.[3]

An example:

```
s = bitstring.pack('>qqqq', 10, 11, 12, 13)
```

is equivalent to

```
s = bitstring.pack('intbe:64, intbe:64, intbe:64, intbe:64', 10, 11, 12, 13)
```

Just as in the `struct` module you can also give a multiplicative factor before the format character, so the previous example could be written even more concisely as

```
s = bitstring.pack('>4q', 10, 11, 12, 13)
```

You can of course combine these format strings with other initialisers, even mixing endianness (although I'm not sure why you'd want to):

```
s = bitstring.pack('>6h3b, 0b1, <9L', *range(18))
```

This rather contrived example takes the number 0 to 17 and packs the first 6 as signed big-endian 2-byte integers, the next 3 as single bytes, then inserts a single `1` bit, before packing the remaining 9 as little-endian 4-byte unsigned integers.

2.4. 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 private functions 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.

Note that these properties can potentially be very expensive in terms of both computation and memory requirements. For example if you have initialised a `BitString` from a 10 GB file object and ask for its binary string representation then that string will be around 80 GB in size!

For the properties described below we will use these:

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

3 For the single byte codes `b` and `B` the endianness doesn't make any difference but you still need to specify one so that the code can be parsed correctly.

bin

The most fundamental interpretation is perhaps as a binary string (a 'bitstring'). 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 the zeros are 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 bits 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.

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

oct

For an octal interpretation use the `oct` property. 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'
>>> b.oct
'0o7'
>>> (b + '0b0').oct
ValueError: Cannot convert to octal unambiguously - not multiple of 3 bits.
```

uint / uintbe / uintle / uintne

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

```
>>> a.uint
283
>>> b.uint
7
```

For byte-wise big-endian, little-endian and native-endian interpretations use `uintbe`, `uintle` and `uintne` respectively. These will raise a `ValueError` if the `BitString` is not a whole number of bytes long.

```
>>> s = BitString('0x000001')
>>> s.uint     # bit-wise big-endian
1
>>> s.uintbe   # byte-wise big-endian
1
>>> s.uintle   # byte-wise little-endian
65536
>>> s.uintne   # byte-wise native-endian (will equal 1 on big-endian platform!)
65536
```

int / intbe / intle / intne

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
```

For byte-wise big, little and native endian signed integer interpretations use `intbe`, `intle` and `intne` respectively. These work in the same manner as their unsigned counterparts described above.

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 instead wish to read the next bits in the stream and interpret them as a code use the `read` function with a '`ue`' format string. See Appendix B for a short explanation of this type of integer representation.

```
>>> s = BitString(ue=12)
>>> s.bin
'0b0001101'
>>> s.append(BitString(ue=3))
BitString('0x1a4')
>>> print s.read('ue, ue'))
[12, 3]
```

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 B for more information.

```
>>> s = BitString(‘0x164b’)
>>> s.se
BitStringError: BitString is not a single exponential-Golomb code.
>>> while s.bitpos < s.length:
...     print s.read('se')
-5
2
0
-1
```

3. Slicing, Dicing and Splicing

Manipulating binary data can be a bit of a challenge in Python. One of its strengths is that you don't have to worry about the low level data, but this can make life difficult when what you care about is precisely the thing that is safely hidden by high level abstractions. In this section some more methods are described that treat data as a series of bits, rather than bytes.

3.1. Slicing

Slicing can be done in couple of ways. The `slice` function takes three arguments: the first bit position you want, one past the last bit position you want and a multiplicative factor which defaults to 1. 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
```

Stepping in slices

The step parameter (also known as the stride) can be used in slices. Its use is rather non-standard as it effectively gives a multiplicative factor to apply to the start and stop parameters, rather than skipping over bits.

For example this makes it much more convenient if you want to give slices in terms of bytes instead of bits. Instead of writing `s[a*8:b*8]` you can use `s[a:b:8]`.

When using a step, the `BitString` is effectively truncated to a multiple of the step, so `s[::8]` is equal to `s` if `s` is an integer number of bytes, otherwise it is truncated by up to 7 bits. This means that, for example, the final seven complete 16-bit words could be written as `s[-7::16]`.

```
>>> a = BitString('0x470000125e')
>>> print a[0:4:8]                  # The first four bytes
0x47000012
>>> print a[-3::4]                  # The final three nibbles
0x25e
```

Negative slices are also allowed, and should do what you'd expect. So for example `s[::-1]` returns a bit-reversed copy of `s` (which is similar to `s.reversebits()`, which does the same operation on `s` in-place). As another example, to get the first 10 bytes in reverse byte order you could use `s_bytereversed = s[0:10:-8]`.

```
>>> print a[:-5:-4]                 # Final five nibbles reversed
0xe5210
>>> print a[::-8]                   # The whole BitString byte reversed
0x5e12000047
```

3.2. 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'[4]
```

If you want to join a large number of `BitString` objects then the function `join` can be used to improve efficiency and readability. It works like the ordinary string `join` function in that it uses the `BitString` that it is called on as a separator when joining the list of `BitString` objects it is given. If you don't want a separator then it can be called on an empty `BitString`.

```
bslist = [BitString(uint=n, length=12) for n in xrange(1000)]
s = BitString('0b1111').join(bslist)
```

4 You could also just create it in one go without joining using BitString('0b000, 0xf')

3.3. Truncating, inserting, deleting and overwriting

truncatestart / truncateend

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 if a slice is used a new `BitString` is returned and the `BitString` being operated on remains unchanged.

insert

`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')
>>> a.insert('0xffff', 16)
>>> a.hex
'0x0011ffff2233'
```

overwrite

`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 / deletebytes

`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'
```

3.4. The BitString as 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. The table below gives some of the equivalent ways of using functions and the standard slice notation.

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

3.5. Splitting

`split`

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', bytealigned=True):
...     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.

`cut`

If you just want to split into equal parts then use the `cut` function. This takes a number of bits as its first argument and returns a generator for chunks of that size.

```
>>> a = BitString('0x47001243')
>>> for byte in a.cut(8):
...     print byte.hex
0x47
0x00
0x12
0x43
```

4. Reading, Unpacking and Navigating

4.1. Reading and unpacking

A common need is to parse a large `BitString` into smaller parts. Functions for reading in the `BitString` as if it were a file or stream 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 themselves 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. The property `bytepos` is also available, and is useful if you are only dealing with byte data and don't want to always have to divide the bit position by eight. Note that if you try to use `bytepos` and the `BitString` isn't byte aligned (i.e. `bitpos` isn't a multiple of 8) then a `BitStringError` exception will be raised.

`readbits / readbytes`

For simple reading of a number of bits you can use `readbits` or `readbytes`. The following example does some simple parsing of an MPEG-1 video stream[5].

```
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
```

`read`

As well as the `readbit` / `readbyte` functions there is also a plain `read` function. This takes a format string similar to that used in the `auto` initialiser. If only one token is provided then a single value is returned, otherwise a list of values is returned. The format string consists of comma separated tokens that describe how to interpret the next bits in the `BitString`. The tokens are:

`int:`*n*	→	*n* bits as an unsigned integer.
`uint:`*n*	→	*n* bits as a signed integer.
`intbe:`*n*	→	*n* bits as a byte-wise big-endian unsigned integer.
`uintbe:`*n*	→	*n* bits as a byte-wise big-endian signed integer.
`intle:`*n*	→	*n* bits as a byte-wise little-endian unsigned integer.
`uintle:`*n*	→	*n* bits as a byte-wise little-endian signed integer.
`intne:`*n*	→	*n* bits as a byte-wise native-endian unsigned integer.
`uintne:`*n*	→	*n* bits as a byte-wise native-endian signed integer.
`hex:`*n*	→	*n* bits as a hexadecimal string.
`oct:`*n*	→	*n* bits as an octal string.
`bin:`*n*	→	*n* bits as a binary string.
`bits:`*n*	→	*n* bits as a new `BitString`.
`bytes:`*n*	→	*n* bytes as a new `BitString`.
`ue`	→	next bits as an unsigned exponential-Golomb code.
`se`	→	next bits as a signed exponential-Golomb code.

So in the previous example we could have combined three reads as:

```
start_code, width, height = s.read('hex:32, uint:12, uint:12')
```

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` to after the read elements.

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

5 The stream is provided in the test directory if you downloaded the source archive.

The complete list of read and peek functions is `read(*format)`, `readbit()`, `readbits(n)`, `readbyte()`, `readbytes(n)`, `peek(*format)`, `peekbit()`, `peekbits(n)`, `peekbyte()` and `peekbytes(n)`.

`unpack`

The `unpack` function works in a very similar way to `read`. The major difference is that it interprets the whole `BitString` from the start, and takes no account of the current `bitpos`. It's a natural complement of the `pack` function.

```
s = pack('uint:10, hex, int:13, 0b11', 130, '3d', -23)
a, b, c, d = s.unpack('uint:10, hex, int:13, bin:2')
```

4.2. 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. Whether you use the functions or the properties is purely a personal choice.

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`
`tellbit()`	→	`bitpos`
`tellbyte()`	→	`bytepos`

Note that you can only use `bytepos` or the `advance/retreatbyte(s)` functions if the position is byte aligned, i.e. the bit position is a multiple of 8. Otherwise a `BitStringError` exception is raised.

For example:

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

4.3. Finding and replacing

`find / rfind`

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 be found at any bit position - to allow it to only be found on byte boundaries set `bytealigned=True`.

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

`rfind` does much the same as `find`, except that it will find the last occurrence, rather than the first.

```
>>> t = BitString('0x0f231443e8')
>>> found = t.rfind('0xf')          # Search all bit positions in reverse
>>> print found, t.bitpos
True 31                             # Found within the 0x3e near the end
```

For all of these finding functions you can optionally specify a `startbit` and / or `endbit` to narrow the search range. Note though that because it's searching backwards `rfind` will start at `endbit` and end at `startbit` (so you always need `startbit < endbit`).

findall

To find all occurrences of a `BitString` inside another (even overlapping ones), use `findall`. This returns a generator for the bit positions of the found strings.

```
>>> r = BitString('0b011101011001')
>>> ones = r.findall('0b1')
>>> print list(ones)
[1, 2, 3, 5, 7, 8, 11]
```

replace

To replace all occurrences of one `BitString` with another use `replace`. The replacements are done in-place, and the number of replacements made is returned.

```
>>> s = BitString('0b110000110110')
>>> s.replace('0b110', '0b1111')
3            # The number of replacements made
>>> s.bin
'0b111100011111111'
```

5. Miscellany

5.1. 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 the bits of the `BitString` in place and returns `self`. You can optionally specify a range of bits to reverse.

```
>>> a = BitString('0b000001101')
>>> a.reversebits().bin
'0b101100000'
>>> a.reversebits(0, 4).bin
'0b110100000'
```

`reversebytes`

This reverses the bytes of the `BitString` in place and returns `self`. You can optionally specify a range of bits to reverse. If the length to reverse isn't a multiple of 8 then a `BitStringError` is raised.

```
>>> a = BitString('0x123456')
>>> a.reversebytes().hex
'0x563412'
>>> a.reversebits(0, 16).hex
'0x345612'
```

`tostring`

Returns the data contained in the `BitString` as a Python string. If the `BitString` isn't a whole number of bytes long then it will be made so by appending up to seven zero bits.

5.2. 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. It's recommended that you use the `length` property instead of the function as a limitation of Python means that the function will raise an `OverflowError` if the `BitString` has more than `sys.maxsize` elements (that's typically 256MB of data). 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).

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

`__invert__`

To invert all the bits in a `BitString` use the ~ operator, which returns a bit-inverted copy.

```
>>> 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')*8
>>> print a.bin
0b1010101010101010
```

`__copy__`

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

```
>>> import copy
>>> a = BitString('0x4223fbddec2231')
>>> b = copy.copy(a)
>>> b == a
True
>>> b is a
False
```

It's not terribly exciting, and isn't even the preferred method of making a copy. Using `b = BitString(a)` is another option, 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).

```
>>> 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
```

6. Examples

6.1. Creation

There are lots of ways of creating new `BitString` objects. The most flexible is via the `auto` parameter which is used in this example.

```
# Multiple parts can be joined with a single expression...
s = BitString('0x000001b3, uint:12=352, uint:12=288, 0x1, 0x3')

# and extended just as easily
s += 'uint:18=48000, 0b1, uint:10=4000, 0b100'

# To covert to an ordinary string use the data property
open('video.m2v', 'wb').write(s.data)

# The information can be read back with a similar syntax
start_code, width, height = s.read('hex:32, uint:12, uint:12')
aspect_ratio, frame_rate = s.read('bin:4, bin:4')
```

6.2. Manipulation

```
s = BitString('0x0123456789abcdef')

del s[4:8]                      # deletes the '1'
s.insert('0xcc', 12)            # inserts 'cc' between the '3' and '4'
s.overwrite('0b01', 30)         # changes the '6' to a '5'

# This replaces every '1' bit with a 5 byte Ascii string!
s.replace('0b1', BitString(data='hello'))

s.truncateend(1001)             # deletes final 1001 bits
s.reversebits()                 # reverses whole BitString
s.prepend('uint:12=44')         # prepend a 12 bit integer
```

6.3. Parsing

This example creates a class that parses a structure that is part of the H.264 video standard.

```
class seq_parameter_set_data(object):
    def __init__(self, s):
        """Interpret next bits in BitString s as an SPS."""
        # Read and interpret bits in a single expression:
        self.profile_idc = s.read('uint:8')
        # Multiple reads in one go returns a list:
        self.constraint_setx_flags = s.read('uint:1, uint:1, uint:1, uint:1')
        self.reserved_zero_4bits = s.read('bin:4')
        self.level_idc = s.read('uint:8')
        self.seq_parameter_set_id = s.read('ue')
        if self.profile_idc in [100, 110, 122, 244, 44, 83, 86]:
            self.chroma_format_idc = s.read('ue')
            if self.chroma_format_idc == 3:
                self.separate_colour_plane_flag == s.read('uint:1')
            self.bit_depth_luma_minus8 = s.read('ue')
            self.bit_depth_chroma_minus8 = s.read('ue')
            # etc.

>>> s = BitString('0x6410281bc0')
>>> sps = seq_parameter_set_data(s)
>>> print sps.profile_idc
100
>>> print sps.level_idc
40
>>> print sps.reserved_zero_4bits
0b0000
>>> print sps.constraint_setx_flags
[0, 0, 0, 1]
```

A. Reference

The bitstring module provides just one class, `BitString`, whose public methods, special methods and properties are detailed in this section.

Note that in places where a `BitString` can be used as a parameter, any other valid input to the `auto` initialiser can also be used. This means that the parameter can also be a format string which consists of tokens:

- Starting with `hex=`, or simply starting with `0x` implies hexadecimal.
  e.g. `0x013ff, hex=013ff`
- Starting with `oct=`, or simply starting with `0o` implies octal.
  e.g. `0o755, oct=755`
- Starting with `bin=`, or simply starting with `0b` implies binary.
  e.g. `0b0011010, bin=0011010`
- Starting with `int` or `uint` followed by a length in bits then `=` gives base-2 integers.
  e.g. `uint:8=255, int:4=-7`
- To get big, little and native-endian whole-byte integers append `be`, `le` or `ne` respectively to the `uint` or `int` identier.
  e.g. `uintle:32=1, intne:16=-23`
- Starting with `ue=` or `se=` implies an exponential-Golomb coded integer.
  e.g. `ue=12, se=-4`

Multiples tokens can be joined by separating them with commas, so for example `'se=4, 0b1, se=-1'` represents the concatenation of three elements.

The `auto` parameter also accepts a list or tuple, whose elements will be evaluated as booleans (imagine calling `bool()` on each item) and the bits set to `1` for `True` items and `0` for `False` items.

Finally if you pass in a `file` object, presumably opened in read-binary mode, then the `BitString` will be formed from the contents of the file.

For the `read`, `unpack`, `pack` and `peek` functions you can use compact format strings similar to those used in the struct and array modules. These start with an endian identifier: `>` for big-endian, `<` for little-endian or `@` for native-endian. This must be followed by at least one of these codes:

`b`	→	8 bit signed integer
`B`	→	8 bit unsigned integer
`h`	→	16 bit signed integer
`H`	→	16 bit unsigned integer
`l`	→	32 bit signed integer
`L`	→	32 bit unsigned integer
`q`	→	64 bit signed integer
`Q`	→	64 bit unsigned integer

A.1. Class properties

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

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 s.length.

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

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.

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, but this padding behaviour is set to be removed in a future version so you should use tostring() instead if you need this behaviour.

```
>>> s = BitString(data='\x12\xff\x30')
>>> s.data
'\x12\xff0'
>>> s.hex = '0x12345678'
>>> s.data
'\x124Vx'
```

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')
>>> s.hex
'0xf0'
>>> s.hex = 'abcdef'
>>> s.hex
'0xabcdef'
```

int

s.int

Read and write property for setting and getting the signed two's complement integer representation of the BitString.

When used as a setter the value must fit into the current length of the BitString, else a ValueError will be raised.

```
>>> s = BitString('0xf3')
>>> s.int
-13
>>> s.int = 1232
ValueError: int 1232 is too large for a
BitString of length 8.
```

intbe

s.intbe

Read and write property for setting and getting the byte-wise big-endian signed two's complement integer representation of the BitString.

Only valid if s is whole-byte, in which case it is equal to s.int, otherwise a ValueError is raised.

When used as a setter the value must fit into the current length of the BitString, else a ValueError will be raised.

`intle`

`s.intle`

Read and write property for setting and getting the byte-wise little-endian signed two's complement integer representation of the `BitString`.

Only valid if `s` is whole-byte, in which case it is equal to `s[::-8].int`, i.e. the integer representation of the byte-reversed `BitString`.

When used as a setter the value must fit into the current length of the `BitString`, else a `ValueError` will be raised.

`intne`

`s.intne`

Read and write property for setting and getting the byte-wise native-endian signed two's complement integer representation of the `BitString`.

Only valid if `s` is whole-byte, and will equal either the big-endian or the little-endian integer representation depending on the platform being used.

When used as a setter the value must fit into the current length of the `BitString`, else a `ValueError` will be raised.

`length`

`s.length`

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

This is almost equivalent to using `len(s)`, expect that for large `BitString` objects `len()` may fail with an `OverflowError`, whereas the `length` property continues to work (the limit is `sys.maxint`, which equates to 256MB on my machine).

`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`.

```
>>> s = BitString('0b111101101')
>>> s.oct
'0o755'
>>> s.oct = '01234567'
>>> s.oct
'0o01234567'
```

`se`

`s.se`

Read and write property for setting and getting the signed exponential-Golomb code representation of the `BitString`.

The property is set from an signed integer, and when used as a getter a `BitStringError` will be raised if the `BitString` is not a single code.

```
>>> s = BitString(se=-40)
>>> s.bin
0b0000001010001
>>> s += ‘0b1’
>>> s.se
BitStringError: BitString is not a single
exponential-Golomb code.
```

`ue`

`s.ue`

Read and write property for setting and getting the unsigned exponential-Golomb code representation of the `BitString`.

The property is set from an unsigned integer, and when used as a getter a `BitStringError` will be raised if the `BitString` is not a single code.

`uint`

`s.uint`

Read and write property for setting and getting the unsigned base-2 integer representation of the `BitString`.

When used as a setter the value must fit into the current length of the `BitString`, else a `ValueError` will be raised.

`uintbe`

`s.uintbe`

Read and write property for setting and getting the byte-wise big-endian unsigned base-2 integer representation of the `BitString`.

When used as a setter the value must fit into the current length of the `BitString`, else a `ValueError` will be raised.

`uintle`

`s.uintle`

Read and write property for setting and getting the byte-wise little-endian unsigned base-2 integer representation of the `BitString`.

When used as a setter the value must fit into the current length of the `BitString`, else a `ValueError` will be raised.

`uintne`

`s.uintne`

Read and write property for setting and getting the byte-wise native-endian unsigned base-2 integer representation of the `BitString`.

When used as a setter the value must fit into the current length of the `BitString`, else a `ValueError` will be raised.

A.2. Class methods

`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.

`append`

`s.append(bs)`

Join a `BitString` to the end of the current `BitString`. Returns `self`.

```
>>> s = BitString('0xbad')
>>> s.append('0xf00d')
BitString('0xbadf00d')
```

`bytealign`

`s.bytealign()`

Aligns to the start of the next byte (so that `s.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.

```
>>> s = BitString('0xabcdef')
>>> s.advancebits(3)
>>> s.bytealign()
5
>>> s.bitpos
8
```

`cut`

```
s.cut(bits, startbit=None, endbit=None,
      count=None)
```

Returns a generator for slices of the `BitString` of length `bits`.

At most `count` items are returned and the range is given by the slice `[startbit:endbit]`, which defaults to the whole `BitString`.

```
>>> s = BitString('0x1234')
>>> for nibble in s.cut(4):
...     s.prepend(nibble)
>>> print s
0x43211234
```

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. Is equivalent to `del s[bitpos:bitpos+bits]`.

```
>>> s = BitString('0b1111001')
>>> s.deletebits(2, 4)
BitString('0b11111')
```

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.

empty

`s.empty()`

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

```
>>> s = BitString('0b0')
>>> s.empty()
False
>>> del s[:]
>>> s.empty()
True
```

find

*`s.find(bs, startbit=None, endbit=None,
        bytealigned=False)`*

Searches for `bs` in the current `BitString` and sets `bitpos` to the start of `bs` and returns `True` if found, otherwise it returns `False`.

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 the whole `BitString`.

```
>>> s = BitString('0x0023122')
>>> s.find('0b000100', bytealigned=True)
True
>>> s.bitpos
16
```

findall

*`s.findall(bs, startbit=None, endbit=None,
          count=None, bytealigned=False)`*

Searches for all occurrences of `bs` (even overlapping ones) and returns a generator of their bit positions.

If `bytealigned` is `True` then `bs` will only be looked for at byte aligned positions. `startbit` and `endbit` optionally define a search range and default to the whole `BitString`.

```
>>> s = BitString('0xab220101')*5
>>> list(s.findall('0x22',
          bytealigned=True))
[8, 40, 72, 104, 136]
```

insert

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

Inserts `bs` at `bitpos` and returns `self`. After insertion `bitpos` will be immediately after the inserted `BitString`.

The default for `bitpos` is the current position.

```
>>> s = BitString('0xccee')
>>> s.insert('0xd', 8)
BitString('0xccdee')
>>> s.insert('0x00')
BitString('0xccd00ee')
```

join

`s.join(bsl)`

Returns the concatenation of the `BitString` objects in the list `bsl` joined with `s` as a separator.

```
>>> s = BitString().join(['0x0001ee',
'uint:24=13', '0b0111'])
>>> print s
0x0001ee00000d7

>>> s = BitString('0b1').join(['0b0']*5)
>>> print s.bin
0b010101010
```

overwrite

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

Replaces the contents of the current `BitString` with `bs` at `bitpos` and returns `self`. After overwriting `bitpos` will be immediately after the overwritten section.

The default for `bitpos` is the current position.

```
>>> s = BitString(length=10)
>>> s.overwrite('0b111', 3)
BitString('0b0001110000')
>>> s.bitpos
6
```

---

peek

`s.peek(*format)`

Reads from the current bit position `bitpos` in the `BitString` according the the format string(s) and returns either a single `BitString` or a list of `BitString` objects, but does not advance the position.

For information on the `format` string see the entry for the `read` function.

---

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.

If multiple `bits` are specified then a list of `BitString` objects is returned.

```
>>> s = BitString('0xf01')
>>> s.bitpos = 4
>>> s.peekbits(4)
BitString('0x0')
>>> s.peekbits(8)
BitString('0x01')
>>> for bs in s.peekbits(2, 2, 8):
...     print bs
0b11
0b11
0x01
```

---

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.

If multiple `bytes` are specified then a list of `BitString` objects is returned.

---

prepend

`s.prepend(bs)`

Inserts `bs` at the beginning of the current `BitString`. Returns `self`.

```
>>> BitString('0b0').prepend('0xf')
BitString('0b11110')
```

---

read

`s.read(*format)`

Reads from current bit position `bitpos` in the `BitString` according the the format string(s) and returns either a single `BitString` or a list of `BitString` objects.

`format` is one or more strings with comma separated tokens that describe how to interpret the next bits in the `BitString`. The tokens are:

`int:`*n*	*n* bits as an unsigned integer.
`uint:`*n*	*n* bits as a signed integer.
`intbe:`*n*	*n* bits as a big-endian unsigned integer.
`uintbe:`*n*	*n* bits as a big-endian signed integer.
`intle:`*n*	*n* bits as a little-endian unsigned int.
`uintle:`*n*	*n* bits as a little-endian signed integer.
`intne:`*n*	*n* bits as a native-endian unsigned int.
`uintne:`*n*	*n* bits as a native-endain signed int.
`hex:`*n*	*n* bits as a hexadecimal string.
`oct:`*n*	*n* bits as an octal string.
`bin:`*n*	*n* bits as a binary string.
`ue`	next bits as an unsigned exp-Golomb.
`se`	next bits as a signed exp-Golomb.
`bits:`*n*	*n* bits as a new `BitString`.
`bytes:`*n*	*n* bytes as a new `BitString`.

```
>>> s = BitString('0x23ef55302')
>>> s.read('hex12')
'0x23e'
>>> s.read('bin:4, uint:5, bits:4')
['0b1111', 10, BitString('0xa')]
```

The `read` function is useful for reading exponential-Golomb codes, which can't be read easily by `readbits` as their lengths aren't know beforehand.

```
>>> s = BitString('se=-9', 'ue=4')
>>> s.read('se')
-9
>>> s.read('ue')
4
```

---

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.

If multiple `bits` are specified then a list of `BitString` objects is returned.

```
>>> s = BitString('0x0001e2')
>>> s.readbits(16)
BitString('0x0001')
>>> s.readbits(3).bin
'0b111'
>>> s.bitpos = 0
>>> s.readbits(16, 3)
[BitString('0x0001'), BitString('0b111')]
```

---

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.

If multiple `bytes` are specified then a list of `BitString` objects is returned.

---

replace

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

Finds occurrences of `old` and replaces them with `new`. Returns the number of replacements made.

If `bytealigned` is `True` then replacements will only be made on byte boundaries. `startbit` and `endbit` give the search range and default to `0` and `s.length` respectively. If `count` is specified then no more than this many replacements will be made.

```
>>> s = BitString('0b0011001')
>>> s.replace('0b1', '0xf')
3
>>> print s.bin
0b0011111111001111
>>> s.replace('0b1', '', count=6)
6
>>> print s.bin
0b0011001111
```

---

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 bits in the `BitString` in-place and returns self.

`startbit` and `endbit` give the range and default to `0` and `s.length` respectively.

```
>>> a = BitString('0b10111')
>>> a.reversebits().bin
'0b11101'
```

---

reversebytes

s.reversebytes(startbit=None, endbit=None)

Reverses bytes in the `BitString` in-place and returns self.

`startbit` and `endbit` give the range and default to `0` and `s.length` respectively. If `endbit - startbit` is not a multiple of 8 then a `BitStringError` is raised.

Can be used to change the endianness of the `BitString`.

```
>>> s = BitString('uintle:32=1234')
>>> print s.reversebytes().uintbe
1234
```

---

rfind

*s.rfind(bs, startbit=None, endbit=None,
        bytealigned=False)*

Searches backwards for `bs` 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 `s.length` respectively.

Note that as it's a reverse search it will start at `endbit` and finish at `startbit`.

```
>>> s = BitString('0o031544')
>>> s.rfind('0b100')
True
>>> s.bitpos
15
>>> s.rfind('0b100', endbit=17)
True
>>> s.bitpos
12
```

---

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`, or `s.bitpos = bytepos*8`.

---

slice

s.slice(startbit, endbit, step)

Returns the `BitString` slice `s[startbit*step : endbit*step]`.

The `step` parameter gives a multiplicative factor for the start and end positions, so for example using `step` of `8` allows the slice to be given in terms of byte indices rather than bit indices.

It's use is equivalent to using the slice notation; see `__getitem__` for examples.

---

split

*s.split(delimiter, startbit=None,
        endbit=None, count=None,
        bytealigned=False)*

Splits `s` into sections that start with `delimiter`. Returns a generator for `BitString` objects.

The first item generated is always the bits before the first occurrence of `delimiter` (even if empty). A slice can be optionally specified with `startbit` and `endbit`, while `count` specifies the maximum number of items generated.

```
>>> s = BitString('0x42423')
>>> [bs.bin for bs in s.split('0x4')]
['', '0b01000', '0b01001000', '0b0100011']
```

---

tellbit

s.tellbit()

Returns the current bit position.

Equivalent to using the `bitpos` property as a getter.

```
>>> s = BitString('int:12=109')
>>> s.read('hex:12')
'0x06d'
>>> s.tellbit()
12
>>> s.tellbit() == s.bitpos
True
```

---

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 aligned.

---

tofile

`s.tofile(f)`

Writes the `BitString` to the file object `f`.

The data written will be padded at the end with between zero and seven `'0'` bits to make it byte aligned.

```
>>> f = open('newfile', 'wb')
>>> BitString('0x1234').tofile(f)
```

---

tostring

`s.tostring()`

Returns the `BitString` as a Python string.

The returned value will be padded at the end with between zero and seven `'0'` bits to make it byte aligned.

The `tostring` function can also be used to output your `BitString` to a file - just open a file in binary write mode and write the function's output.

```
>>> s.data = 'hello'
>>> s += '0b01'
>>> s.tostring()
'hello@'
```

---

truncateend

`s.truncateend(bits)`

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

A `ValueError` is raised if you try to truncate a negative number of bits, or more bits than the `BitString` contains.

```
>>> s = BitString('0xabcdef')
>>> s.truncateend(12)
BitString('0xabc')
```

---

truncatestart

`s.truncatestart(bits)`

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

A `ValueError` is raised if you try to truncate a negative number of bits, or more bits than the `BitString` contains.

```
>>> s = BitString('0xabcdef')
>>> s.truncatestart(12)
BitString('0xdef')
```

---

unpack

`s.unpack(*format)`

Interprets the whole `BitString` according to the format string(s) and returns either a single `BitString` or a list of `BitString` objects.

`format` is one or more strings with comma separated tokens that describe how to interpret the next bits in the `BitString`. See the entry for `read` for details.

```
>>> s = BitString('int:4=-1, 0b1110')
>>> i, b = s.unpack('int:4, bin')
```

If a token doesn't supply a length (as with `bin` above) then it will try to consume the rest of the `BitString`. Only one such token is allowed.

A.3. Class special methods

__add__ / __radd__

`s1 + s2`

Concatenate two `BitString` objects and return the result. Either `s1` or `s2` can be `auto` initialised.

```
s = BitString(ue=132) + '0xff'
s2 = '0b101' + s
```

__and__ / __rand__

`s1 & s2`

Returns the bit-wise AND between `s1` and `s2`, which must have the same length otherwise a `ValueError` is raised.

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

__contains__

`bs in s`

Returns `True` if `bs` can be found in `s`, otherwise returns `False`.

Equivalent to using `find`, 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
```

__delitem__

del s[startbit:endbit:step]

Deletes the slice specified.

After deletion `bitpos` will be at the deleted slice's position.

__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
>>> a = BitString(uint=13, length=8)
>>> b = BitString(uint=13, length=10)
>>> a == b
False
```

__getitem__

s[startbit:endbit:step]

Returns a slice of `s`.

The usual slice behaviour applies except that the `step` parameter gives a multiplicative factor for `startbit` and `endbit` (i.e. the bits 'stepped over' are included in the slice).

```
>>> s = BitString('0x0123456')
>>> s[0:4]
BitString('0x1')
>>> s[0:3:8]
BitString('0x012345')
```

__iadd__

s1 += s2

Append a `BitString` to the current `BitString` and return the result.

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

__ilshift__

s <<= n

Shifts the bits in `s` in place to the left by `n` places. Returns `self`. Bits shifted off the left hand side are lost, and replaced by `0` bits on the right hand side.

__imul__

*s *= n*

Concatenates `n` copies of `s` and returns `self`. Raises `ValueError` if `n < 0`.

```
>>> s = BitString('0xef')
>>> s *= 3
>>> print s
0xefefef
```

__init__

```
s = BitString(auto=None, length=None,
              offset=0, data=None,
              filename=None, hex=None,
              bin=None, oct=None,
              uint=None, int=None,
              uintbe=None, intbe=None,
              uintle=None, intle=None,
              uintne=None, intne=None,
              ue=None, se=None)
```

Creates a new `BitString`. You must specify at most one of the initialisers `auto`, `data`, `bin`, `hex`, `oct`, `uint`, `int`, `uintbe`, `intbe`, `uintle`, `intle`, `uintne`, `intne`, `se`, `ue` or `filename`. If no initialiser is given then a zeroed `BitString` of `length` bits is created.

`offset` is optional for most initialisers, but only really useful for `data` and `filename`. It gives a number of bits to ignore at the start of the `BitString`.

Specifying `length` is mandatory when using the various integer initialisers. It must be large enough that a `BitString` can contain the integer in `length` bits. It is an error to specify `length` when using the `ue` or `se` initialisers. For other initialisers `length` can be used to truncate data from the end of the input value.

```
>>> s1 = BitString(hex='0x934')
>>> s2 = BitString(oct='0o4464')
>>> s3 = BitString(bin='0b001000110100')
>>> s4 = BitString(int=-1740, length=12)
>>> s5 = BitString(uint=2356, length=12)
>>> s6 = BitString(data='\x93@',
length=12)
>>> s1 == s2 == s3 == s4 == s5 == s6
True
```

For information on the use of the `auto` initialiser see the introduction to this appendix.

```
>>> s = BitString('uint:12=32, 0b110')
>>> t = BitString('0o755, ue:12, int:
3=-1')
```

__invert__

~s

Returns the `BitString` with every bit inverted, that is all zeros replaced with ones, and all ones replaced with zeros.

If the `BitString` is empty then a `BitStringError` will be raised.

```
>>> s = BitString('0b1110010')
>>> print ~s
0b0001101
>>> print ~s & s
0b0000000
```

__irshift__

s >>= n

Shifts the bits in `s` in place by `n` places to the right and returns `self`. The `n` left-most bits will become zeros.

```
>>> s = BitString('0b110')
>>> s >>= 2
>>> s.bin
'0b001'
```

__len__

len(s)

Returns the length of the `BitString` in bits if it is less than `sys.maxsize`, otherwise raises `OverflowError`.

It's recommended that you use the `length` property rather than the `len` function because of the function's behaviour for large `BitString` objects, although calling the special function directly will always work.

```
>>> s = BitString(filename='11GB.mkv')
>>> s.length
93944160032L
>>> len(s)
OverflowError: long int too large to
convert to int
>>> s.__len__()
93944160032L
```

__lshift__

s << n

Returns the `BitString` with its bits shifted `n` places to the left (or `s.len` if it's less). The `n` right-most bits will become zeros.

```
>>> s = BitString('0xff')
>>> s << 4
BitString('0xf0')
```

__mul__ / __rmul__

*s * n / n * s*

Return `BitString` consisting of `n` concatenations of `s`.

```
>>> a = BitString('0x34')
>>> b = a*5
>>> print b
0x3434343434
```

__ne__

s1 != s2

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

__or__ / __ror__

s1 | s2

Returns the bit-wise OR between `s1` and `s2`, which must have the same length otherwise a `ValueError` is raised.

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

__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')
```

`__rshift__`

`s >> n`

Returns the `BitString` with its bits shifted `n` places to the right (or `s.len` if it's less). The `n` left-most bits will become zeros.

```
>>> s = BitString('0xff')
>>> s >> 4
BitString('0x0f')
```

`__setitem__`

`s1[startbit:endbit:step] = s2`

Replaces the slice specified with `s2`.

```
>>> s = BitString('0x00112233')
>>> s[1:2:8] = '0xfff'
>>> print s
0x00fff2233
>>> s[-12:] = '0xc'
>>> print s
0x00fff2c
```

`__str__`

`print s`

Prints a representation of `s`, trying to be as brief as possible.

If `s` is a multiple of 4 bits long then hex will be used, otherwise either binary or a mix of hex and binary will be used. Very long strings will be truncated with '`...`'.

```
>>> s = BitString('0b1')*7
>>> print s
0b1111111
>>> print s + '0b1'
0xff
```

`__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
```

A.4. Module methods

`pack`

`s = bitstring.pack(format, *values,`
`**kwargs)`

Packs the values and keyword arguments according to the format string and returns a new `BitString`.

The format string consists of comma separated tokens of the form `name:length=value`. See the entry for `read` for more details.

The tokens can be 'literals', like `0xef`, `0b110`, `uint:8=55`, etc. which just represent a set sequence of bits.

They can also have the `value` missing, in which case the values contined in `*values` will be used.

```
>>> a = pack('bin:3, hex:4', '001', 'f')
>>> b = pack('uint:10', 33)
```

A dictionary or keyword arguments can also be provided. These will replace items in the format string.

```
>>> c = pack('int:a:b', a=10, b=20)
>>> d = pack('int:8=a, bin=b, int:4=a',
             a=7, b='0b110')
```

Plain names can also be used as follows:

```
>>> e = pack('a, b, b, a', a='0b11',
             b='0o2')
```

Tokens starting with an endianness identifier (`<`, `>` or `@`) implies a struct-like compact format string. For example this packs three little-endian 16-bit integers:

```
>>> f = pack('<3h', 12, 3, 108)
```

And of course you can combine the different methods in a single `pack`.

A `ValueError` will be raised if the `*values` are not all used up by the format string, and if a value provided doesn't match the length specified by a token.

B. 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:

Bit pattern	**Unsigned**	**Signed**
`1`	0	0
`010`	1	1
`011`	2	-1
`00100`	3	2
`00101`	4	-2
`00110`	5	3
`00111`	6	-3
`0001000`	7	4
`0001001`	8	-4
`0001010`	9	5
`0001011`	10	-5
`0001100`	11	6
`...`	...	...

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.read('ue')
```

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.

C. Internals

I am including some information on the internals of the `BitString` class here, things that the general user shouldn't need to know. The objects and methods described here all start with an underscore, which means that they are a private part of the implementation, not a part of the public interface and that that I reserve the right to change, rename and remove them at any time!

This appendix isn't complete, and may not even be accurate as I am in the process of refactoring the core, so with those disclaimers in mind...

The data in a `BitString` can be considered to consist of three parts.

The byte data, either contained in memory, or as part of a file.

A length in bits.

An offset to the data in bits.

Storing the data in byte form is pretty essential, as anything else could be very memory inefficient. Keeping an offset to the data allows lots of optimisations to be made as it means that the byte data doesn't need to be altered for almost all operations. An example is in order:

```
a = BitString('0x01ff00')
b = a[7:12]
```

This is about as simple as it gets, but let's look at it in detail. First `a` is created by parsing the string as hexadecimal (as it starts with `0x`) and converting it to three data bytes `\x01\xff\x00`. By default the length is the bit length of the whole string, so it's 24 in this case, and the offset is zero.

Next, `b` is created from a slice of `a`. This slice doesn't begin or end on a byte boundary, so one way of obtaining it would be to copy the data in `a` and start doing bit-wise shifts to get it all in the right place. This can get really very computationally expensive, so instead we utilise the offset and length parameters.

The procedure is simply to copy the byte data containing the substring and set the offset and length to get the desired result. So in this example we have:

```
a : data = '\x01\xff\x00', offset = 0, len = 24
b : data = '\x01\xff', offset = 7, len = 5
```

This method also means that BitString objects initialised from a file don't have to copy anything into memory - the data instead is obtained with a byte offset into the file. This brings us onto the different types of datastores used.

The `BitString` has a `_datastore` member, which at present is either a `_MemArray` class or a `_FileArray` class. The `_MemArray` class is really just a light wrapper around an `array.array` object that contains the real byte data, so when we were talking about the data earlier I was really referring to the byte data contained in the `array.array`, in the `_MemArray`, in the `_datastore`, in the `BitString` (but that seemed a bit much to give you in one go).