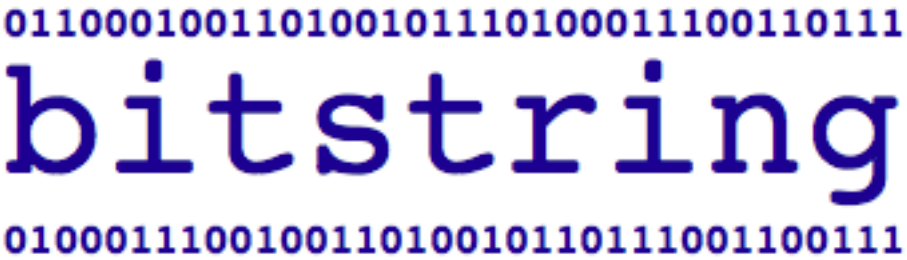

bitstring Documentation

Release 1.3

Scott Griffiths

March 18, 2010

CONTENTS

Part I

User Manual

CHAPTER

ONE

INTRODUCTION

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

The bitstring module provides two classes, `BitString` and `Bits`, instances of which can be constructed from integers, floats, hex, octal, binary, strings or files, but they all just represent a string of binary digits. The `Bits` class differs from `BitString` in that it is immutable and so does not support methods that would modify it after creation. I shall use the general term 'bitstring' when referring to either the `Bits` or `BitString` class, and use the class names for parts that apply to only one or the other.

`BitString` objects 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.

Bitstrings are designed to be as lightweight as possible and can be considered to be just a list of binary digits. They are however stored 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 *Reference*, while the rest of this manual acts more like a tutorial or guided tour. Below are 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:

```
# from a binary string
a = BitString('0b001')
# from a hexadecimal string
b = BitString('0xff470001')
# straight from a file
c = BitString(filename='somefile.ext')
# from an integer
d = BitString(int=540, length=11)
# using a format string
d = BitString('int:11=540')
```

Easily construct new bitstrings:

```
# 5 copies of 'a' followed by two new bytes
e = 5*a + '0xcdcd'
```

```
# put a single bit on the front
e.prepend('0b1')
# take a slice of the first 7 bits
f = e[7:]
# replace 3 bits with 9 bits from octal string
f[1:4] = '0o775'
# find and replace 2 bit string with 16 bit string
f.replace('0b01', '0xee34')
```

Interpret the bitstring however you want:

```
>>> print(e.hex)
'0x9249cdcd'
>>> print(e.int)
-1840656947
>>> print(e.uint)
2454310349
```

1.1 Getting Started

The easiest way to install `bitstring` is to use `easy_install` via:

```
sudo easy_install bitstring
```

or similar.

If you want an earlier version, or need other files in the full package, you can download and extract the contents of the .zip provided on the project's website.

First download the latest release for either Python 2.4 / 2.5 or Python 2.6 / 3.0 / 3.1 (see the Downloads tab on the project's homepage). Note that this manual covers only the Python 2.6 and later version. An earlier version is available for Python 2.4 / 2.5, which is available on the project's homepage.

If you then extract the contents of the zip file you should find:

- `bitstring.py` : The bitstring module itself.
- `test/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.
- `doc/bitstring_manual.pdf` : This manual as a PDF.
- `doc/html` : This manual as HTML.

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 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 and 2.5 and a separate version for Python 2.6, 3.0 and 3.1) or something unexpected has happened - in which case please tell me about it.

CHAPTER

TWO

CREATION AND INTERPRETATION

You can create bitstrings in a variety of ways. Internally they are stored as byte arrays, which 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.

2.1 The Bits and BitString classes

The two classes provided by the bitstring module are `Bits` and `BitString`. The major difference between them is that `BitString` objects are mutable, whereas `Bits` objects cannot be changed after creation.

Most of the exampes in this manual use the `BitString` class rather than the `Bits` class, which is partly historical (the `Bits` class is a new addition) but also because the `BitString` is more versatile and so probably your best choice when starting to use the module.

To summarise when to use each class:

- If you need to change the contents of the bitstring then you must use `BitString`. Truncating, replacing, inserting, appending etc. are not available for `Bits` objects.
- If you need to use a bitstring as the key in a dictionary or as a member of a `set` then you must use `Bits`. As `BitString` objects are mutable they do not support hashing and so cannot be used in these ways.
- If you don't need the extra functionality of `BitString` objects then using `Bits` might be faster and more memory efficient. Currently the speed difference is quite marginal, but this is expected to improve in future releases.

The `Bits` class is the base class of `BitString`. This means that for example `isinstance(s, Bits)` will be true if `s` is an instance of either class.

2.2 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, integer, bool or another bitstring.
- `bytes` : A `bytes` object (a `str` in Python 2.6), 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.
- `float` / `floatbe`, `floatle`, `floatne`: Big, little and native endian floating point numbers.
- `se`, `ue` : Signed or unsigned exponential-Golomb coded integers.
- `filename` : Directly from a file, without reading into memory.

2.2.1 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')
```

2.2.2 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')
```

2.2.3 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')
```

2.2.4 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.

2.2.5 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 bitstrings.

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 bitstrings, 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`).

2.2.6 From a floating point number

```
>>> f1 = BitString(float=10.3, length=32)
>>> f2 = BitString('float:64=5.4e31')
```

Floating point numbers can be used for initialisation provided that the bitstring is 32 or 64 bits long. Standard Python floating point numbers are 64 bits long, so if you use 32 bits then some accuracy could be lost.

Note that the exact bits used to represent the floating point number could be platform dependent. Most PCs will conform to the IEEE 754 standard, and presently other floating point representations are not supported (although they should work on a single platform - it just might get confusing if you try to interpret a generated bitstring on another platform).

Similar to the situation with integers there are big and little endian versions. The plain `float` is big endian and so `floatbe` is just an alias.

As with other initialisers you can also auto initialise, as demonstrated with the second example below:

```
>>> little_endian = BitString(floatle=0.0, length=64)
>>> native_endian = BitString('floatne:32=-6.3')
```

2.2.7 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
'0b00000000001100100101'
```

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 a section on them (see *Exponential-Golomb Codes*).

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.

2.2.8 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(bytes='\x00\x01\x02\xff', length=28, offset=1)
b = BitString(bytes=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.

2.2.9 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 bitstrings 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)
```

2.3 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, an integer, a bool 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. Finally you can use an integer to create a zeroed bitstring of that number of bits.

```
>>> fromhex = BitString('0x01ffc9')
>>> frombin = BitString('0b01')
>>> fromoct = BitString('0o7550')
>>> fromint = BitString('int:32=10')
>>> fromfloat = BitString('float:64=0.2')
>>> acopy = BitString(fromoct)
>>> fromlist = BitString([True, False, False])
>>> f = open('somefile', 'rb')
>>> fromfile = BitString(f)
>>> zeroed = BitString(1000)
>>> frombool = BitString(True)
```

It can also be used to convert between the `BitString` and `Bits` classes:

```
>>> immutable = Bits('0xabc')
>>> mutable = BitString(immutable)
>>> mutable += '0xdef'
>>> immutable = Bits(mutable)
```

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 bitstrings 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)
```

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

2.4 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 function 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 items 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 a signed integer.
`uint:n`	`n` bits as an unsigned integer.
`intbe:n`	`n` bits as a big-endian whole byte signed integer.
`uintbe:n`	`n` bits as a big-endian whole byte unsigned integer.
`intle:n`	`n` bits as a little-endian whole byte signed integer.
`uintle:n`	`n` bits as a little-endian whole byte unsigned integer.
`intne:n`	`n` bits as a native-endian whole byte signed integer.
`uintne:n`	`n` bits as a native-endian whole byte unsigned integer.
`float:n`	`n` bits as a big-endian floating point number (same as `floatbe`).
`floatbe:n`	`n` bits as a big-endian floating point number (same as `float`).
`floatle:n`	`n` bits as a little-endian floating point number.
`floatne:n`	`n` bits as a native-endian floating point number.
`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.
`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.
float:n=f	big-endian floating point number f in n bits.
floatbe:n=f	big-endian floating point number f in n bits.
floatle:n=f	little-endian floating point number f in n bits.
floatne:n=f	native-endian floating point number f 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 Bits.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]
```

2.4.1 Compact format strings

Another option when using pack, as well as other methods such as Bits.read and BitString.byteswap, 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 (except when used with BitString.byteswap):

>	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
f	32 bit floating point number
d	64 bit floating point number

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. 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 format string can be parsed correctly.

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 endiannesses (although I'm not sure why you'd want to):

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

This rather contrived example takes the numbers 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.

2.5 Interpreting Bitstrings

Bitstrings 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 = Bits('0x123')
>>> b = Bits('0b111')
```

2.5.1 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 bitstrings can use this property and it is used to test equality between bitstrings.

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

Note that the initial zeros are significant; for bitstrings the zeros are just as important as the ones!

2.5.2 hex

For whole-byte bitstrings 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.
```

2.5.3 oct

For an octal interpretation use the `oct` property. Octal values are prefixed with `0o`, which is the Python 2.6 / 3 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.
```

2.5.4 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 be 1 on a big-endian platform!)
65536
```

2.5.5 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.

2.5.6 float / floatbe / floatle / floatne

For a floating point interpretation use the `float` property. This uses your machine's underlying floating point representation and will only work if the bitstring is 32 or 64 bits long.

Different endiannesses are provided via `floatle` and `floatne`. Note that as floating point interpretations are only valid on whole-byte bitstrings there is no difference between the bit-wise big-endian `float` and the byte-wise big-endian `floatbe`.

Note also that standard floating point numbers in Python are stored in 64 bits, so use this size if you wish to avoid rounding errors.

2.5.7 bytes

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 `bytes` interpretation, which returns a `bytes` object (which is equivalent to an ordinary `str` in Python 2.6).

If the length of the bitstring isn't a multiple of eight then a `ValueError` will be raised. This is because there isn't an unequivocal representation as `bytes`. You may prefer to use the method `tobytes` as this will be pad with between one and seven zero bits up to a byte boundary if neccessary.

```
>>> open('somefile', 'wb').write(a.tobytes())
>>> open('anotherfile', 'wb').write(('0x0'+a).bytes)
>>> a1 = BitString(filename='somefile')
>>> a1.hex
'0x1230'
>>> a2 = BitString(filename='anotherfile')
>>> a2.hex
'0x0123'
```

Note that the `tobytes` method automatically padded with four zero bits at the end, whereas for the other example we explicitly padded at the start to byte align before using the `bytes` property.

2.5.8 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 *Exponential-Golomb Codes* for a short explanation of this type of integer representation.

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

2.5.9 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 *Exponential-Golomb Codes* for more information.

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

CHAPTER

THREE

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 takes three arguments: the first position you want, one past the last position you want and a multiplicative factor which defaults to 1.

The third argument (the 'step') will be described shortly, but most of the time you'll probably just need the bit-wise slice, where for example `a[10:12]` will return a 2-bit bitstring of the 10th and 11th bits in `a`, and `a[32]` will return just the 32nd bit.

```
>>> a = BitString('0b00011110')
>>> b = a[3:7]
>>> print(a, b)
0x1e 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
```

3.1.1 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 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 using `s.reverse()`, 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 `BitString.append` and `BitString.prepend` methods.

```
# 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')
a4.append('0xf')
a5 = BitString('0xf')
a5.prepend('0b000')
a6 = BitString('0b000')
a6 += '0xf'
```

Note that the final three methods all modify a bitstring, and so will only work with `BitString` objects, not the immutable `Bits` objects.

If you want to join a large number of bitstrings then the method `Bits.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)
```

3.3 Truncating, inserting, deleting and overwriting

The functions in this section all modify the bitstring that they operate on and so are not available for `Bits` objects.

3.3.1 Deleting and truncating

To delete bits just use `del` as you would with any other container:

```
>>> a = BitString('0b00011000')
>>> del a[3:5]                # remove 2 bits at pos 3
>>> a.bin
'0b000000'
>>> b = BitString('0x112233445566')
>>> del b[24:40]
>>> b.hex
'0x11223366'
```

You can of course use this to truncate the start or end bits just as easily:

```
>>> a = BitString('0x001122')
>>> del a[-8:]   # remove last 8 bits
>>> del a[:8]    # remove first 8 bits
>>> a == '0x11'
True
```

3.3.2 `insert`

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

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

3.3.3 `overwrite`

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

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

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 methods and the standard slice notation.

Using functions	Using slices
`s.insert(bs, pos)`	`s[pos:pos] = bs`
`s.overwrite(bs, pos)`	`s[pos:pos + bs.len] = bs`
`s.append(bs)`	`s[s.len:s.len] = bs`
`s.prepend(bs)`	`s[0:0] = bs`

3.5 Splitting

3.5.1 split

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

```
>>> a = BitString('0x4700004711472222')
>>> for s in a.split('0x47', bytealigned=True):
...     print(s.hex)

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.

3.5.2 cut

If you just want to split into equal parts then use the `Bits.cut` method. 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
```

CHAPTER

FOUR

READING, PARSING AND UNPACKING

4.1 Reading and parsing

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 bitstrings. These new objects are top-level bitstring objects and can be interpreted using properties or could be read from themselves to form a hierarchy of reads.

In order to behave like a file or stream, every bitstring has a property `pos` which is the current position from which reads occur. `pos` 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. `pos` isn't a multiple of 8) then a `BitStringError` exception will be raised.

4.1.1 `readbit(s)` / `readbitlist` / `readbyte(s)` / `readbytelist`

For simple reading of a number of bits you can use `Bits.readbits` or `Bits.readbytes`. The following example does some simple parsing of an MPEG-1 video stream (the stream is provided in the `test` directory if you downloaded the source archive).

```
>>> s = Bits(filename='test/test.m1v')
>>> print(s.pos)
0
>>> start_code = s.readbytes(4).hex
>>> width = s.readbits(12).uint
>>> height = s.readbits(12).uint
>>> print(start_code, width, height, s.pos)
0x000001b3 352 288 56
>>> s.pos += 37
>>> flags = s.readbits(2)
>>> constrained_parameters_flag = flags.readbit().uint
>>> load_intra_quantiser_matrix = flags.readbit().uint
>>> print(s.pos, flags.pos)
95 2
```

If you want to read multiple items in one go you can use `Bits.readbitlist` or `Bits.readbytelist`. These take one or more integer parameters and return a list of bitstring objects. So for example instead of writing:

```
a = s.readbytes(4)
b = s.readbyte()
c = s.readbytes(3)
```

you can equivalently use just:

```
a, b, c = s.readbytelist(4, 1, 3)
```

4.1.2 read / readlist

As well as the Bits.readbits / Bits.readbytes methods there are also plain Bits.read / Bits.readlist methods. These takes a format string similar to that used in the auto initialiser. Only one token should be provided to Bits.read and a single value is returned. To read multiple tokens use Bits.readlist, which unsurprisingly returns a list.

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 a signed integer.
uint:n	n bits as an unsigned integer.
intbe:n	n bits as a byte-wise big-endian signed integer.
uintbe:n	n bits as a byte-wise big-endian unsigned integer.
intle:n	n bits as a byte-wise little-endian signed integer.
uintle:n	n bits as a byte-wise little-endian unsigned integer.
intne:n	n bits as a byte-wise native-endian signed integer.
uintne:n	n bits as a byte-wise native-endian unsigned integer.
float:n	n bits as a big-endian floating point number (same as floatbe).
floatbe:n	n bits as a big-endian floating point number (same as float).
floatle:n	n bits as a little-endian floating point number.
floatne:n	n bits as a native-endian floating point number.
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 bytes object.
ue	next bits as an unsigned exponential-Golomb code.
se	next bits as a signed exponential-Golomb code.

So in the earlier example we could have written:

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

and we also could have combined the three reads as:

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

where here we are also taking advantage of the default uint interpretation for the second and third tokens.

You are allowed to use one 'stretchy' token in a Bits.readlist. This is a token without a length specified which will stretch to fill encompass as many bits as possible. This is often useful when you just want to assign something to 'the rest' of the bitstring:

```
a, b, everthing_else = s.readlist('intle:16, intle:24, bits')
```

In this example the `bits` token will consist of everything left after the first two tokens are read, and could be empty.

It is an error to use more than one stretchy token, or to use a `ue` or `se` token after a stretchy token (the reason you can't use exponential-Golomb codes after a stretchy token is that the codes can only be read forwards; that is you can't ask "if this code ends here, where did it begin?" as there could be many possible answers).

4.1.3 peek

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 is first 2 bytes '0x4732'
else:
    s.find('0x47')
```

The complete list of read and peek functions is `read(format)`, `readlist(*format)`, `readbit()`, `readbits(bits)`, `readbitlist(*bits)`, `readbyte()`, `readbytes(bytes)`, `readbytelist(*bytes)`, `peek(*format)`, `peeklist(*format)`, `peekbit()`, `peekbits(bits)`, `peekbitlist(*bits)`, `peekbyte()`, `peekbytes(bytes)` and `peekbytelist(*bytes)`.

4.2 Unpacking

The `Bits.unpack` method works in a very similar way to `Bits.readlist`. The major difference is that it interprets the whole bitstring from the start, and takes no account of the current `pos`. 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')
```

4.3 Seeking

The properties `pos` and `bytepos` are available for getting and setting the position, which is zero on creation of the bitstring.

Note that you can only use `bytepos` 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.pos
0
>>> s.bytepos += 2
>>> s.pos                       # note pos verses bytepos
16
>>> s.pos += 4
```

```
>>> print(s.read('bin:4'))    # the final nibble '0x6'
0b0110
```

4.4 Finding and replacing

4.4.1 find / rfind

To search for a sub-string use the Bits.find method. 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
```

Bits.rfind does much the same as Bits.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.pos)
True 31                              # Found within the 0x3e near the end
```

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

4.4.2 findall

To find all occurrences of a bitstring inside another (even overlapping ones), use Bits.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]
```

4.4.3 replace

To replace all occurrences of one BitString with another use BitString.replace. The replacements are done in-place, and the number of replacements made is returned. This methods changes the contents of the bitstring and so isn't available for the Bits class.

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

CHAPTER

FIVE

MISCELLANY

5.1 Other Functions

5.1.1 bytealign

`Bits.bytealign` advances between zero and seven bits to make the `pos` a multiple of eight. It returns the number of bits advanced.

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

5.1.2 reverse

This simply reverses the bits of the `BitString` in place. You can optionally specify a range of bits to reverse.

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

5.1.3 reversebytes

This reverses the bytes of the `BitString` in place. 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()
>>> a.hex
'0x563412'
>>> a.reversebytes(0, 16)
```

```
>>> a.hex
'0x345612'
```

5.1.4 tobytes

Returns the byte data contained in the bitstring as a `bytes` object (equivalent to a `str` if you're using Python 2.6). This differs from using the plain `bytes` property in that if the bitstring isn't a whole number of bytes long then it will be made so by appending up to seven zero bits.

```
>>> BitString('0b1').tobytes()
'\x80'
```

5.1.5 tofile

Writes the byte data contained in the bitstring to a file. The file should have been opened in a binary write mode, for example:

```
>>> f = open('newfile', 'wb')
>>> BitString('0xffee3241fed').tofile(f)
```

In exactly the same manner as with `Bits.tobytes`, up to seven zero bits will be appended to make the file a whole number of bytes long.

5.1.6 startswith / endswith

These act like the same named functions on strings, that is they return `True` if the bitstring starts or ends with the parameter given. Optionally you can specify a range of bits to use.

```
>>> s = BitString('0xef133')
>>> s.startswith('0b111011')
True
>>> s.endswith('0x4')
False
```

5.1.7 ror / rol

To rotate the bits in a `BitString` use `BitString.ror` and `BitString.rol` for right and left rotations respectively. The changes are done in-place.

```
>>> s = BitString('0x00001')
>>> s.rol(6)
>>> s.hex
'0x00040'
```

5.2 Special Methods

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

5.2.1 __len__

This implements the `len` function and returns the length of the bitstring in bits.

It's recommended that you use the `len` 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
```

5.2.2 __str__ / __repr__

These get called when you try to print a bitstring. As bitstrings 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 `Bits.__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 bitstrings from tying up your interactive session while they print themselves.

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

5.2.3 __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
```

5.2.4 __invert__

To get a bit-inverted copy of a bitstring use the ~ operator:

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

5.2.5 __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, for example use `a.int >>= 2`.

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

5.2.6 __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
```

5.2.7 __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 the only method of making a copy. Using `b = BitString(a)` is another option, but `b = a[:]` may be more familiar to some.

5.2.8 __and__ / __or__ / __xor__ / __iand__ / __ior__ / __ixor__

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
```

Part II

Reference

CHAPTER

SIX

CLASSES

6.1 BitString and Bits

The bitstring module provides just two classes, `BitString` and `Bits`. These share many methods as `Bits` is the base class for `BitString`. The distinction between them is that `Bits` represents an immutable sequence of bits whereas `BitString` objects support many methods that mutate their contents.

If you need to change the contents of a bitstring then you must use the `BitString` class. If you need to use bitstrings as keys in a dictionary or members of a set then you must use the `Bits` class (`Bits` are hashable). Otherwise you can use whichever you prefer, but note that `Bits` objects can potentially be more efficent than `BitString` objects. In this section the generic term 'bitstring' means either a `Bits` or a `BitString` object.

Note that the bit position within the bitstring (the position from which reads occur) can change without affecting the equality operation. This means that the `pos` and `bytepos` properties can change even for a `Bits` object.

The public methods, special methods and properties of both classes are detailed in this section.

6.1.1 The auto initialiser

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 and = 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` identifier. e.g. `uintle:32=1`, `intne:16=-23`
- For floating point numbers use `float:` followed by the length in bits and = and the number. The default is big-endian, but you can also append `be`, `le` or `ne` as with integers. e.g. `float:64=0.2`, `floatle:32=-0.3e12`
- 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.

Parentheses and multiplicative factors can also be used, for example `2*(0b10, 0xf)` is equivalent to `0b10, 0xf, 0b10, 0xf`. The multiplying factor must come before the thing it is being used to repeat.

The auto parameter also accepts other types:

- 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.
- A positive integer, used to create a bitstring of that many zero bits.
- A file object, presumably opened in read-binary mode, from which the bitstring will be formed.
- A bool (True or False) which will be converted to a single 1 or 0 bit respectively.

6.1.2 Compact format strings

For the Bits.read, Bits.unpack, Bits.peek methods and pack function 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:

Code	Interpretation
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
f	32 bit floating point number
d	64 bit floating point number

For more detail see *Compact format strings*.

6.2 The Bits class

class Bits (*[auto, length, offset, **kwargs]*)

Creates a new bitstring. You must specify either no initialiser, just an auto value, or one of the keyword arguments bytes, bin, hex, oct, uint, int, uintbe, intbe, uintle, intle, uintne, intne, se, ue, float, floatbe, floatle, floatne or filename. If no initialiser is given then a zeroed bitstring of length bits is created.

The initialiser for the Bits class is precisely the same as for BitString.

offset is optional for most initialisers, but only really useful for bytes 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 = Bits(hex='0x934')
>>> s2 = Bits(oct='0o4464')
>>> s3 = Bits(bin='0b001000110100')
>>> s4 = Bits(int=-1740, length=12)
>>> s5 = Bits(uint=2356, length=12)
>>> s6 = Bits(bytes='\x93@', length=12)
>>> s1 == s2 == s3 == s4 == s5 == s6
True
```

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

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

allset (*pos*)
Returns `True` if one or many bits are all set to `1`, otherwise returns `False`.

pos can be either a single bit position or an iterable of bit positions. Negative numbers are treated in the same way as slice indices and it will raise an `IndexError` if `pos < -s.len` or `pos > s.len`

See also `Bits.allunset`.

allunset (*pos*)
Returns `True` if one or many bits are all set to `0`, otherwise returns `False`.

pos can be either a single bit position or an iterable of bit positions. Negative numbers are treated in the same way as slice indices and it will raise an `IndexError` if `pos < -s.len` or `pos > s.len`

See also `Bits.allset`.

anyset (*pos*)
Returns `True` if any of one or many bits are set to `1`, otherwise returns `False`.

pos can be either a single bit position or an iterable of bit positions. Negative numbers are treated in the same way as slice indices and it will raise an `IndexError` if `pos < -s.len` or `pos > s.len`

See also `Bits.anyunset`.

anyunset (*pos*)
Returns `True` if any of one or many bits are set to `0`, otherwise returns `False`.

pos can be either a single bit position or an iterable of bit positions. Negative numbers are treated in the same way as slice indices and it will raise an `IndexError` if `pos < -s.len` or `pos > s.len`

See also `Bits.anyset`.

bytealign ()
Aligns to the start of the next byte (so that `pos` 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 = Bits('0xabcdef')
>>> s.pos += 3
>>> s.bytealign()
5
>>> s.pos
8
```

cut (*bits, [start, end, count]*)
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 *[start:end]*, which defaults to the whole bitstring.

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

endswith (*bs, [start, end]*)
Returns `True` if the bitstring ends with the sub-string *bs*, otherwise returns `False`.

A slice can be given using the *start* and *end* bit positions and defaults to the whole bitstring.

```
>>> s = Bits('0x35e22')
>>> s.endswith('0b10, 0x22')
True
>>> s.endswith('0x22', start=13)
False
```

find (*bs, [start, end, bytealigned=False]*)
Searches for *bs* in the current bitstring and sets `pos` 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). *start* and *end* give the search range and default to the whole bitstring.

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

findall (*bs, [start, end, count, 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. *start* and *end* optionally define a search range and default to the whole bitstring.

The *count* paramater limits the number of items that will be found - the default is to find all occurences.

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

join (*sequence*)
Returns the concatenation of the bitstrings in the iterable *sequence* joined with `self` as a separator.

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

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

peek (*format*)
Reads from the current bit position `pos` in the bitstring according to the *format* string and returns result.

The bit position is unchanged.

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

peeklist (**format, **kwargs*)
Reads from current bit position `pos` in the bitstring according to the *format* string(s) and returns a list of results.

A dictionary or keyword arguments can also be provided. These will replace length identifiers in the format string. The position is not advanced to after the read items.

See the entries for `Bits.read` and `Bits.readlist` for more information.

peekbit()
Returns the next bit in the current bitstring as a new bitstring but does not advance the position.

peekbits(*bits*)
Returns the next *bits* bits of the current bitstring as a new bitstring but does not advance the position.

```
>>> s = Bits('0xf01')
>>> s.pos = 4
>>> s.peekbits(4)
Bits('0x0')
>>> s.peekbits(8)
Bits('0x01')
```

peekbitlist(**bits*)
Reads multiple *bits* from the current position and returns a list of bitstring objects, but does not advance the position.

```
>>> s = Bits('0xf01')
>>> for bs in s.peekbits(2, 2, 8):
...     print(bs)
0b11
0b11
0x01
>>> s.pos
0
```

peekbyte()
Returns the next byte of the current bitstring as a new bitstring but does not advance the position.

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.

peekbytelist(**bytes*)
Reads multiple *bytes* from the current position and returns a list of bitstring objects, but does not advance the position.

```
>>> s = Bits('0x34eedd')
>>> print(s.peekbytelist(1, 2))
[Bits('0x34'), Bits('0xeedd')]
```

read(*format*)
Reads from current bit position `pos` in the bitstring according the the format string and returns a single result. If not enough bits are available then all bits to the end of the bitstring will be used.

format is a token string that describe how to interpret the next bits in the bitstring. The tokens are:

int:n	n bits as a signed integer.
uint:n	n bits as an unsigned integer.
float:n	n bits as a floating point number.
intbe:n	n bits as a big-endian signed integer.
uintbe:n	n bits as a big-endian unsigned integer.
floatbe:n	n bits as a big-endian float.
intle:n	n bits as a little-endian signed int.
uintle:n	n bits as a little-endian unsigned int.
floatle:n	n bits as a little-endian float.
intne:n	n bits as a native-endian signed int.
uintne:n	n bits as a native-endian unsigned int.
floatne:n	n bits as a native-endian float.
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 bytes object.

For example:

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

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

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

readlist(**format, **kwargs*)

Reads from current bit position pos in the bitstring according to the *format* string(s) and returns a list of results. If not enough bits are available then all bits to the end of the bitstring will be used.

A dictionary or keyword arguments can also be provided. These will replace length identifiers in the format string. The position is advanced to after the read items.

See the entry for Bits.read for information on the format strings.

For multiple items you can separate using commas or given multiple parameters:

```
>>> s = Bits('0x43fe01ff21')
>>> s.readlist('hex:8, uint:6')
['0x43', 63]
>>> s.readlist('bin:3', 'intle:16')
['0b100', -509]
```

```
>>> s.pos = 0
>>> s.readlist('hex:b, uint:d', b=8, d=6)
['0x43', 63]
```

readbit()
Returns the next bit of the current bitstring as a new bitstring and advances the position.

readbits(*bits*)
Returns the next *bits* bits of the current bitstring as a new bitstring and advances the position.

```
>>> s = Bits('0x0001e2')
>>> s.readbits(16)
Bits('0x0001')
>>> s.readbits(3).bin
'0b111'
```

readbitlist(**bits*)
Reads multiple *bits* from the current bitstring and returns a list of bitstring objects. The position is advanced to after the read items.

```
>>> s = Bits('0x0001e2')
>>> s.readbitlist(16, 3)
[Bits('0x0001'), Bits('0b111')]
>>> s.readbitlist(1)
[Bits('0b0')]
```

readbyte()
Returns the next byte of the current bitstring as a new bitstring and advances the position.

readbytes(*bytes*)
Returns the next *bytes* bytes of the current bitstring as a new bitstring and advances the position.

readbytelist(**bytes*)
Reads multiple bytes from the current bitstring and returns a list of bitstring objects.

The position is advanced to after the read items.

rfind(*bs, [start, end, bytealigned=False]*)
Searches backwards for *bs* in the current bitstring and returns `True` if found, otherwise returns `False`.

If *bytealigned* is `True` then it will look for *bs* only at byte aligned positions. *start* and *end* give the search range and default to `0` and `len` respectively.

Note that as it's a reverse search it will start at *end* and finish at *start*.

```
>>> s = Bits('0o031544')
>>> s.rfind('0b100')
True
>>> s.pos
15
>>> s.rfind('0b100', end=17)
True
>>> s.pos
12
```

split(*delimiter, [start, end, count, bytealigned=False]*)
Splits the bitstring 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 *start* and *end*, while *count* specifies the maximum number of items generated.

If *bytealigned* is `True` then the delimiter will only be found if it starts at a byte aligned position.

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

startswith(*bs, [start, end]*)

Returns `True` if the bitstring starts with the sub-string *bs*, otherwise returns `False`.

A slice can be given using the *start* and *end* bit positions and defaults to the whole bitstring.

tobytes()

Returns the bitstring as a `bytes` object (equivalent to a `str` in Python 2.6).

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

The `Bits.tobytes` method 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 = Bits(bytes='hello')
>>> s += '0b01'
>>> s.tobytes()
'hello@'
```

tofile(*f*)

Writes the bitstring to the file object *f*, which should have been opened in binary write mode.

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')
>>> Bits('0x1234').tofile(f)
```

unpack(**format, **kwargs*)

Interprets the whole bitstring according to the *format* string(s) and returns a list of bitstring objects.

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

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 `Bits.read` for details.

```
>>> s = Bits('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.

__add__(*bs*)

__radd__(*bs*)

`s1 + s2`

Concatenate two bitstring objects and return the result. Either bitstring can be 'auto' initialised.

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

__and__(*bs*)

__rand__(*bs*)

`s1 & s2`

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

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

__contains__(*bs*)

`bs in s`

Returns `True` if *bs* can be found in the bitstring, otherwise returns `False`.

Equivalent to using `Bits.find`, except that `pos` will not be changed so you don't know where it was found.

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

__copy__()

`s2 = copy.copy(s1)`

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

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

__eq__(*bs*)

`s1 == s2`

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

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

__getitem__(*key*)

`s[start:end:step]`

Returns a slice of the bitstring.

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

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

__hash__()
hash(s)

Returns an integer hash of the Bits.

This method is not available for the BitString class, as only immutable objects should be hashed. You typically won't need to call it directly, instead it is used for dictionary keys and in sets.

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

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

__lshift__(*n*)
s << n

Returns the bitstring with its bits shifted *n* places to the left. The *n* right-most bits will become zeros.

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

__mul__(*n*)

__rmul__(*n*)
s * n / n * s

Return bitstring consisting of *n* concatenations of another.

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

__ne__(*bs*)

```
s1 != s2
```

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

__or__(*bs*)

__ror__(*bs*)

```
s1 | s2
```

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

```
>>> print(Bits('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 will be given.

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

__rshift__(*n*)

```
s >> n
```

Returns the bitstring with its bits shifted *n* places to the right. The *n* left-most bits will become zeros.

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

__str__()

```
print(s)
```

Used to print a representation of of the bitstring, trying to be as brief as possible.

If the bitstring 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 = Bits('0b1')*7
>>> print(s)
0b1111111
>>> print(s + '0b1')
0xff
```

__xor__(*bs*)

__rxor__(*bs*)
s1 ^ s2

Returns the bit-wise XOR between two bitstrings, which must have the same length otherwise a ValueError is raised.

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

6.3 The BitString class

class BitString()
The Bits class is the base class for BitString and so (with the exception of Bits.__hash__) all of its methods are also available for BitString objects. The initialiser is also the same as for Bits and so won't be repeated here.

A BitString is a mutable Bits, and so the one thing all of the methods listed here have in common is that they can modify the contents of the bitstring.

append(*bs*)
Join a BitString to the end of the current BitString.

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

byteswap(*format, [start, end, repeat=True]*)
Change the endianness of the BitString in-place according to the *format*. Return the number of swaps done.

The *format* can be an integer, an iterable of integers or a compact format string similar to those used in pack (described in *Compact format strings*). It gives a pattern of byte sizes to use to swap the endianness of the BitString. Note that if you use a compact format string then the endianness identifier (<, > or @) is not needed, and if present it will be ignored.

start and *end* optionally give a slice to apply the transformation to (it defaults to the whole BitString). If *repeat* is True then the byte swapping pattern given by the *format* is repeated in its entirety as many times as possible.

```
>>> s = BitString('0x00112233445566')
>>> s.byteswap(2)
3
>>> s
BitString('0x11003322554466')
>>> s.byteswap('h')
3
>>> s
BitString('0x00112233445566')
>>> s.byteswap([2, 5])
1
>>> s
BitString('0x11006655443322')
```

insert(*bs, [pos]*)
Inserts *bs* at *pos*. After insertion the property pos will be immediately after the inserted bitstring.

The default for *pos* is the current position.

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

invert (*pos*)
Inverts one or many bits from `1` to `0` or vice versa. *pos* can be either a single bit position or an iterable of bit positions. Negative numbers are treated in the same way as slice indices and it will raise `IndexError` if `pos < -s.len` or `pos > s.len`.

overwrite (*bs, [pos]*)
Replaces the contents of the current `BitString` with *bs* at *pos*. After overwriting `pos` will be immediately after the overwritten section.

The default for *pos* is the current position.

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

prepend (*bs*)
Inserts *bs* at the beginning of the current `BitString`.

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

replace (*old, new, [start, end, count, 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. *start* and *end* give the search range and default to `0` and `len` 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
```

reverse (*[start, end]*)
Reverses bits in the `BitString` in-place.

start and *end* give the range and default to `0` and `len` respectively.

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

reversebytes (*[start, end]*)
Reverses bytes in the BitString in-place.

start and *end* give the range and default to 0 and len respectively. Note that *start* and *end* are specified in bits so if end - start 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')
>>> s.reversebytes()
>>> print(s.uintbe)
1234
```

rol (*bits, [start, end]*)
Rotates the contents of the BitString in-place by *bits* bits to the left.

start and *end* define the slice to use and default to 0 and len respectively.

Raises ValueError if bits < 0.

```
>>> s = BitString('0b01000001')
>>> s.rol(2)
>>> s.bin
'0b00000101'
```

ror (*bits, [start, end]*)
Rotates the contents of the BitString in-place by *bits* bits to the right.

start and *end* define the slice to use and default to 0 and len respectively.

Raises ValueError if bits < 0.

set (*pos*)
Sets one or many bits to 1. *pos* can be either a single bit position or an iterable of bit positions. Negative numbers are treated in the same way as slice indices and it will raise IndexError if pos < -s.len or pos > s.len.

Using s.set(x) is considerably more efficent than other equivalent methods such as s[x] = 1, s[x] = "0b1" or s.overwrite('0b1', x).

See also BitString.unset.

```
>>> s = BitString('0x0000')
>>> s.set(-1)
>>> print(s)
0x0001
>>> s.set((0, 4, 5, 7, 9))
>>> s.bin
'0b1000110101000001'
```

unset (*pos*)
Sets one or many bits to 0. *pos* can be either a single bit position or an iterable of bit positions. Negative numbers are treated in the same way as slice indices and it will raise IndexError if pos < -s.len or pos > s.len.

Using `s.unset(x)` is considerably more efficent than other equivalent methods such as `s[x] = 0`, `s[x] = "0b0"` or `s.overwrite('0b0', x)`.

See also `BitString.set`.

__delitem__(*key*)

`del s[start:end:step]`

Deletes the slice specified.

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

__iadd__(*bs*)

`s1 += s2`

Return the result of appending *bs* to the current bitstring.

Note that for `BitString` objects this will be an in-place change, whereas for `Bits` objects using += will not call this method - instead a new object will be created (it is equivalent to a copy and an `Bits.__add__`).

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

__setitem__(*key, value*)

`s1[start:end:step] = s2`

Replaces the slice specified with a new value.

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

6.4 Class properties

Bitstrings use a wide range of properties for getting and setting different interpretations on the binary data, as well as accessing bit lengths and positions.

The different interpretations such as `bin`, `hex`, `uint` etc. are not stored as part of the object, but are calculated as needed. Note that these are only available as 'getters' for `Bits` objects, but can also be 'setters' for the mutable `BitString` objects.

bin

Property for 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. For `BitString` objects it can also be used as a setter, in which case 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'), only for BitStrings, not Bits.
s.bin += '1'
```

bytepos

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.

bytes

Property representing the underlying byte data that contains the bitstring.

For `BitString` objects it can also be set using an ordinary Python string - the length will be adjusted to contain the data.

When used as a getter the bitstring must be a whole number of byte long or a `ValueError` will be raised.

An alternative is to use the `tobytes` method, which will pad with between zero and seven `0` bits to make it byte aligned if needed.

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

hex

Property representing the hexadecimal value of the bitstring.

When used as a getter the value will be preceded by `0x`, which is optional when setting the value of a `BitString`. 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

Property for the signed two's complement integer representation of the bitstring.

When used on a `BitString` 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

Property for the byte-wise big-endian signed two's complement integer representation of the bitstring.

Only valid for whole-byte bitstrings, 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
Property for the byte-wise little-endian signed two's complement integer representation of the bitstring.

Only valid for whole-byte bitstring, 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
Property for the byte-wise native-endian signed two's complement integer representation of the bitstring.

Only valid for whole-byte bitstrings, 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.

float

floatbe
Property for the floating point representation of the bitstring.

The bitstring must be either 32 or 64 bits long to support the floating point interpretations, otherwise a ValueError will be raised.

If the underlying floating point methods on your machine are not IEEE 754 compliant then using the float interpretations is undefined (this is unlikely unless you're on some very unusual hardware).

The float property is bit-wise big-endian, which as all floats must be whole-byte is exactly equivalent to the byte-wise big-endian floatbe.

floatle
Property for the byte-wise little-endian floating point representation of the bitstring.

floatne
Property for the byte-wise native-endian floating point representation of the bitstring.

len

length
Read-only property that give the length of the bitstring in bits (len and length are equivalent).

This is almost equivalent to using the len() built-in function, except that for large bitstrings len() may fail with an OverflowError, whereas the len property continues to work.

oct
Property for 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 of a BitString. If the bitstring is not a multiple of three bits long then getting its octal value will raise a ValueError.

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

pos

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 `len`.

The `pos` and `bitpos` properties are exactly equivalent - you can use whichever you prefer.

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

se

Property for 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

Property for 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

Property for 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

Property for 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

Property for 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

Property for 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.

6.5 Exceptions

exception BitStringError

Used for miscellaneous exceptions where built-in exception classes are not appropriate.

CHAPTER

SEVEN

MODULE FUNCTIONS

pack (*format, [*values, **kwargs]*)

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

Parameters

- *format* – string with comma separated tokens
- *values* – extra values used to contruct the `BitString`
- *kwargs* – a dictionary of token replacements

Return type `BitString`

The format string consists of comma separated tokens of the form `name:length=value`. See the entry for `Bits.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 contained 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 (see *Compact format strings*). 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.

CHAPTER

EIGHT

DEPRECATED METHODS

These methods were all present in the 1.0 release, but have now been deprecated to simplify the API as they have trivial alternatives and offer no extra functionality.

It is likely that they will be removed in version 2.0 so their use is discouraged.

advancebit()
Advances position by 1 bit.

Equivalent to `s.pos += 1`.

advancebits(*bits*)
Advances position by `bits` bits.

Equivalent to `s.pos += bits`.

advancebyte()
Advances position by 8 bits.

Equivalent to `s.pos += 8`.

advancebytes(*bytes*)
Advances position by `8*bytes` bits.

Equivalent to `s.pos += 8*bytes`.

delete(*bits, [pos]*)
Removes `bits` bits from the `BitString` at position `pos`.

Equivalent to `del s[pos:pos+bits]`.

retreatbit()
Retreats position by 1 bit.

Equivalent to `s.pos -= 1`.

retreatbits(*bits*)
Retreats position by `bits` bits.

Equivalent to `s.pos -= bits`.

retreatbyte()
Retreats position by 8 bits.

Equivalent to `s.pos -= 8`.

retreatbytes(*bytes*)
Retreats position by `bytes*8` bits.

Equivalent to `s.pos -= 8*bytes`.

seek(*pos*)
Moves the current position to `pos`.

Equivalent to `s.pos = pos`.

seekbyte(*bytepos*)
Moves the current position to `bytepos`.

Equivalent to `s.bytepos = bytepos`, or `s.pos = bytepos*8`.

slice(*[start, end, step]*)
Returns the `BitString` slice `s[start*step : end*step]`.

It's use is equivalent to using the slice notation `s[start:end:step]`; see `__getitem__` for examples.

tell()
Returns the current bit position.

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

tellbyte()
Returns the current byte position.

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

truncateend(*bits*)
Remove the last `bits` bits from the end of the `BitString`.

Equivalent to `del s[-bits:]`.

truncatestart(*bits*)
Remove the first `bits` bits from the start of the `BitString`.

Equivalent to `del s[:bits]`.

Part III

Appendices

Gathered together here are a few odds and ends that didn't fit well into either the user manual or the reference section. The only unifying theme is that none of them provide any vital knowledge about `bitstring`, and so they can all be safely ignored.

CHAPTER

NINE

EXAMPLES

9.1 Creation

There are lots of ways of creating new bitstrings. 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 bytes property
open('video.m2v', 'wb').write(s.bytes)

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

9.2 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(bytes='hello'))

del s[-1001:]                   # deletes final 1001 bits
s.reverse()                     # reverses whole BitString
s.prepend('uint:12=44')         # prepend a 12 bit integer
```

9.3 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_flags = s.readlist('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_flags)
[0, 0, 0, 1]
```

CHAPTER

TEN

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:

```
00100110110110101100010010 0101
```

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.pos != a.len:
    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.

CHAPTER

ELEVEN

OPTIMISATION TECHNIQUES

The `bistring` module aims to be as fast as reasonably possible, and although there is more work to be done optimising some operations it is currently quite well optimised without resorting to C extensions.

There are however some pointers you should follow to make your code efficient, so if you need things to run faster then this is the section for you.

11.1 Use combined read and interpretation

When parsing a bitstring one way to write code is in the following style:

```
width = s.readbits(12).uint
height = s.readbits(12).uint
flags = s.readbits(4).bin
```

This works fine, but is not very quick. The problem is that the call to `Bits.readbits` constructs and returns a new bitstring, which then has to be interpreted. The new bitstring isn't used for anything else and so creating it is wasted effort. Instead of using `Bits.readbits` (or similar methods) it is better to use a single method that does the read and interpretation together:

```
width = s.read('uint:12')
height = s.read('uint:12')
flags = s.read('bin:4')
```

This is much faster, although not as fast as the combined call:

```
width, height, flags = s.readlist('uint:12, uint:12, bin:4')
```

11.2 Choose between Bits and BitString

If you don't need to modify your bitstring after creation then prefer the immutable `Bits` over the mutable `BitString`. This is typically the case when parsing, or when creating directly from files.

As of version 1.2.0 the speed difference between the classes is marginal, but the speed of `Bits` is expected to improve. There are also memory usage optimisations that can be made if objects are known to be immutable so there should be improvements in this area too.

One anti-pattern to watch out for is using += on a `Bits` object. For example, don't do this:

```
s = Bits()
for i in range(1000):
    s += '0xab'
```

Now this is inefficient for a few reasons, but the one I'm highlighting is that as the immutable bitstring doesn't have an __iadd__ special method the ordinary __add__ gets used instead. In other words `s += '0xab'` gets converted to `s = s + '0xab'`, which creates a new `Bits` from the old on every iteration. This isn't what you'd want or possibly expect. If `s` had been a `BitString` then the addition would have been done in-place, and have been much more efficient.

11.3 Use dedicated functions for bit setting and checking

If you need to set or check individual bits then there are special functions for this. For example one way to set bits would be:

```
s = BitString(1000)
for p in [14, 34, 501]:
    s[p] = '0b1'
```

This creates a 1000 bit bitstring and sets three of the bits to '1'. Unfortunately the crucial line spends most of its time creating a new bitstring from the '0b1' string. It is much faster (and I mean at least an order of magnitude) to use the `BitString.set` method:

```
s = BitString(1000)
s.set([14, 34, 501])
```

As well as `BitString.set`, `BitString.unset` and `BitString.invert` there are also checking methods `Bits.allset` and `Bits.allunset`. So rather than using

```
if s[100] == '0b1' and s[200] == '0b1':
    do_something()
```

it's much better to say

```
if s.allset((100, 200)):
    do_something()
```

CHAPTER

TWELVE

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 section 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 : bytes = '\x01\xff\x00', offset = 0, len = 24
b : bytes = '\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 a `bytearray` 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 `bytearray`, in the `MemArray`, in the `_datastore`, in the `BitString` (but that seemed a bit much to give you in one go).